Text Summarization Techniques Applied to Source Code Summary Generation and Evaluation

Jairo Hernán Aponte Melo

Universidad Nacional de Colombia
Facultad de Ingeniería, Departamento de Ingeniería de Sistemas e Industrial
Bogotá, Colombia
2012
Text Summarization Techniques
Applied to Source Code Summary Generation and Evaluation

Jairo Hernán Aponte Melo

Tesis presentada como requisito parcial para optar al título de:
Doctor en Ingeniería - Sistemas y Computación

Director:
Ph.D. Andrian Marcus

Línea de Investigación:
Mantenimiento y Evolución de Software
Grupo de Investigación:
Colectivo de Investigación en Ingeniería de Software - CoISWE

Universidad Nacional de Colombia
Facultad de Ingeniería, Departamento de Ingeniería de Sistemas e Industrial
Bogotá, Colombia
2012
DEDICATION

To my mom, Mercedes, and my little family: Luz and Ana María
Acknowledgments

First of all, I would like to express deep gratitude to my advisor, Dr. Andrian Marcus, who has been my greatest support during these years. He allowed me the opportunity to tackle the exciting project of conducting empirical research in software engineering. Without his experience and advice it would have been impossible to achieve the goals of this research work. I have learned a lot from him about diverse aspects of research.

A huge thanks to Sonia Haiduc for working with me throughout this research journey. From the first minute until today she has been a wonderful coworker and an unconditional friend. I appreciate the extra effort she has put in every project we have undertaken. I was lucky enough to have her permanent help all this time.

I would like to thank also Laura Moreno for her contributions to my work. She started as one of the participants of our summarization experiments, but she gradually became more involved in this research project. Special thanks to her for having built the tools for automatic summarization of Java classes, which certainly are a concrete proof of the feasibility of using summarization techniques for describing software artifacts.

I was privileged to have collaborated with Sebastiano Panichella, a student at the University of Sannio, and his advisor, Dr. Massimiliano Di Penta, exploring an approach for extracting summaries from developers’ communications. So many thanks to them.

I am also very thankful to all the members of the ColSWE and SEVERE research group. In particular, I greatly appreciated the contribution of Fabio Navarrete, Christian Rodríguez, Oscar Chaparro, and Mario Linares. They collaborated in many ways and deserve special mention for their tireless effort during this endeavor.

I also greatly appreciate the service of the dissertation committee members: Dr. Silvia Takahashi, Dr. Massimiliano Di Penta, and Dr. Carlos Mario Zapata. Their observations and comments on my initial proposal were an invaluable and timely feedback for this research project.
Especial thanks to the helpful steering group and staff at the School of Engineering, including Dr. Diego Hernandez, Dr. Hugo Herrera, Dr. Luis Fernando Niño, Dr. Fabio González, and Dr. Germán Hernández.

Finally, I thank to my sister, Yolanda Aponte, and my mother-in-law, Matilde Ospina, who helped me with the financial support required by the University.
Resumen

Esta tesis explora el uso de tecnologías de resumen de textos para la generación automática de descripciones de artefactos de código fuente. En primer lugar, la tesis reporta los resultados de estudios empíricos que tuvieron como propósito investigar cómo los programadores resumen artefactos de código, durante el proceso de compresión de los mismos. Estos resultados son útiles para explicar cómo los programadores sintetizan trozos de código fuente, para obtener conocimiento acerca del proceso de comprensión, para explicar el rol de las partes del artefacto en ese proceso, y para el desarrollo de herramientas de resumen automático de código fuente. En segundo lugar, este trabajo de tesis describe varios métodos para la creación automática de descripciones textuales, cortas y precisas, para varios tipos de artefactos de código. Los resultados de las evaluaciones indican que las técnicas de resumen de texto son adecuadas para el resumen automático de código fuente, teniendo en cuenta que los desarrolladores generalmente están de acuerdo con los resúmenes producidos mediante los métodos descritos. En consecuencia, estos resúmenes pueden ser útiles para mejorar los procesos de comprensión de software que usualmente ocurren cuando se realizan tareas de mantenimiento de software.

Palabras clave: mantenimiento de software, resumen de código fuente, comprensión de programas, documentación automática de software, investigación empírica
Abstract

This dissertation explores the use of text summarization technology for generating automatic descriptions of source code artifacts. Firstly, the thesis reports the results of empirical studies aimed at investigating how developers summarize code artifacts when understanding them. These results are useful in explaining how developers abstract source code, in gaining insights about the comprehension process, in explaining the role of code elements in such a process, and for the development of automatic source code summarization tools. Secondly, the dissertation describes several approaches for creating short and accurate textual descriptions for various types of code entities. The results of the evaluations indicate that text summarization techniques are suitable for automatic source code summarization since developers generally agree with the summaries produced by the proposed methods. Thus, these summaries can be useful for improving software comprehension processes, which usually occur during software maintenance tasks.

Keywords: software maintenance, source code summarization, program comprehension, automatic software documentation, empirical research
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resumen</td>
<td>VI</td>
</tr>
<tr>
<td>Abstract</td>
<td>VII</td>
</tr>
<tr>
<td>List of figures</td>
<td>1</td>
</tr>
<tr>
<td>List of tables</td>
<td>2</td>
</tr>
<tr>
<td><strong>1. Introduction</strong></td>
<td>5</td>
</tr>
<tr>
<td>1.1 Motivation and problem addressed</td>
<td>7</td>
</tr>
<tr>
<td>1.2 Contributions</td>
<td>9</td>
</tr>
<tr>
<td>1.3 Dissertation organization</td>
<td>10</td>
</tr>
<tr>
<td>1.4 Bibliographical notes</td>
<td>11</td>
</tr>
<tr>
<td><strong>2. Background on Text Summarization</strong></td>
<td>13</td>
</tr>
<tr>
<td>2.1 The concept of summary and its attributes</td>
<td>13</td>
</tr>
<tr>
<td>2.2 External factors and types of summaries</td>
<td>17</td>
</tr>
<tr>
<td>2.3 Evaluation measures and types of evaluations</td>
<td>18</td>
</tr>
<tr>
<td>2.3.1 Intrinsic evaluation</td>
<td>19</td>
</tr>
<tr>
<td>2.3.2 Extrinsic evaluation</td>
<td>20</td>
</tr>
<tr>
<td>2.3.3 Measures for summarization evaluation</td>
<td>22</td>
</tr>
<tr>
<td><strong>3. Related Research in Software Artifacts Summarization</strong></td>
<td>25</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>25</td>
</tr>
<tr>
<td>3.2 Summarizing documentation</td>
<td>28</td>
</tr>
<tr>
<td>3.3 Summarizing source code</td>
<td>29</td>
</tr>
<tr>
<td>3.3.1 Static approaches</td>
<td>30</td>
</tr>
<tr>
<td>3.3.2 Dynamic approaches</td>
<td>35</td>
</tr>
<tr>
<td>3.3.3 Static and Dynamic approaches</td>
<td>37</td>
</tr>
<tr>
<td>3.4 Multi-artifact approaches</td>
<td>38</td>
</tr>
<tr>
<td><strong>4. Human vs. Automatic Summarization: A Exploratory Case Study</strong></td>
<td>41</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>41</td>
</tr>
<tr>
<td>4.2 Generation of automatic summaries</td>
<td>41</td>
</tr>
<tr>
<td>4.3 Study definition and design</td>
<td>42</td>
</tr>
<tr>
<td>4.3.1 Research questions</td>
<td>43</td>
</tr>
<tr>
<td>4.3.2 Context of the study</td>
<td>43</td>
</tr>
<tr>
<td>4.3.3 Experimental design and procedure</td>
<td>44</td>
</tr>
<tr>
<td>4.3.4 Results and discussion</td>
<td>45</td>
</tr>
<tr>
<td>4.3.5 Threats to validity</td>
<td>50</td>
</tr>
<tr>
<td>4.4 Conclusions</td>
<td>51</td>
</tr>
<tr>
<td><strong>5. Human Summarization of Source Code Artifacts</strong></td>
<td>53</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>53</td>
</tr>
<tr>
<td>5.2 Study definition and design</td>
<td>55</td>
</tr>
<tr>
<td>5.2.1 Description of the context of the study</td>
<td>55</td>
</tr>
<tr>
<td>5.2.2 Research questions</td>
<td>57</td>
</tr>
</tbody>
</table>
List of figures

Figure 2-1: Taxonomy of summary evaluation measures (Adapter from (Steinberger and Ježek, 2009)) ............................................................ 22
Figure 3-1 Source code with meaningful identifiers ........................................ 29
Figure 3-2 Source code with meaningless identifiers ........................................... 29
Figure 3-3: Summary generation process for methods (Adapted from (Sridhara, et al., 2010)) ................................................................. 32
Figure 3-4: Corpus creation process followed by ((Haiduc, et al., 2010)) .............. 33
Figure 3-5 A term-based summary for the method setLanguage ......................... 33
Figure 3-6: Summary generation process for software changes (Adapted from (Buse and Weimer, 2010)) ............................................... 37
Figure 3-7: Summary generation process for failed tests (Taken from (Zhang, et al., 2011)) ................................................................. 38
Figure 5-1: Median percentage of different parts of speech for different kinds of artifacts (sentence-based summaries) ............................................. 66
Figure 5-2: Median percentage of different parts of speech for different kinds of artifacts (term-based summaries) ........................................... 66
Figure 6-1: automatic summary for the class MPlayerHandler and a fragment of its declaration ........................................................................ 112
Figure 6-2: Steps of the proposed mining approach ........................................... 126
Figure 6-3: Precision and method coverage for different levels of similarity ......... 135
List of tables

Table 2-1: External factors that affect summarization (Adapted from (Jones, 2007)) .......................................................... 18
Table 3-1: Software summarization approaches (Adapted from (Moreno and Aponte, 2012)) ................................................................. 26
Table 4-1: Features of the methods selected for the study .......................................................... 43
Table 4-2: Collected summaries for the method navigateDir(File dir) ................................................................. 45
Table 4-3: Pyramid score computation for the automatic summary of the method showCdInfo .................................................................................. 46
Table 4-4: The pyramid scores for human and automatic summaries .................................................................................. 47
Table 4-5: Size of the intersection between terms chosen and words in the descriptions ................................................................. 49
Table 4-6: Jaccard index between pairs of developers for the method showCdInfo ................................................................. 50
Table 5-1: Characteristics of the methods chosen for the study .................................................................................. 56
Table 5-2: Characteristics of the classes chosen for the study .................................................................................. 56
Table 5-3: Characteristics of the method sequences chosen for the study .................................................................................. 56
Table 5-4: Characteristics of the packages chosen for the study .................................................................................. 57
Table 5-5: Descriptive statistics about the length in the sentence-based and term-based summaries .................................................................................. 63
Table 5-6: Descriptive statistics of the percentage of sentence-based summaries terms present in the source code artifact .................................................................................. 64
Table 5-7: Examples of sentence-based and term-based summaries .................................................................................. 64
Table 5-8: Percentages of term-based summaries containing at least one term belonging to a given code location .................................................................................. 68
Table 5-9: Ranking of parts of source code according to the importance for summarizing: (a) a method, and (b) a class .................................................................................. 71
Table 5-10: Ranking of parts of source code according to the importance for summarizing (a) a method sequence, and (b) a package .................................................................................. 72
Table 5-11: Gold standard summary for the method ripTracks .................................................................................. 80
Table 5-12: Relevance of terms in gold standard summaries and intersection with sentence-based summaries .................................................................................. 80
Table 6-1: The 24 highest ranked types of summaries .................................................................................. 92
Table 6-2: The average number of relevant terms in lead and VSM summaries, their union and intersection .................................................................................. 94
Table 6-3: The Lead+VSM summaries and their scores .................................................................................. 95
Table 6-4: Average scores for different numbers of terms .................................................................................. 97
Table 6-5: Average scores for different weights .................................................................................. 99
Table 6-6: Average agreement between all pairs of developers .................................................................................. 100
Table 6-7: The number and the ratio of relevant terms in Lead, VSM and Lead+VSM summaries .................................................................................. 101
Table 6-8: Class stereotype taxonomy (Adapted from (Dragan, et al., 2010)) .................................................................................. 107
Table 6-9: Questions and possible answers used in the evaluation study .................................................................................. 114
Table 6-10: Distribution of participant responses .................................................................................. 117
Table 6-11: Distribution of participant responses per each stereotype .................................................................................. 118
Table 6-12 Average agreement among evaluators.................................123
Table 6-13 Characteristics of the two subject systems ................................................131
Table 6-14: Number of paragraphs and method coverage after applying filtering from steps 3, 4, and 5 of the approach ........................................................................................................135
Table 6-15: Examples of true positive paragraphs for Eclipse .........................139
Table 6-16: Examples of true positive paragraphs for Lucene ............................139
Table 6-17: Examples of false positive paragraphs for Eclipse ..........................140
Table 6-18: Examples of false positive paragraphs for Lucene ..........................141
Table 6-19: Examples of false negative paragraphs for Eclipse .......................141
Table 6-20: Examples of false negative paragraphs for Lucene ..........................142
1. Introduction

The complexity and size of modern software systems are the causes of many types of managerial and technical difficulties. At managerial level for instance, as software projects grow larger, the difficulty of communications among team members increases, and also the likelihood of cost overruns and schedule delays. From a technical standpoint, the increased complexity and size of software applications substantially raise the number of product flaws and the difficulty of making the code reliable and maintainable (Banker, et al., 1993; Brooks, 1987).

In recent decades, as a result of the efforts devoted to control software costs and mitigating the problems mentioned above, many tools and techniques have been designed to make software development as rapid and inexpensive as possible. In recent years, however, the focus of these efforts has shifted from the initial development phase to the maintenance stage, since the costs and difficulties have moved from the early steps of the system lifespan to its subsequent maintenance and enhancement (Alkhatib, 1992; Nosek and Palvia, 1990). Thus, software systems that are relatively inexpensive to develop but difficult to maintain are not unusual.

After a long evolution, a software system invariably accumulates a long series of changes due to factors such as the implementation of new functionalities, defect corrections, and design enhancements (Boehm, 2010). Often, in this evolutionary process various development teams are involved, and sometimes they use different paradigms, maintenance processes, and tools. For the current development team, understanding and modifying such a system is usually a big intellectual and engineering challenge, where the loss of knowledge about the system implementation and its domain is one of the most critical obstacles. This phenomenon is often corroborated by the fact that the documentation is rarely updated in synch with the code and other artifacts.
Since the gradual change of existing artifacts is the core of the evolution of modern software systems, a great effort has been devoted by software engineering researchers to understand the cognitive processes behind reading and understanding software artifacts made by others. This is because software comprehension is crucial in a number of maintenance tasks such as concept location (Rajlich and Wilde, 2002), impact analysis (Arnold, 1996), change propagation and refactoring (Fowler, et al., 1999; Rajlich, 2006; Rajlich and Gosavi, 2004). From a practical point of view, it is often observed that developers have difficulties coping with the increasing size and complexity of software systems, when they are searching and browsing large parts of source code while conducting maintenance activities (Corbi, 1989). For instance, during concept location, developers have to determine the parts of the system that they need to modify in order to meet a change request or a bug report. Similarly, during the impact analysis stage, developers require the identification of all software components that will be affected by an incremental change, since often seemingly small modifications can ripple throughout the system to cause major unintended impacts elsewhere. In both cases, at least a partial understanding of the involved software artifacts is needed.

As a consequence, it is widely accepted that the effort and time spent understanding parts of a software system are a significant proportion of the resources needed to maintain existing code (Corbi, 1989; Sneed, 1995). Researchers in program comprehension have proposed a wide variety of cognitive theories about how developers understand source code and, based on them, they have elicited specific requirements for tools that support comprehension processes, which are frequently used to create mental representations of various levels of abstraction (Buckner, et al., 2005; Shaft and Vessey, 2006; Storey, 2006; von Mayrhauser and Vans, 1995; Von Mayrhauser and Vans, 1996). These requirements include features such as documenting domain knowledge concepts, browsing from high-level abstractions or concepts to low-level details and vice versa, searching of code snippets, and allowing the developer to query on the role of source code artifacts.

Additionally, early work on program comprehension and mental models also highlighted the importance of the textual information in capturing and representing knowledge about the system and its domain (Biggerstaff, et al., 1993; Boyd, 1999; Pennington, 1987). For instance, the identifiers chosen by programmers as names for files, packages, classes, methods, attributes, etc. contain valuable information (Antoniol, et al., 2007; Caprile and
Tonella, 1999; Lawrie, et al., 2006; Takang, et al., 1996) and account approximately for 70% of the source code of a software system (Deissenbock and Pizka, 2005). These names often serve as a starting point in many program comprehension tasks, and therefore, it is widely accepted that naming conventions play a fundamental role within software development and evolution. For example, the Java community encourages developers to follow naming conventions in an attempt to self-document software artifacts, and hence make their artifacts easier to understand by themselves and for other developers. In the specific case of Java code, readability is considered important because it means that less time and effort are spent trying to figure out what the code does.

On the other hand, several field studies have shown that source code comments have a significant impact on programmers’ ability to understand software systems (Takang, et al., 1996; Woodfield, et al., 1981). Comments might contain valuable information, including programmers’ assumptions and intentions, code’s purpose, design decisions, authors’ information, etc. Although the usefulness of source code comments is unquestionable, more often than not, there is a lack of good comments within source code artifacts. What often happens is that, as software evolves, developers forget to write comments or keep them up to date, as well as, the external documentation of the system (Feilkas, et al., 2009). Outdated documentation and comments, no longer consistent with the source code, have been pointed out as a latent source of new bugs, since future developments could be based on incorrect or confusing information.

1.1 Motivation and problem addressed

This thesis focuses on the study and automatic generation of descriptions of source code artifacts. We argue that offering descriptions of code artifacts to developers when they are searching, browsing or navigating the code might reduce the amount of code they need to read, and therefore, decrease the time and effort required to perform a wide variety of maintenance tasks. In this context, such a description or summary is a shortened version of the original chunk of source code, and its main intent is to highlight the major points or relevant content units from the original (much longer) artifact in order to help developers to quickly get the gist of it. For instance, by using these descriptions, developers could go through a list of search results fast and would make more informed decisions on which parts of the source code they need to analyze in detail.
Since source code summarization is a new but promising technique in the field of software comprehension, there are many aspects and questions that are worth to investigate. For instance, (i) how useful are source code summaries for software comprehension?, (ii) what would be the desirable features of these summaries and how we can assess them?, (iii) how do people summarize software artifacts?, (iv) is it feasible to generate summaries automatically?, (v) which are the cognitive processes that underlie their generation?, among others. The aim of this work is to establish the foundations of the generation and evaluation of source code summaries.

Specifically, this research examines the automatic and manual generation of source code summaries, the evaluation of their intrinsic quality characteristics (Hariharan and Srinivasan, 2010), and the cognitive processes that underlie their generation. The approach is inspired on automated text summarization (Baxendale, 1958; Luhn, 1958), which is the creation of a shortened version of a text by a computer program (Brandow, et al., 1995; Edmundson, 1969). The outcome of this procedure should still contain the most important points of the original text, and it can be used for different purposes. Summarization technology has been extensively used in text analytics and is broadly applied today by internet search engines (Heng-Yao, et al., 2008; Steinberger, et al., 2008). However, its use in the software engineering field has been limited to a few applications (see chapter 3).

One of the fundamental problems addressed by this research is to determine what should be the content and structure of such a source code summary. In the proposed scenario, software artifacts are documents characterized by complexity, heterogeneity, formal grammatical structure; and often they include natural language sentences or clauses that keep part of the meaning of the document. Therefore, the research includes a study of how various types of source code entities can be summarized, such as methods, classes, packages, and chains of interdependent methods.

A pair of problems closely related to the generation of summaries is the assessment of the agreement among summarizers, and the design of measures for summary evaluation that reduce the effect of the subjectivity of human evaluators (Jing, et al., 1998). Thus, the study of co-selection and content-based measures used to evaluate text summaries was
also addressed by this research. In this respect, we evaluated which of these measures can be adopted or adapted to source code summary evaluation. In addition, we assessed the level of agreement between human summarizers and compared our results with those obtained in text summarization. In the end, this work points out advantages and disadvantages of various summary evaluation methods and inter-human agreement measures.

1.2 Contributions

The contributions of this dissertation can be grouped into two conceptual areas: the study of human generated summaries and the generation of automatic summaries.

To the best of our knowledge this is the first work in software engineering that addresses the study of how humans summarize source code. By performing empirical studies aimed at using developers’ expertise, we uncovered important information related to the length of the summaries generated by developers; the parts of the source code artifacts that convey relevant information and should be included in automatic summaries; the relationships between the artifacts’ terms and the words used in natural language descriptions of their purpose; and the parts of speech often used when summarizing various types of artifacts. The observations that we made can be used as building blocks to define guidelines aimed at explaining to developers how to describe code artifacts when communicating with project newcomers, and also, heuristics to be used in the design or improvement of automatic code summary generators. Additionally, our results can be used to explain how developers abstract source code, thus helping to gain insights about the comprehension process and the role played by code elements in such a process.

Moreover, considering that source code summarization is emerging as a new research area, it is urgent to define formal evaluation methods that make possible the comparison across systems. However, most of the works in this field have performed ad-hoc, informal evaluations that authors considered appropriate for their particular purposes. So far ours is the first research that proposes the use of human generated summaries of code artifacts as input for generating gold standard summaries, which in turn can be used as
test-bed within the design of intrinsic evaluation methods suitable to formally evaluate and compare the outcomes of automatic summarization systems.

Regarding the automatic generation of source code summaries, we firstly investigated the use of automatic text summarization techniques to generate source code summaries. We proposed and evaluated summarization approaches that combine techniques making use of the position of terms in code artifacts and TR techniques, which often capture the meaning of methods and classes. The evaluation of these approaches included the analysis of the parameters that impact the production of summaries such as the weighting scheme, the length of the summaries, and the split of compound identifiers. As a second type of summarization tool, we proposed a novel technique to automatically generate human readable summaries for Java classes. The performed evaluation shows that the generated summaries are readable and understandable, they are concise, and, in most cases, they are not missing essential information. While the techniques used to generate the summaries are adapted from prior work, this is the first technique that automatically generates natural language summaries of classes. Finally, as an alternative to summary generation, we proposed an approach to automatically extract method descriptions from communications in bug tracking systems and mailing lists. This research proved that developer communications, such as mailing lists and bug reports, contain textual information that can be automatically extracted and used to describe methods from Java source code. It is worth to mention that this is a first approach aimed at mining source code descriptions from external unstructured artifacts, and therefore, it opens a new line of work where other sources of information can be explored, and new approaches for mining descriptions of software artifacts at a higher level of abstraction, such as classes and packages, must be designed.

1.3 Dissertation organization

The rest of the document is organized as follows. Basic concepts of text summarization, required as background material for the other chapters, are presented in chapter 2. A comprehensive survey of the current research and the main applications of the summarization technology in the field of software engineering are included in chapter 3. A preliminary case study, that was our first step towards the utilization of text summarization techniques for generating summaries of source code artifacts, is described in chapter 4.
Several aspects of human summarization are discussed in chapter 5, as well as the results of an empirical study aimed at investigating how developers summarize code artifacts when understanding them. Three different approaches to automatically generate summaries of software artifacts and the evaluation of their outcomes are presented in chapter 6. Finally, we draw conclusions and discuss open issues and future work in chapter 7.

1.4 Bibliographical notes

This section reports the parts of this dissertation that have been previously published. It also outlines the materials that were produced in collaboration with other researchers.

Some portions of chapter 3 were published in (Moreno and Aponte, 2012), a chapter of a research book edited by Editorial Universidad Nacional de Colombia and produced by members of the Software Engineering Research Group – CoSWE (Aponte, et al., 2012) that focuses on current research trends in software evolution and maintenance.

The material in chapter 4 is partially based on the results reported in (Haiduc, et al., 2010) and presented at the ACM/IEEE 32nd International Conference on Software Engineering, within the New Ideas and Emerging Results (NIER) Track.

Some portions of the material presented in chapter 5 are based on (Moreno and Aponte, 2011), a paper presented at the XXXVII Latin American Conference of Informatics, CLEI, 2011, that won the Best Paper Award at this conference. This paper was extended as a journal paper in (Moreno and Aponte, 2012) and published in a special issue of best papers presented at CLEI 2011. Thus, chapter 5 reflects these two publications. Additionally, the evaluation of summaries contains portions of (Moreno, et al., 2011). An extended version of the case study presented in this chapter has been submitted to the Empirical Software Engineering Journal, EMSE.

Section 6.2 includes some material published in (Haiduc, et al., 2010), a work presented at the 17th Working Conference on Reverse Engineering, WCRE 2010. Furthermore, section 6.3 describes a work submitted to the 35th International Conference on Software Engineering, ICSE 2013). Also, section 6.4 briefly discusses a research conducted in
collaboration with Dr. Massimiliano Di Penta, Dr. Gerardo Canfora, and Sebastiano Panichella (Panichella, et al., 2012); all of them are researchers at the University of Sannio.
2. Background on Text Summarization

This chapter presents a theoretical background on text summarization. Instead of being a complete state of the art about this area of knowledge, this chapter summarizes the concepts needed for contextualizing the conducted research, and better understanding of the next chapters which describe summarization of software artifacts. For an exhaustive and updated literature review of text summarization, there are several references fairly complete (Jones, 2007; Steinberger and Ježek, 2009; Suneetha, 2011).

2.1 The concept of summary and its attributes

As a starting point, one can simply say that a summary is a text or list of sentences produced from one or more documents that presents the main points in a concise form. Before giving more elaborated definitions of the term summary, it is worth to mention two examples of summaries very common and useful in everyday life, although they come from two completely different sources. First, the abstract of a scientific paper is a short description of it, which contains all information needed by the reader to understand the objectives of the research, how it was conducted, and what were the results and their significance. Frequently, readers use the abstract to quickly decide whether or not it is worth making a more detailed reading of the paper. That is why many computer services make this section of a paper available to the readers in advance, and also, the literature on academic writing emphasizes the guidelines for writing up an abstract in order to have the greatest impact in as few words as possible. As a second example, in the domain of news articles, news generators\(^1\) have gained interest since 1990’s, and are now very popular tools for people in general and journalists in particular. Given a topic, they select the most significant information from a number of publications and display a title and the lead paragraph or the most relevant sentences. Readers can use these summaries just to

\(^1\) [https://news.google.com/](https://news.google.com/)
get superficial information about a topic, or to identify the most suitable publication for their particular purposes.

In the context of (human) text summarization, as the first example above suggests, the purpose of a summary is to give a reader an abbreviated and objective account of the main ideas of a text. Usually, “a summary has between one and three paragraphs or one hundred to three hundred words, depending on the length and complexity of the original essay and the intended audience and purpose” (Reid, 2003).

In regard to human produced summaries, many other definitions of the word summary have been given in the literature and each one emphasizes a particular list of required or desirable features for a summary. For instance:

- “an abstract summarizes the essential contents of a particular knowledge record, and it is a true surrogate of the document” (Cleveland, 1983),
- “an abbreviated, accurate representation of the content of a document preferably prepared by its author(s) for publication with it. Such abstracts are also useful in access publications and machine-readable databases” (ANSI, 1979),
- “the primary function of abstracts is to indicate and predict the structure and content of the text” (van Dijk, 1977),
- “the abstract is a time saving device that can be used to find a particular part of the article without reading it; [...] knowing the structure in advance will help the reader to get into the article; [...] as a summary of the article, it can serve as a review, or as a clue to the content”. Also, an abstract gives “an exact and concise knowledge of the total content of the very much more lengthy original, a factual summary which is both an elaboration of the title and a condensation of the report [...] if comprehensive enough, it might replace reading the article for some purposes” (Graetz, 1985).

Within the field of automatic text summarization, there are less ambitious definitions that convey the idea of automatable processes. For instance:
Background on Text Summarization

- “The main goal of a summary is to present the main ideas in a document in less space [...] information content in a document appears in bursts, and one can therefore distinguish between more and less informative segments. Identifying the informative segments at the expense of the rest is the main challenge in summarization” (Radev, et al., 2002),
- “a summary is a concise representation of a document’s content to enable the reader to determine its relevance to a specific information” (Johnson, 1995),
- “a summary is a text produced from one or more texts, that contains a significant portion of the information in the original text(s), and is not longer than half of the original text(s)”. (Hovy, 2003),
- “a reductive transformation of source text to summary text through content reduction by selection and/or generalization on what is important in the source.” (Sparck Jones, 1999)

Despite the heterogeneity of all of these definitions, two internal attributes of a summary are present in most of them, one related to the length and the other, to the content. Regarding the first attribute, it is required the summary to be short, although there is no agreement on how much. In some cases, absolute limits have been established. For example, an ANSI standard recommended 250 words (ANSI, 1979), and similarly, some publishers of scientific papers impose a limit for the length of an abstract. In other cases, researchers pointed out that imposing an arbitrary limit on summary’s length is not suitable for their quality, but that a length of around 10% is usually enough. In a similar way, E. Hovy, a researcher in human language processing technology, suggested that the length of the summary should be kept less than half of the source’s size (Hovy, 2003). Other authors point out that the summary length should be independent from the length of the original document (Goldstein, et al., 1999). In experiments of relevance assessment tasks on news articles, was found that summaries as short as 17% of full text length speed up decision-making by almost a factor of 2 with no statistically significant degradation in accuracy (Mani, et al., 2002).

Regarding the content, the summary should preserve important information. Accurately representing the main ideas, while omitting the less important details, is one of the major goals of a summary generator or writer. The rationale and also the importance of this
requirement is that no one has the time or is able to read everything, but often people have to make important decisions based only on what they assimilate after a quick and fragmentary reading. The technology of automatic text summarization is becoming indispensable for dealing with this problem (Mani and Maybury, 1999).

There are several features that are all related to the content of a summary:

- **Accuracy**: it must contain only information that is in the original document. An exception is a *critical summary* which discusses and criticizes the content of the original document. This kind of summary may contain opinions and new ideas related to the content of the original text.

- **Conciseness or Succinctness**: it must be concise and terse. It should express a great deal in just a few words. It should be a concise representation of a document’s content. Another term that refers to this feature is *redundancy*, which is used in a negative sense to express that a summary contains superfluous and unneeded information.

- **Informativeness**: it must serve to inform or provide information contained in the original text. Depending on how much information the summary provides, it can be:
  - *Indicative*: it should predict the structure and content of the original document. In this case, the summary can serve as a review or as a clue to the content of the document. After reading the summary, the reader should be able to decide the relevance of the document to a specific information or purpose.
  - *Informative*: it should be a true surrogate of the original document. In this case, the summary might replace the reading the document, at least for some purposes.

- **Coherence**: this feature refers to the fact that there should be a logical, orderly and consistent relationship among the words, clauses and sentences of the summary.
Since summarization is not context free, the performance of a summarization system and the quality of generated summaries cannot be assessed based only on the intrinsic or internal features of them, mentioned in the summary definitions. It is essential to consider external factors that together determine the setup within which a summary is used. Thus, considering all the internal and external aspects highlighted above, text summarization “is the process of distilling the most important information from a source (or sources) to produce an abridged version for a particular user (or users) and task (or tasks)”. (Mani and Maybury, 1999).

2.2 External factors and types of summaries

Sparck Jones (Sparck Jones, 1999; Sparck Jones, 2001) defined and classified these external factors that must be considered when designing and evaluating summarization systems. They are divided in three major categories (Table 2-1):

- **Input factors** are the characteristics of the original document(s) and their intended readers that may affect the summarization approach. In regard to the original documents, the most remarkable input factors are the language (monolingual vs. multilingual), the register or linguistic style, the genre, and the units or size of source(s). This last one determines if the input comprises single or multiple documents, and whether the elimination of content redundancy (or repetitive information) becomes a key issue for the summarizing techniques. Regarding the potential readers of the original documents, their background knowledge may influence the amount of contextual information present in the originals.

- **Purpose factors** are specific characteristics of the intended use and audience of the summaries. Thus, the way in which the summaries are going to be used and perhaps some characteristics of the people who will use them may affect the summarization approach and its evaluation method. For instance, a summary is evaluative if it points out the opinion of the author on a particular subject; it is indicative if it helps the reader to decide whether the original
document is worth reading; if the summary completely replaces the reading of original documents, it is informative.

- **Output factors** are requirements to be met by the generated summary. They can refer simply to the output format of the summary (e.g., bullet item list vs. prose paragraph), or they can be related to more sophisticated features such as coherence, reduction, and derivation. According to the derivation characteristic, a summary can be *extractive* if it only contains units extracted from the original document; or *abstractive*, when it includes some units which are not in the original (Nenkova, et al., 2007). Thus, producing abstracts is harder because they represent additional challenges involving analysis, topic fusion and generation of natural language (Murphy, et al., 2001).

### Table 2-1: External factors that affect summarization (Adapted from (Jones, 2007))

<table>
<thead>
<tr>
<th>Context Factors</th>
<th>Type</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
<td>Characteristics of the source material</td>
<td>Language, structure, genre, length, medium, unit</td>
<td></td>
</tr>
<tr>
<td><strong>Purpose</strong></td>
<td>Characteristics of the audience and intended use</td>
<td>Formality, location, time, destination</td>
<td></td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td>Characteristics of the resulting summary</td>
<td>Format, language, structure, medium, length, coverage</td>
<td></td>
</tr>
</tbody>
</table>

### 2.3 Evaluation measures and types of evaluations

Evaluation has been highly important because it allows researchers to assess the results of a summarization approach or system, compare the results of different methods, and identify and understand the drawbacks of a particular summarization process. Nowadays, evaluation is essential for any kind of summarization (Hariharan and Srinivasan, 2010), and therefore, the summarization community has developed and evaluated various measures for use in large-scale evaluation events, e.g., SUMMAC (Mani, et al., 2002) and the Document Understanding Conference\(^2\). The main difficulty for evaluating is that there is no clear idea of what constitutes a good summary. Actually, it is often possible to obtain various perfectly acceptable summaries for the same document. This issue is closely

related to the subjectivity and lack of agreement among people when they read and summarize a text or when they assess the quality of summaries generated by others (Copeck and Szpakowicz, 2004; Mizzaro, 1999).

As a consequence of the diversity of internal and external factors that may impact the suitability of a summarization system and its outcomes, currently, there is not a unique approach to assess how good a summary is. Roughly, there are two types of evaluation according to what they measure: intrinsic and extrinsic evaluation (Mani, 2001). Intrinsic evaluation measures internal properties of the abstracts such as informativeness, coherence, and redundancy, whereas extrinsic evaluation aims to establish whether the summaries are good instruments to support real-user’s work, and therefore, it focuses more on external factors. Regardless of the type of evaluation, two important variables that are often controlled within any assessment process are the compression ratio, i.e., the length of the summary with respect to the original document, and the way the summary is presented to the user. Thus, it is important to be able to evaluate summaries at different compression rates, and also, find a manner for presenting the summaries, suitable to the user’s needs (Goldstein, et al., 2000).

2.3.1 Intrinsic evaluation

When using this kind of evaluation, the quality of a summary can be established mainly with two approaches. In the first alternative, the peer summary, i.e. the summary being evaluated, is reviewed by human judges, using some pre-established guidelines. The second choice is to measure the similarity of the peer abstract to some reference summary given by (experts) humans, which is often called the gold standard reference summary.

When human judges read and assess the peer summary, the results of the evaluation are interpretable as they can explain the rationale behind their opinions and scores. However, there are serious disadvantages because different human judges can reach conflicting conclusions; additionally, the process is time consuming, expensive and cannot be easily repeated. An evaluation which could use a scoring program instead of human judgments is preferable, since it is easily controllable and repeatable (Mani, 2001).
On the other hand, proximity to a gold standard summary is a criterion that can be automated to a high degree, and also, it is often preferred because it is fast and not directly influenced by the human subjectivity. Apart from that, once gold standard reference summaries are available, the resulting test-bed can be used as a stable benchmark to carry out comparative evaluation of automated systems, and also, as a tool for iterative refinement of summarization techniques. However, this second approach often requires procedures to compound different human-generated summaries for creating the ideal gold standard summary of each document (Halteren and Teufel, 2003). In addition, it might be necessary to assess the level of agreement among human summarizers when they select relevant content units or create abstracts for a document, since several studies in the field of text summarization have showed that subjectivity plays an important role, when removing unwanted or redundant information for summarizing a document (Hobson, et al., 2007; Kolluru and Gotoh, 2009; Lin and Hovy, 2003).

In order to perform repeatable, inexpensive, and automatically-scorable intrinsic evaluations, the development of annotated corpora is needed. The annotation is often just an indication of whether each sentence of the original document is important enough to be included in the summary. In more elaborated cases, in addition to indicate that a sentence or clause is relevant, it is necessary to score such relevance using a numerical scale, or rank the sentences in the order of importance. Such corpora can be produced manually or automatically, and is very suitable for evaluating extractive summarization approaches.

### 2.3.2 Extrinsic evaluation

Extrinsic evaluations are aimed at assessing the influence of summarization on tasks like relevance assessment (Oka and Ueda, 2000), reading comprehension (Morris, et al., 1992), following instructions, etc. The main goal of an extrinsic evaluation is to determine the effect of summaries when people perform a particular task. Thus, humans are asked to perform a task using summaries instead of the original documents and the accuracy, or some other properties of this task are measured. The hypothesis is that when the summaries are good, the accuracy does not decrease considerably; while the time spent doing the task is significantly reduced. From a pragmatic standpoint, this kind of
evaluation is considered as the true assessment because it allows researchers to verify whether the summarization technology works in real life.

A variety of tasks have been considered within extrinsic evaluation. For instance, finding documents relevant to a specified subject from a large collection (relevant assessment), classifying documents, executing instructions using only summaries of the respective guidelines, and sometimes, testing the summarizer as part of question answering systems. In all these cases, the influence of summarization on accuracy, effort, and efficiency in the task is carefully studied (Jing, et al., 1998) and in some cases promising results have been reported. For instance, in (Dorr, et al., 2004) it is highlighted that “it is possible to save time using summaries for relevance assessment without adversely impacting the degree of accuracy that would be possible with full documents”. Interestingly, it is also reported a small yet statistically significant correlation between some of the intrinsic measures and a user's performance in an extrinsic task.

In some experiments of extrinsic evaluation researchers have replaced humans in task-based evaluations with automatic artifacts, so that automatic devices perform the same task and are evaluated automatically. This approach, known as automatic extrinsic evaluation, has the advantage of being fast and repeatable (Kuo, et al., 2002). Reported examples of this kind of evaluation are (Brandow, et al., 1995), where the summaries were used for a text retrieval task, and (Kolcz, et al., 2001), where the task was text categorization.

The decision whether to use an extrinsic evaluation method or an intrinsic one depends primarily on the external factors that may affect the setup within which the summaries will be used. Moreover, researchers recommend considering the maturity of the summarization technology at hand. Thus, they suggest that in early stages of the technology development it is better to use intrinsic evaluation that is more suitable to evaluate and incrementally improve the individual components of a summarization system, since they are more useful to developers in terms of offering valuable feedback. When the technology reaches a certain level of maturity and robustness, it is feasible to design more situated, task-based evaluations of the summarization system as a black-box, involving real users (Mani, 2001).
2.3.3 Measures for summarization evaluation

Several approaches have been proposed to assess the quality of the (automatic) summaries either intrinsically, by measuring their inner quality attributes, usually against an ideal gold standard summary written by an expert; or extrinsically by measuring their effectiveness for a given task.

(Steinberger and Ježek, 2009) present a taxonomy of evaluation measures (Figure 2-1) which classifies intrinsic approaches in those that evaluate text quality (e.g. grammaticality, non-redundancy, coherence, etc.), and those that assess summaries' content, by co-selection (precision, recall, etc.) or content itself (cosine similarity, unit overlap, etc.).

Various schemes for evaluating peer summaries are available when gold standard summaries have been formed (Hariharan and Srinivasan, 2010). One is the precision and recall based evaluation, where precision is used to know if a peer summary contains all the sentences of the gold standard summary; recall is used to how many sentences in the gold standard summary are in the peer summary; and additionally, accuracy is used to estimate how many sentences were correctly included in the peer summary and how many were correctly excluded. These measures are binary measures, i.e., all or nothing assessments that give a conservative and pessimistic estimate, and therefore, often fail to bring out the correct evaluation of the peer summary. (Steinberger and Ježek, 2009) point
out that the main problem with these measures is that human judges often disagree on what the top p% most relevant sentences are in a document.

A second group of measures is the utility based evaluation. Within this group, (Radev and Tam, 2003) proposed a content evaluation which, unlike traditional co-selection metrics such as precision or recall, considers the utility score of every sentence of the input document according to each human judge who evaluated the summary. Therefore, this Relative Utility metric reduces the problems of disagreement between judges when choosing the top sentences within a text because it considers the degree of importance of each summary constituent. Thus, the utility based scheme is better choice for evaluation of summaries. The drawback of this evaluation approach is that human judges not only have to choose the sentences for the summary, but give a number to indicate the degree to which each sentence should be part of the summary.

In a similar way, other intrinsic measures avoid the low agreement among human evaluators, by taking into account that two sentences could express the same information even if they are written different. Cosine similarity and unit overlap are examples of these content-based measures, as well as the Pyramid method, proposed in (Nenkova, et al., 2007). This semiautomatic model focuses on determining if a peer summary conveys the same as a set of manual models or a goal standard summary, by weighting content units based on their occurrence in a corpus of manual summaries.

Within extrinsic approaches, the impact of using summaries on several tasks has been assessed. For instance, document classification, question answering, instruction execution, information retrieval and relevance assessments are common tasks where this impact has been evaluated. To exemplify, in a relevance assessment task the usefulness of summaries (i.e. if they provide enough and useful information) is measured when determining the importance of a document to a specific topic. A clear example of it is Relative Prediction introduced in (Hobson, et al., 2007).
3. Related Research in Software Artifacts Summarization

3.1 Introduction

When doing maintenance of a software system, a complete and exhaustive understanding of the entire source code is often not feasible or not required. What happens in practice is that whenever developers need to do maintenance, they must locate and understand the parts of the software system relevant to the task at hand (Lakhotia, 1993). Additionally, detailed studies of developers’ activities have shown that they spend much of their time exploring unfamiliar parts of the software system’s source code (Singer, et al., 1997), searching, browsing, relating and collecting code and information that may be needed to complete the task (Ko, et al., 2006) (LaToza, et al., 2006).

During these time consuming activities, developers use several reading techniques, typical of reading for research purposes, such as scanning, skimming and detailed reading (Starke, et al., 2009), and therefore, they often make decisions based on diverse levels of understanding of the reviewed source code. When scanning, developers just look for specific key terms, and often, they have a good idea of what they want to find. Modern IDEs support scanning by built-in searching tools, highlighting and coloring key words, and indenting source files. Skimming is often used after scanning, when the developer considers that it is worth getting further information, and therefore, it is used to quickly identify the main ideas of a source file (e.g., read only the header of a method and maybe the leading comments when available). When the code is well documented internally (e.g., good preceding comments, meaningful names and parameters), skimming is often sufficient to get the level of knowledge required to perform the task at hand. However, more often than not, good comments are missing and method headers contain words that the developer is not familiar with. In such cases, developers have little choice
but to read the implementation of the artifact and even more than that, other related artifacts. This takes significant effort and time.

Therefore, an important goal of software engineering research has been finding ways - theories and tools - that help developers search, understand and modify source code more effectively. One of the alternatives already explored is text summarization technology (Jones, 2007) since offering developers a description of the source code, which can be read fast and leads to better understanding than scanning and skimming, could allow them to make more informed decisions on which parts of the source code they need to analyze in detail.

Even though summarization is an emerging issue within software engineering field, several approaches have been proposed to reduce the data found in software artifacts or to generate new human readable documentation. In this context, an artifact is defined as any product derived from development process which describes the process, functionality, design, or implementation of software. These derivatives include source code, binaries, and majorly, documentation. Currently, there is at least one summarization work that deals with each one of them separately, but in some cases artifacts are used together for producing more accurate descriptions of their content. Table 3-1 presents a brief description of such approaches.

**Table 3-1: Software summarization approaches (Adapted from (Moreno and Aponte, 2012))**

<table>
<thead>
<tr>
<th>ARTIFACT</th>
<th>APPROACH</th>
<th>INPUT</th>
<th>OUTPUT</th>
<th>EVALUATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documentation</td>
<td>Summarizing bug reports through machine learning classifiers (Rastkar, et al., 2010)</td>
<td>Bug reports</td>
<td>Text-based summaries</td>
<td>Type: intrinsic, online and offline Attributes: informativeness, redundancy, relevance, coherence Measures: co-selection and content-based</td>
</tr>
<tr>
<td>Source code (dynamic approaches)</td>
<td>Summarizing the content of large traces through Routines filtering (Hamou-Lhadj and Lethbridge, 2006)</td>
<td>Traces</td>
<td>UML sequence diagrams</td>
<td>Type: intrinsic, online Attributes: informativeness, effectiveness Measures: human judges' opinions</td>
</tr>
<tr>
<td>ARTIFACT</td>
<td>APPROACH</td>
<td>INPUT</td>
<td>OUTPUT</td>
<td>EVALUATION</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>Source code (static approaches)</td>
<td>Documenting program changes by symbolic execution comparison and natural language processing (Buse and Weimer, 2010)</td>
<td>Source code of two versions of a program</td>
<td>Textual description of program differences on runtime behavior</td>
<td>Type: intrinsic, online and offline Attributes: informativeness, Conciseness Measures: human judges’ opinions, comparison with human-written log messages</td>
</tr>
<tr>
<td></td>
<td>Building reflection models comparing high-level models and source code structural data (Murphy, et al., 2001)</td>
<td>A high-level model, a source model, and a mapping from the source to the high-level model</td>
<td>Reflection models</td>
<td>Type: extrinsic, task-based Tasks: experimental reengineering, design conformance Measures: effectiveness, efficiency</td>
</tr>
<tr>
<td></td>
<td>Identifying topics in source code using information retrieval and map visualization (Kuhn, et al., 2007)</td>
<td>Source code</td>
<td>Semantic clusters</td>
<td>Informal evaluation of the clustering result</td>
</tr>
<tr>
<td></td>
<td>Summarizing source code artifacts applying text retrieval techniques (Haiduc, et al., 2010; Haiduc, et al., 2010)</td>
<td>Source code artifacts</td>
<td>Term-based summaries</td>
<td>Type: intrinsic, online and offline Attributes: informativeness, Conciseness, redundancy Measures: human judges’ opinions, Pyramid</td>
</tr>
<tr>
<td></td>
<td>Summarizing Java methods by relevance heuristics and templates (Sridhara, et al., 2010)</td>
<td>Source code of methods</td>
<td>Text-based summaries</td>
<td>Type: intrinsic, online Attributes: accuracy, content adequacy, conciseness Measures: human judges’ opinions</td>
</tr>
<tr>
<td>Source code (static and dynamic approaches)</td>
<td>Documenting a failed test by running and analyzing several versions of the test (Zhang, et al., 2011)</td>
<td>A failed test</td>
<td>Code comments that provide potentially useful facts about the failure</td>
<td>Type: extrinsic, task-based Tasks: fixing bugs Measures: effectiveness, efficiency</td>
</tr>
<tr>
<td>Documentation + Source Code</td>
<td>Building lexical source models using keyword matching techniques (Murphy, et al., 2001)</td>
<td>Source code Data files Documentation</td>
<td>Lexical source models</td>
<td>Type: extrinsic, task-based Tasks: experimental reengineering, design conformance Measures: effectiveness, efficiency</td>
</tr>
<tr>
<td></td>
<td>Summarizing software concerns applying static analysis, information retrieval and natural language processing techniques (Rastkar, 2010)</td>
<td>Source code Historical repositories Bug repositories Wikis and Documentation</td>
<td>Text-based summaries</td>
<td>None</td>
</tr>
</tbody>
</table>
### 3.2 Summarizing documentation

From a practical standpoint, the approaches that deal with software documentation usually apply natural language summarization techniques on natural language texts, i.e., the traditional text summarization process. An evident approach for summarizing software documentation is the one presented in (Rastkar, et al., 2010), where bug reports are synthesized by machine learning techniques.

An important issue about bug reports is that they often comprise two parts: one with predefined values in fixed fields, and another one with free-form texts such as a title, a bug description, and a sequence of comments related to its lifecycle. In that sense, bug reports are somewhat similar to email conversations, and in consequence, same techniques applied in the latter case might be useful for summarizing the former one. Along these lines, in (Rastkar, et al., 2010) are extracted the most relevant sentences from bug reports based on three conversation-based classifiers trained on structural, participants, length and lexical features, with different corpora: annotated email threads, meetings, and bug reports.

The evaluation of the three methods was performed from several perspectives. For instance, each system was evaluated against a baseline (random) classifier to measure its effectiveness using the ROC curve. Later, systems were compared between them applying the standard intrinsic measures of precision, recall and F-score. Next, Pyramid (Nenkova, et al., 2007), a content-based measure, was used to compare the three classifiers, using the multiple human annotations available for each bug report. Not surprisingly, the classifier trained with the bug report corpus outperforms the other two
systems. The informativeness of each feature considered in the summarizers was also evaluated: the features with the highest F-score were those related to length. Finally, a group of developers were asked to evaluate informativeness, redundancy, relevance, and coherence of summaries, and actually their results were acceptable.

### 3.3 Summarizing source code

When the aim of summarization is describing source code, one crucial issue has to be considered: source code is a mixed artifact which contains information for communicating to both humans (the developers) and machines (the compilers). In (Kuhn, et al., 2007), it is acutely explained this situation by means of an example similar to the next one. Suppose a random chunk of code as the one in Figure 3-1 (extracted from aTunes\(^3\) system, a full-featured audio player and manager):

**Figure 3-1 Source code with meaningful identifiers**

```java
public static void setLanguage(String fileName) {
    if (fileName != null)
        languageBundle = getLanguageFile(TRANSLATIONS_DIR + '/' + fileName);
    else
        languageBundle = getLanguageFile(TRANSLATIONS_DIR + '/' + DEFAULT_LANGUAGE_FILE);
}
```

Program instructions within this method can be interpreted by compilers, even if identifiers are replaced by arbitrary letters (Figure 3-2). Then, functionality is unaltered despite identifiers do not reveal the intention of the code:

**Figure 3-2 Source code with meaningless identifiers**

```java
public static void f(String x) {
    if (x != null)
        y = g(TD + '/' + x);
    else
        y = g(TD + '/' + DLF);
}
```

\(^3\) http://www.atunes.org/
On the other hand, if the information removed from source code is the formal one, i.e., keywords and syntactic tokens, textual information represented by the terms composing identifiers still remains, providing the reader with an idea of the purpose of the code:

```
set language string file name file name language bundle get language file translations dir file name language bundle get language file translations dir default language file
```

Usually source code analysis has been performed statically or dynamically. The elemental difference between them is that static analysis does not involve executing code, whereas dynamic analysis studies the behavior of code during program execution. In these circumstances, dynamic approaches depend on the program input and the program itself (Binkley, 2007).

### 3.3.1 Static approaches

Static approaches usually consider syntactic and semantic properties of source code. The first step in any static approach is commonly tokenization, where composite identifiers are split into words according to capital letters, underscores or other special characters. On this wise, method names such as `cdda2WavFile` or `cdda_2_wav_file` are transformed into `cdda, 2, wav, file`.

The most notable techniques for describing source code statically can be distinguished by the coherence factor that is determined by the output frequency (Hovy and Lin, 1999). This means that a summary is *fluent* if it consists of well-formed sentences that are related to each other forming coherent paragraphs. Otherwise, if it comprises individual words or text fragments which do not keep any relation, the summary is said to be *diffluent*. As an illustration, for the `setLanguage` method previously mentioned (Figure 3-1), an example of a fluent sentence-based summary is:

*This method sets application language. If the file name is defined, it is used; if not, the default language file is applied.*

For the same method, a diffluent term-based summary could be:
Sentence-based summaries

One of the most outstanding approaches of software summarization which deals directly with source code by creating descriptions of Java methods is (Sridhara, et al., 2010). The essentials on this proposal are heuristics, for selecting the central statements of code within a method (\textit{s\_units}), and templates, for generating natural language sentences and reducing redundancy. In such manner, the algorithm applied to obtain the descriptive comments starts as usual with source code preprocessing, which includes tokenization and abbreviation expansion. Then, the \textit{action}, \textit{theme} and \textit{secondary arguments} for methods are obtained using a \textit{Software Word Usage Model (SWUM)} that captures linguistic, structural and occurrence relationships of words within code. Next, some heuristics are applied to identify the most relevant units of code within a method. Five kinds of statements are considered as relevant: \textit{ending units}, \textit{void-return units}, \textit{same-action units}, \textit{data-facilitating units}, and \textit{controlling units}. The relevance and role of these statements were drawn as an inference from a study of a set of comments from open source Java programs, and an opinion survey with Java developers about the need of certain units within methods descriptions. At last, those units are lexicalized from predefined templates. For instance, the fixed template to an assignment is:

\begin{verbatim}
action theme secondary-args and get return-type
\end{verbatim}

Hence the text generated for the following \textit{s\_unit}

\begin{verbatim}
title = getNameWithoutExtension()
\end{verbatim}

is

\begin{quote}
Get name without extension and get title
\end{quote}

Similar kind of templates were designed to variables, single method calls, return statements, nested and composed method calls, conditional expressions and loop expressions. The whole summarizing process is shown in Figure 3-3.
Roughly speaking, this technique produces acceptable summaries for methods, as stated by an informal evaluation where some Java developers were asked to judge the accuracy, content adequacy, and conciseness of text generated for individual s_units and the whole summaries. However, developers disagreed when assessing the level of detail required in summaries, which suggests that selection of statements should be studied carefully. In addition, the proposal is designed for and limited to methods, making it unable to produce that kind of comments at other granularity levels, such as classes.

- **Term-based summaries**
  (Haiduc, et al., 2010) generate (majorly) extractive summaries for Java methods and classes using several approaches based on a combination between techniques making use of the position of terms in software artifacts and text retrieval techniques. Two basic phases are executed in this extent: corpus creation and relevant terms selection. First, a corpus is created from source code artifacts by extracting identifiers and comments, but on this occasion, as a particular case, tokenization is an optional step. Then, the corpus can be composed of split identifiers, original identifiers, or both Figure 3-4.
After filtering out terms which do not carry out a specific meaning, the summaries are generated by selecting the terms with the highest scores, obtained by applying algebraic reduction methods, Vector Space Model (VSM) (Salton, et al., 1974) and Latent Semantic Indexing (LSI) (Deerwester, 1988), with variations in the formulas for weighting terms (e.g., log, tf-idf, binary entropy). Figure 3-5 depicts a term-based summary for the method `setLanguage`, where the terms of the summary are highlighted.

The outcomes of all approaches were compared against random and lead summaries, using intrinsic measures assessed by human evaluators. Random summaries comprise artifacts terms chosen in a haphazard way, whereas lead summaries are built with the first terms of the artifact declaration. The results until this point of the study showed that each variation of lead method outperformed the other techniques regardless the weighting and length options. Although VSM summaries also obtained good scores, they were not as good as lead ones. Nevertheless, it was found that this technique extracts relevant terms from parts of code where the lead method has no effect at all, i.e., other lines
besides the header of the source code artifact. Therefore, those techniques are complementary and their union produces summaries with a greater amount of relevant terms, mainly for methods.

- **Model-based summaries**
  
  Source code descriptions can be represented as well through models such as diagrams, maps or graphs. For instance, in (Murphy, et al., 2001) it is presented the *software reflection model*, where the developer initially selects a high-level task-specific model of the software system, which is used as a framework for summarizing information in the source code. This high-level model is delineated through interconnected boxes that refer to the task at hand, and represent the main modules of the system and the interactions between them. After choosing the high-level model, the developer uses a syntactic analysis tool to extract structural information from the source code, i.e., the methods and the method calls. Next, the user performs a mapping of the high-level model to the entities and relationships extracted from the code. He can use a series of tools to support the task development (like regular expression matching or mapping several source code entities into a single map entry), but the mapping is mostly performed manually. Lastly, the developer computes a reflection model, which is a comparison between the high-level model, the source code structural information and the constructed map, and represents a summary of the structural information contained in the source code of the system.

- **Summaries as by-products**
  
  Other approaches like (Kuhn, et al., 2007; Poshyvanyk and Marcus, 2007) do not have as an explicit aim generating that kind of short descriptions, but still, their results can be considered as (partial) summaries. For example, in (Kuhn, et al., 2007) the main objective was to analyze software without taking into account external documents, in order to provide a first impression of an unfamiliar system. In this case, Latent Semantic Indexing (LSI) was used to compute the similarity between source code artifacts, and then, to describe the topic of clusters of artifacts with labels extracted from the same source code. These labels capture the important concepts within each linguistic cluster, revealing the intention of the code.

By the same token, a combination of LSI and Formal Concept Analysis (FCA) was used by (Poshyvanyk and Marcus, 2007) for indexing source code and organizing the results in
a concept lattice, respectively. The initial aim of this work was the reduction of developers’ effort when searching in source code, providing them with a list of artifacts related to a query entered by the user. Even so, the resulting representation of relevant information, labeling topics, concepts and relationships among them, can be recognized as an approximation to source code summarization.

3.3.2 Dynamic approaches

As stated before, in order to analyze behavioral aspects, there are some methods that require the execution of a program, or at least, one of its slices or traces. In contrast to static approaches, dynamic ones are able to analyze the behavior of variables and control structures, detect data dependencies, and collect and log temporal information. Moreover, dynamic approaches allow developers to observe the flow and behavior of a program under determined conditions.

- **Summaries of large traces**

In (Hamou-Lhadj and Lethbridge, 2006), the most representative software routines of large traces are identified by selection or generalization of executed content. Thus, the summary is a simplification of the original input that is reduced when removing low level implementation details and utilities (through a utilityhood metric), but in this case the obtained description is represented in a UML sequence diagram fashion. By utility, they refer to any software element designed and implemented to be accessed or called from anywhere in source code (e.g. accessing methods or constructors), whereas by implementation detail, refers to any element whose absence does not interfere in the comprehension of a component.

Basically, the algorithm for extracting the relevant routines from a scenario proposed in (Hamou-Lhadj and Lethbridge, 2006), starts by instrumenting source code for logging the method calls. Afterward, a static call graph is built and pruned by removing the low-level implementation details mentioned above. Next, the utilityhood of each remaining node of the graph is computed, based on fan-in and fan-out metrics; and then, the unnecessary routines are removed, i.e., those nodes whose utilityhood was the lowest. This process is repeated until reaching the amount of routines required by the user. Finally, the routines
which are still present in the graph are represented in a light version of a UML sequence diagram, in order to visualize the trace behavior.

The evaluation of this approach was performed informally by analyzing a summary obtained from a particular trace of a program, through a questionnaire answered by developers with intermediate and advanced knowledge of the system. The questions were addressed to assess the quality of the summary by asking about how well its content represented the trace process, and how effective could it be in software maintenance. Again, just as in (Sridhara, et al., 2010), the amount of details included in the summaries and the amount required by the users was a disagreement point. Still, these kind of short descriptions were marked as useful when understanding different scenarios of a system.

- **Summarizing software changes**

Another direction is taken by (Buse and Weimer, 2010), where code changes are documented by describing the effects they have in the behavior of a program. This description, commonly known as commit message in control version systems, is generated automatically when executing a couple of versions of the same system in search of differences between their control flows (delta). As a result, the proposed algorithm considers the new behavior of the system and the conditions under it is produced.

To this end, `<statement, path predicate>` pairs are obtained by running two different versions of the system, using symbolic values as input variables. In this context, the path predicate indicates the conditions under which the statement is executed. By comparing both sets of pairs, the statements whose path predicate have been added, removed or modified, are identified. Of course not all statements become part of the summary: a process of filtering is performed for retaining only method invocations, field assignments, return, and throw statements. Then, some summarization transformations are applied to those statements previously found. For instance, if the first version of a chunk of code is

```java
if (interrupted) deleteFile();
```
and the second one is

```java
if (interrupted) revertProcess();
```

the whole change transformation for conditions expresses the change as

```java
if interrupted, do revertProcess() instead of deleteFile()
```

A similar kind of templates is defined for predicates, hierarchical structures and method calls. Other transformations are applied to reduce the size of the documentation, and some others to improve readability. Major steps of the approach are in Figure 3-6.

Figure 3-6: Summary generation process for software changes (Adapted from (Buse and Weimer, 2010))

The evaluation of this method assessed quantitative features like size and content of the summary. This last one was evaluated through a comparison rubric which contrasted the description of changes with a set of humans' annotations. Additionally, developers considered qualitative features of summaries like usefulness, readability and accuracy. In general, the results were satisfactory, and the summaries could be a complement for control version systems, reducing developers' effort when describing changes.

### 3.3.3 Static and Dynamic approaches

A combination of static and dynamic approaches is presented in (Zhang, et al., 2011) where an automatic technique for summarizing failed tests is presented. This summarization system generates test code comments that help programmers finding the causes of a failure, and therefore, reduces the time and effort required to fix a bug.
Figure 3-7 shows the major modules of the system and stages of the summarization approach. The first step is the generation of mutated tests by repeatedly replacing expressions in the test code. The objects needed for executing the tests are also generated. In the second step, this set of slightly mutated tests is executed, and then, static slicing is used to remove all irrelevant statements from each execution trace obtained. Next, the system identifies suspicious statements (that have a strong correlation with the test failure) analyzing the execution traces and the outcomes obtained in the previous stage. Finally, for each suspicious instruction, the system makes a generalization of the observed properties that relate the instruction with the values used to pass the test successfully. These generalizations are converted into documentation.

The authors evaluated their approach in two ways. Initially, they documented several failed tests of five open source systems and sent these tests to the developers of those systems, asking them if the generated comments would be helpful for analyzing failure causes. They received positive feedback from them revealing that the generated comments are meaningful and concise. After that, they performed a task-based evaluation where a set of developers used their system and an existing tool for understanding and fixing bugs. They measured efficiency and effectiveness of the generated abstracts for these maintenance tasks. The results showed that the approach is useful for bug diagnosis.

### 3.4 Multi-artifact approaches

All previous approaches give an insight into specific software artifacts. They are explicitly focused on single type of documents, avoiding external files which might in fact complement their results. However more often than not, the information provided by just one artifact is not enough. This is especially true in some maintenance tasks that demand several sources of knowledge in order to be successfully completed, as in the case of bug
correction where data contained in bug reports, source code, and even in specification documents, can support maintainers’ work.

As an illustration, the second model proposed in (Murphy, et al., 2001), called *lexical source model*, is constructed using keyword matching techniques (e.g., *grep* or *awk*) to find structural information in the source code, by specifying regular expressions related to different types of structural constructs (i.e., method declarations, method calls, variable definition, etc.). In this approach, a developer is able to specify a set of patterns for finding structural information in source code (e.g., define a regular expression to match a function call), a set of actions to be executed when specific structural information is found, and a set of rules to combine the structural information found in different files into one model. Several types of artifacts such as data files or documentation files can be scanned using this lexical approach, and the information can be combined into one model. This technique can be used to provide the structural information needed to build the aforesaid *reflection model*.

(Rastkar, 2010) proposed a framework for synthesizing the information related to an evolution task or concern, and its interactions among code, revision history, bug reports, code wikis and available documentation. Basically the approach follows two phases. In the first one, the knowledge provided by sources and related to the concern is extracted and deducted using static analysis methods, data mining and language processing techniques for populating an ontology, which is used in the second phase to complete fixed templates or predefined rules in order to form a text-based summary.

Likewise, (Witte, et al., 2008) apply code analysis and text mining on specification documents, UML diagrams and user’s manuals, with the aim of building an ontology that allows developers to cross and match the semantic knowledge between those elements. The source code and documents are processed in such way that entities are identified and extracted for being associated to a class belonging to the ontology. This approach was evaluated through intrinsic co-selection measures which turned out in acceptable levels of precision and recall for text mining techniques. Nevertheless, the results of code analysis suggested, once again, that naming conventions are fundamental for the quality of the ontology, and also, that disambiguation between parts of code is a desired feature.
Another approach which uses several types of software artifacts is (Putrycz and Kark, 2008). They apply some techniques from information extraction and natural text processing fields to source code and documentation in order to connect the entities found in both of them, in such a way that business rules underlying to design and implementation are rebuilt. Thus, after preprocessing source code, an Abstract Syntax Tree is built and stored in a knowledge database. Then, the collected data is simplified and linked to knowledge documentation. For evaluating the results, the associated key phrases are compared to others sets of documents used by analyzers, through intrinsic measures.
4. Human vs. Automatic Summarization: A Exploratory Case Study

4.1 Introduction

This chapter reports on a preliminary case study that was our first step towards the utilization of text summarization techniques for generating summaries of source code artifacts. We designed a small experiment where we investigated for the first time the use of a technique for generating extractive summaries, asked developers to manually produce indicative summaries, and also, compare them using content-based measures. It was a small in-house experiment with the aim of getting the insight needed in order to design larger studies. First of all, we wanted to have a first estimate of some of the parameters of an experiment to obtain summaries from a large group of developers. For instance, the number and characteristics of the artifacts that would be included in a summarization study, the average time spent by developers when summarizing a method, and the prominent features of the summaries that would be requested. In addition, we wanted to investigate to what extent basic extractive approaches can be used for generating summaries of methods. Regarding this, we asked developers to choose the most relevant terms from the method declaration, and then, we analyzed what kinds of terms were chosen, where they come from, the degree of agreement among the participants, and the relationships between the chosen terms and free-form descriptions of the artifacts. Finally, we wanted to get feedback from the participants about summarization of source code artifacts.

4.2 Generation of automatic summaries

Within the field of automatic text summarization, several approaches based on algebraic reduction methods, including Latent Semantic Indexing (LSI), have been proposed and the results have been promising (Steinberger and Ježek, 2009). Their major strength is
that they work only with the context of terms and thus they are language independent. LSI is an algebraic-statistical technique for extracting and representing the contextual usage of words’ meanings in texts (Deerwester, 1988). The basic idea is that the similarity of meanings between words and sets of words can be calculated based on the mutual constraints provided by the aggregate of all the word contexts in which a given word does or does not appear. Apart from its use in text summarization, LSI has been used in a variety of applications including information retrieval, document categorization, and information filtering.

Thus, for generating automated summaries we decided to use LSI to implement a (mainly) extractive summarizer based on the idea that the list of most informative terms within a method can convey a good indication about its purpose (Cruz and Urrea, 2005; Liu, et al., 2009). The first step in determining this kind of automatic summaries was to use LSI for indexing the corpus obtained by considering each method in the source code as a separate document. Then, the cosine distance between the method declaration and each of the terms in the corpus was computed in the LSI-reduced space. The corpus terms were then ordered in decreasing order based on their similarity with the method, so that the term with the highest cosine similarity to the method would be the first in the list. After this, we constructed the summary by considering only the top five terms in the ordered list. Therefore, this summarizer only uses lexical information, since no structural information is attached to the words selected.

4.3 Study definition and design

The main goal of this exploratory study is to evaluate an extractive approach based on LSI for summarizing Java methods. The quality focus is the similarity between term-based summaries generated by the LSI-based summarizer and human-generated summaries. The perspective is of researchers who want to assess to what extent information retrieval techniques can be used for generating extractive summaries. The context consists of 12 methods of the aTunes system (version 1.6), a full-featured open source audio player and manager, developed in Java.
4.3.1 Research questions

This exploratory study aims at addressing mainly three research questions: (i) how similar are the summaries generated by our LSI-based approach and the summaries created by developers? We want to assess the feasibility of using an extractive approach, successfully used in automatic text summarization, for summarizing source code; (ii) what structural information is important for generating automatic extractive summaries? We think an extractive approach for summarizing source code can be benefited of structural information of the artifact; (iii) what is the degree of agreement among developers when choosing terms? Agreement is an important issue for evaluating text summarizers, and therefore, we wanted to assess it within the context of source code summarization.

4.3.2 Context of the study

The software system we used in our case study is aTunes which consists of 218 classes and 1852 methods. We selected 12 methods, each belonging to different classes in the system. In order to assure variety, we chose the methods to have different properties, i.e., different length (lines of code), containing comments or not, having formal parameters or not, different stereotypes (accessor, mutator, creational, etc.), etc. Table 4-1 shows some relevant features of these methods.

<table>
<thead>
<tr>
<th>Position</th>
<th>Method Name</th>
<th>LOC</th>
<th>Terms</th>
<th># of Parameters</th>
<th>Does it have comments?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>showCdInfo</td>
<td>13</td>
<td>22</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>getContentPanel</td>
<td>34</td>
<td>47</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>getSearches</td>
<td>9</td>
<td>15</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>getGenre</td>
<td>16</td>
<td>21</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>compareTwoTranslations</td>
<td>17</td>
<td>25</td>
<td>2</td>
<td>No</td>
</tr>
<tr>
<td>6</td>
<td>Play</td>
<td>54</td>
<td>46</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>setTitle</td>
<td>8</td>
<td>11</td>
<td>2</td>
<td>No</td>
</tr>
<tr>
<td>8</td>
<td>navigateDir</td>
<td>55</td>
<td>59</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>playListTableModel</td>
<td>5</td>
<td>12</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>10</td>
<td>Run</td>
<td>66</td>
<td>56</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>11</td>
<td>editFiles</td>
<td>8</td>
<td>13</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>12</td>
<td>getFilesAndTrackNumbers</td>
<td>21</td>
<td>28</td>
<td>1</td>
<td>Yes</td>
</tr>
</tbody>
</table>
To obtain the list of terms of each artifact we preprocessed the source code of the methods by:

- Filtering out Java keywords and common English words
- Splitting all compound identifiers following CamelCase and underscore conventions, and discarding the original identifiers
- Keeping numbers
- Keeping just one representative word for each class of words with a common root (i.e., stemming and then reconstructing a word for each stem – we did not want to present developer with stems, which would be difficult to understand)

In regard to comments, only the method `getGenre` has leading comments that explains what the method does. All the other methods do not have comments at all, or has only very few inline comments that do not explain the purpose of the method.

Six graduate students in computer science participated in the study: three from Universidad Nacional de Colombia and 3 from Wayne State University. All of them reported more than three years of experience in Java programming. All were quite familiar with the application domain of aTunes.

### 4.3.3 Experimental design and procedure

For each of the 12 methods, participants were asked to (i) analyze the source code of the method carefully and determine its functionality to the best of their abilities. In case the source code of the method does not provide enough information for them to get a full grasp of its functionality, we encouraged them to study the Java file where the method is defined to get more information about its context, (ii) write a sentence that describes the functionality of the method as they understand it, and (iii) choose 5 terms which they think represent the functionality of the method best. The list of terms to choose from was provided to them. This list contains all of the terms found in the body of the method.

In the case of natural language texts, researchers agree that the length of the summary should be kept less than half of the source's size. With respect to a lower bound for this value, there is no general agreement, although some experiments report compression
ratios of around 10%. Thus, choosing 5 terms means that the compression ratio varies from around 8.5%, for the longest method `navigateDir`, to 45.5%, for the shortest method `setTitles`.

As our participants were located at different geographic sites, we set up an online survey to provide them with the source code of each method and their classes, as well as to collect their summaries. The participants carried out the survey independently, i.e., at the time and space of their choice, without our supervision. Developers spent on average 63 minutes for reading, understanding, and providing the summaries for all the methods.

4.3.4 Results and discussion

We collected six term-based and six sentence-based summaries for each method. After reviewing the descriptions written in English, we decided not to discard any summary since we considered that all of them were acceptable, although some descriptions contained low level implementation details. To illustrate the kind of data we collected, the Table 4-2 shows the summaries for the method `navigateDir(File dir)`.

<table>
<thead>
<tr>
<th>Developer</th>
<th>Sentence-based summaries</th>
<th>Term-based summaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>gets some files from a dir, directory, and a given repository, loading them into the application</td>
<td>dir, file, get, navigate, repository</td>
</tr>
<tr>
<td>D2</td>
<td>Loads a list of audio files found in a directory on the player repository</td>
<td>audio, directory, file, load, repository</td>
</tr>
<tr>
<td>D3</td>
<td>separates the repository files into audio, picture and dirs and calculates the time to load the audio files</td>
<td>audio, directory, duration, load, repository</td>
</tr>
<tr>
<td>D4</td>
<td>The method navigates in a directory to find new audio files and add them to the repository</td>
<td>directory, file, navigate, populate, repository</td>
</tr>
<tr>
<td>D5</td>
<td>read the content of a folder and put the information into some arrays</td>
<td>folder, load, pictures, put, read</td>
</tr>
<tr>
<td>D6</td>
<td>Given a File, generates a HashMap for audio files in the same folder of the given file</td>
<td>audio, hash, map, navigate, path</td>
</tr>
</tbody>
</table>

Evaluation of automatic summaries

We used the Pyramid score (Nenkova, et al., 2007) to intrinsically evaluate the automatic summaries. Pyramid is based on the semantic analysis of multiple human models, and its scores lead to stable evaluation results that are highly correlated with direct overall judgments of the summary quality.
We used the six developer summaries as the reference to evaluate the automatic ones. Let \( n \) be the fixed number of terms included in a summary by a developer (\( n = 5 \) in our case study) and \( t_k, 1 \leq k \leq m \), the unique terms included by all the six developers in their summaries. Each term \( t_k \) is associated with a score \( s_k \), which represents the number of developers that included \( t_k \) in their summary. When an automatic summary of \( n \) terms is generated, the pyramid score for this summary is computed as the ratio between the sum of the scores received by the \( n \) terms in the automatic summary and the maximum sum of scores that could have been achieved by any automatic \( n \)-term summary.

**Table 4-3: pyramid score computation for the automatic summary of the method showCdInfo**

<table>
<thead>
<tr>
<th>Developer</th>
<th>Selected terms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>album</td>
</tr>
<tr>
<td>D1</td>
<td>X</td>
</tr>
<tr>
<td>D2</td>
<td>X</td>
</tr>
<tr>
<td>D3</td>
<td>X</td>
</tr>
<tr>
<td>D4</td>
<td>X</td>
</tr>
<tr>
<td>D5</td>
<td>X</td>
</tr>
<tr>
<td>D6</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 4-3 shows an example of such a pyramid score computation, for the method \( \text{showCdInfo} \), in class \( \text{RipCdDialog} \) of aTunes. This method updates the information displayed in a dialog window in order to show the artist name, CD name and track names of the CD to be ripped. In the example, the six developers are identified by D1-D6, \( n = 5 \) (each summary contains five terms) and \(|T| = 13\) (the total number of unique terms in all summaries is 13). Rows D1-D6 show the term-based summaries of the developers. The last row shows the terms in the automatic summary and contains the scores of those terms in parenthesis. The terms \( cd \), \( data \), \( info \), \( show \), and \( update \) represent the terms with the highest scores, i.e., the terms which were chosen by the most developers. The pyramid score is computed as the ratio between the sum of the scores obtained by terms \( cd \), \( name \), \( set \), \( text \), and \( update \) (which were chosen in the automatic summary) and the scores obtained by terms \( cd \), \( data \), \( info \), \( show \), and \( update \) (the terms with the highest scores).
The pyramid scores obtained in this manner for the automatic summaries are in the last row of the Table 4-4. The scores for the twelve methods ranged between 0.2 and 0.56 with an average of 0.35. In the same table are the pyramid scores for all the human generated summaries. In these cases, the summaries of each developer were scored using the summaries of the other participants as reference. The average score for human generated summaries was 0.7.

Although the scores of the automatic summaries cannot be considered high, and the difference with the ones obtained by human summaries is large, they are encouraging as they are similar to the current results of lsi-based approaches in natural language summarization (Steinberger and Ježek, 2009).

### Table 4-4: The pyramid scores for human and automatic summaries

<table>
<thead>
<tr>
<th>Method</th>
<th>LOC</th>
<th>m1</th>
<th>m2</th>
<th>m3</th>
<th>m4</th>
<th>m5</th>
<th>m6</th>
<th>m7</th>
<th>m8</th>
<th>m9</th>
<th>m10</th>
<th>m11</th>
<th>m12</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>13</td>
<td>34</td>
<td>9</td>
<td>16</td>
<td>17</td>
<td>54</td>
<td>8</td>
<td>55</td>
<td>5</td>
<td>66</td>
<td>8</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td>0.82</td>
<td>0.62</td>
<td>1.00</td>
<td>0.81</td>
<td>0.69</td>
<td>0.50</td>
<td>0.95</td>
<td>0.50</td>
<td>0.79</td>
<td>0.57</td>
<td>0.38</td>
<td>0.93</td>
<td></td>
</tr>
<tr>
<td>D2</td>
<td>0.71</td>
<td>0.57</td>
<td>0.90</td>
<td>0.75</td>
<td>0.87</td>
<td>0.73</td>
<td>0.95</td>
<td>0.92</td>
<td>0.68</td>
<td>0.77</td>
<td>0.89</td>
<td>0.81</td>
<td></td>
</tr>
<tr>
<td>D3</td>
<td>0.72</td>
<td>0.69</td>
<td>0.81</td>
<td>0.87</td>
<td>0.35</td>
<td>0.69</td>
<td>0.75</td>
<td>0.69</td>
<td>0.83</td>
<td>0.57</td>
<td>0.94</td>
<td>0.11</td>
<td></td>
</tr>
<tr>
<td>D4</td>
<td>0.82</td>
<td>0.50</td>
<td>0.80</td>
<td>0.56</td>
<td>0.80</td>
<td>0.69</td>
<td>0.70</td>
<td>0.69</td>
<td>0.89</td>
<td>0.50</td>
<td>0.94</td>
<td>0.81</td>
<td></td>
</tr>
<tr>
<td>D5</td>
<td>0.47</td>
<td>0.77</td>
<td>0.52</td>
<td>0.59</td>
<td>0.80</td>
<td>0.81</td>
<td>0.75</td>
<td>0.13</td>
<td>0.89</td>
<td>0.69</td>
<td>0.94</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>D6</td>
<td>0.71</td>
<td>0.57</td>
<td>0.85</td>
<td>0.59</td>
<td>0.59</td>
<td>0.35</td>
<td>0.80</td>
<td>0.27</td>
<td>0.63</td>
<td>0.57</td>
<td>0.63</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>Automatic summarizer</td>
<td>0.52</td>
<td>0.20</td>
<td>0.29</td>
<td>0.32</td>
<td>0.42</td>
<td>0.33</td>
<td>0.22</td>
<td>0.25</td>
<td>0.55</td>
<td>0.56</td>
<td>0.23</td>
<td>0.20</td>
<td></td>
</tr>
</tbody>
</table>

All in all, the results hint that automatic summarization using TR techniques can be applied effectively also on source code. However, there is space for improvements and better results than in natural language summarization might be achieved by using the additional structural information that is present in source code.

**Origin of relevant terms**

With this goal in mind, we analyzed the developer summaries to determine the source of the terms they chose. We found that, across the twelve methods, 98.7% of the terms that appeared in the name of the method being summarized were chosen by at least one developer as part of the five terms in the summary. In addition, 88.9% of the terms present in the method’s class name and 84.6% of the terms appearing in the formal parameter types were also included by developers in the summary.
This is an indication that the terms belonging to these structural constructs should be included in an automatic summary as well. In contrast, only 20% of the terms in method names appeared in the automated summaries, 12.9% of the terms in the method class names and 30.7% of the terms in the formal parameter types. Clearly, if we are to include the terms from method names, class names, and formal parameters, the pyramid scores for the automated summaries would increase significantly.

The terms present in comments were the ones developers selected the least in their summaries. This result is explained by the lack of comments in the source code of the selected methods. As was mentioned before, only the method `getGenre` has leading comments explaining its purpose. The other five methods have very short inline comments that do not help too much to understand the functionality. For the method `getGenre`, 15 of the terms in the leading comments, out of 18, were chosen by at least one developer. This fact also highlights the relevance of good leading comments for software understanding.

**Intersection of sentence-based summaries and term-based summaries**

Each developer wrote a description of the purpose of the method and chose the five most relevant terms that describe the functionality of the method. We analyzed how many relevant terms appear in the description. Table 4-5 shows the size of the intersection between the set of five terms chosen and the words of the description. To calculate these intersections we considered words with a common root as equals (e.g., shown, show, and shows are all equals). On average, Developer 2 used 3.8 terms out of 5 in his description, while Developer 5 used only 1.7. In average, a developer uses around 57% of his selected terms in his descriptions. This percentage hints again that automatic extractive summarization using TR techniques can be applied successfully also on source code artifacts and the selection of relevant terms might be a promising starting point for generating sentence-based summaries.
Table 4-5: Size of the intersection between terms chosen and words in the descriptions

<table>
<thead>
<tr>
<th>Method</th>
<th>LOC</th>
<th>m1</th>
<th>m2</th>
<th>m3</th>
<th>m4</th>
<th>m5</th>
<th>m6</th>
<th>m7</th>
<th>m8</th>
<th>m9</th>
<th>m10</th>
<th>m11</th>
<th>m12</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>13</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>D2</td>
<td>16</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>D3</td>
<td>17</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>D4</td>
<td>17</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>D5</td>
<td>16</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>D6</td>
<td>16</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Agreement among developers

Another aspect that we analyzed is the agreement among developers. In order to understand how much developers agree when choosing the terms which describe a source code artifact, we assessed the agreement level among participants using the Jaccard coefficient (Rogers and Tanimoto, 1960). For each couple of developers \((A,B)\), the Jaccard similarity coefficient is computed as:

\[
J(A_x, B_x) = \frac{|A_x \cap B_x|}{|A_x \cup B_x|}
\]

where \(A_x\) and \(B_x\) represent the sets of relevant terms marked by developers \(A\) and \(B\) (respectively) for artifact \(x\). In other words, the agreement is equal to the number of relevant terms shared between the two participants, over the number of different relevant terms marked by them. Along these lines, a Jaccard similarity coefficient of one represents highest agreement, whereas zero represents highest disagreement.

For our future studies, it will be important to examine if developers agree or disagree regarding the terms that better describe the purpose of a method. If developers clearly agree, we can draw conclusions based on studies with few developers and use their decisions to evaluate tools for extracting relevant terms. Otherwise, we have to design procedures to combine answers of many participants and extract the consensus.

Table 4-6 summarizes the Jaccard index between the set of terms chosen by each pair of developers, for the method `showCdInfo(_)`.

For this method, the average Jaccard index over all pairs of developers is 0.33.
Table 4-6: Jaccard index between pairs of developers for the method showCdInfo

<table>
<thead>
<tr>
<th>Developer</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
<th>D6</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td></td>
<td>0.43</td>
<td>0.43</td>
<td>0.43</td>
<td>0.25</td>
<td>0.43</td>
</tr>
<tr>
<td>D2</td>
<td>0.43</td>
<td></td>
<td>0.25</td>
<td>0.43</td>
<td>0.11</td>
<td>0.43</td>
</tr>
<tr>
<td>D3</td>
<td>0.43</td>
<td>0.25</td>
<td></td>
<td>0.43</td>
<td>0.43</td>
<td>0.25</td>
</tr>
<tr>
<td>D4</td>
<td>0.43</td>
<td>0.43</td>
<td>0.43</td>
<td></td>
<td>0.25</td>
<td>0.43</td>
</tr>
<tr>
<td>D5</td>
<td>0.25</td>
<td>0.11</td>
<td>0.43</td>
<td>0.25</td>
<td></td>
<td>0.11</td>
</tr>
<tr>
<td>D6</td>
<td>0.43</td>
<td>0.43</td>
<td>0.25</td>
<td>0.43</td>
<td>0.11</td>
<td></td>
</tr>
</tbody>
</table>

The general results show that the average Jaccard measure between pairs of participants is 0.34. The smallest agreement value is 0.19 for method m8, while the highest one is 0.51 for method m3. An agreement of 34% should be considered as low, if you compare it with the agreement among human subjects reported by text summarization researchers. These agreement reports range from 25% to 96% (Jing, et al., 1998), and are affected by several variables such as the type and length of the articles summarized, the length of the summaries generated, and also, the measures used to assess the agreement.

4.3.5 Threats to validity

The results are difficult to generalize as we only considered one system and twelve methods. While we tried to vary their properties, the chosen methods may not necessarily be the most representative of the system and even less so of other systems. Furthermore, aTunes has self-explanatory identifiers and very few comments, so the results might not be valid for systems with poor identifier naming or good comments.

Although participants were experienced Java developers, the results obtained with the real system developers might be different. Additionally, although the collected descriptions are acceptable, it is important to consider that they spent only five minutes, in average, understanding and summarizing each method. So, the results with professional developers that spend more time to familiarize themselves with the architecture and major functionality the system probably will be better and more reliable than the ones we obtained.

Therefore, the results should be taken only as guides for further user studies.
4.4 Conclusions

The results of this preliminary study suggest that we can use text retrieval methods for automatically summarizing source code. Additionally, structural information could improve the results of extractive approaches; however, more analysis is needed to investigate in more detail how structural information can be best integrated with the lexical information in order to generate better summaries. Apart from that, new user studies can be conducted to explain the rationale behind developers’ choices, when producing free-form summaries of source code. The goal of our automated summarization tools is to approximate developers’ reasoning.

In the future of this research area, agreement among human summarizers will be an issue that must be considered, especially when designing an evaluation procedure, or when comparing the results of different summarizers. The relatively low agreement obtained in this experiment points out that evaluation experiments should include multiple human subjects and procedures that mitigate the problem of subjectivity of relevance judgments.
5. Human Summarization of Source Code Artifacts

5.1 Introduction

As mentioned in chapter 1, one of the main goals of software summarization is to support the comprehension needed during the daily tasks of software developers, by providing short, but representative summaries of the source code entities such as, packages, classes, methods, etc. Ideally, such summaries will be informative enough to be used for filtering irrelevant artifacts for a task at hand, 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. In the worst scenario, developers would have to read both 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).

In order to achieve the above stated goal, we must ensure that summaries include the most relevant information about the code they summarize and that they are easy to use by developers. Usability requirements imply basically that we must be able to generate source code summaries automatically. Chapter 6 discusses some alternatives to achieve this goal. On the other hand, determining what is relevant in source code artifacts is one key issue that we need to address and the answer may be different for various types of source code entities (e.g., class vs. method), and also may differ between programming languages. Additionally, source code is a mixed artifact that is made of text and has a specific underlying structure. The summaries have to reflect both the text and the structure. Usually, the text encodes domain semantics (that is, what is the software about) and design rationale (that is, why is the code built this way), whereas the structure encodes the algorithm (that is, how does the software work). Developers often need to understand all these aspects of the code.
Some feasible ways to identify parts of source code artifacts that convey important information, and so they should be included in summaries, are: (1) to study how developers create summaries of various types of source code artifacts, (2) to analyze the summaries generated by them, and (3) to use their expertise as developers for determining what information should be included in the summaries of source code artifacts.

Thus, this chapter presents 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, it describes how these human-generated summaries can be used to carry out an intrinsic evaluation of automatic summarization approaches.

It is important to note that, as was presented in Chapter 3, over the last years, software summarization, and source code summarization in particular, have been the subject of an increasing number of research efforts. All this research, however, focused on automating the summarization process and used heuristics determined by the authors to establish what should be included in the summaries. There is currently no study investigating how developers actually summarize code, to serve as a starting point for automatically building summaries. That is why we address this problem by performing the empirical study described in this chapter.

The results of this study have several important implications. First, they can be used to understand what a code summary should contain, thus serving as guidelines for developers, but also as requirements to conceive better automatic summary generators. At the same time, the study highlights what kinds of textual code elements developers consider most important when summarizing source code. This can be useful in defining code naming conventions that enforce the quality of certain identifiers, and also to build program comprehension tools and recommenders.
5.2 Study definition and design

The goal of this study is to investigate how developers summarize source code, with the purpose of investigating how such a code abstraction task is performed, and what textual elements are being used. The quality focus is source code comprehensibility, while the perspective is of researchers interested in understanding the abstraction activity performed by developers when comprehending source code and interested to identify what a source code summary should contain. The context consists of artifacts extracted from three Java software systems: aTunes\(^4\), Apex Text\(^5\), and Claros\(^6\), and a group of undergraduate and graduate students from Universidad Nacional de Colombia, Bogotá (referred to as Bogotá) and from the University of Sannio, Italy (referred to as Sannio).

5.2.1 Description of the context of the study

Objects

The three software systems used in the study are all open-source and from problem domains familiar to the students. Claros is an on-line Web mail management application and version 1.0, used in our study, is composed of 44 Java classes (6 KLOC) and 34 JSP pages (2 KLOC). Apex Text is a general purpose text editor, which supports syntax highlighting for several programming languages. The version used in our study, i.e., 1.0, is composed of 438 classes and 75 KLOC. The third system, i.e., ATunes 1.6.0, is a media player and manager, which has 221 Java classes and 25 KLOC. While we report their size, one must note that it is not relevant to the task the subjects performed. Our goal is to express the fact that these are rather usual systems.

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, for each system we selected one package, two classes, two methods, and two sequences of methods (each one composed of three methods, one called by the other) to be summarized. The two method sequences were chosen such that one contained methods which were from the same class and the

\(^4\) http://www.atunes.org
\(^5\) http://www.openapex.org/text/
\(^6\) http://www.claros.org/
second contained methods from two different classes. Table 5-1, Table 5-2, Table 5-3, and Table 5-4 show relevant features of the selected artifacts.

Table 5-1: Characteristics of the methods chosen for the study

<table>
<thead>
<tr>
<th>System</th>
<th>Name</th>
<th>LOC</th>
<th>Terms</th>
<th>Unique Terms</th>
<th># of parameters</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>aTunes</td>
<td>ripTracks</td>
<td>48</td>
<td>236</td>
<td>74</td>
<td>3</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>persistRepositoryForFuture</td>
<td>47</td>
<td>222</td>
<td>85</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>Apex Text</td>
<td>print</td>
<td>30</td>
<td>223</td>
<td>75</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>loadClassData</td>
<td>23</td>
<td>134</td>
<td>58</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>Claros</td>
<td>fetchParts</td>
<td>48</td>
<td>268</td>
<td>65</td>
<td>2</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>getNextHeader</td>
<td>68</td>
<td>182</td>
<td>48</td>
<td>2</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 5-2: Characteristics of the classes chosen for the study

<table>
<thead>
<tr>
<th>System</th>
<th>Name</th>
<th>LOC</th>
<th>Terms</th>
<th>Unique Terms</th>
<th># of parameters</th>
<th># of methods</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>aTunes</td>
<td>LanguageTool</td>
<td>135</td>
<td>650</td>
<td>111</td>
<td>6</td>
<td>10</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>AudioScrobblerAlbumsRunnable</td>
<td>111</td>
<td>388</td>
<td>92</td>
<td>4</td>
<td>5</td>
<td>yes</td>
</tr>
<tr>
<td>Apex Text</td>
<td>Logger</td>
<td>322</td>
<td>1464</td>
<td>173</td>
<td>9</td>
<td>12</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>FileAndTextTransferHandler</td>
<td>153</td>
<td>763</td>
<td>178</td>
<td>4</td>
<td>6</td>
<td>yes</td>
</tr>
<tr>
<td>Claros</td>
<td>CLPop3Controller</td>
<td>246</td>
<td>703</td>
<td>1130</td>
<td>8</td>
<td>21</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>MessageController</td>
<td>170</td>
<td>735</td>
<td>144</td>
<td>0</td>
<td>17</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 5-3: Characteristics of the method sequences chosen for the study

<table>
<thead>
<tr>
<th>System</th>
<th>Name of methods</th>
<th>LOC</th>
<th>Terms</th>
<th>Unique Terms</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>aTunes</td>
<td>run; loadRepository; navigateDir</td>
<td>90</td>
<td>465</td>
<td>103</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>run; getDirectory; getArtist</td>
<td>51</td>
<td>308</td>
<td>80</td>
<td>no</td>
</tr>
<tr>
<td>Apex Text</td>
<td>build; getHelpTopics; addTopicsRecursively</td>
<td>49</td>
<td>272</td>
<td>85</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>getExternalConfigFileInputStream; getExternalConfigFile; getFileByCreatingMissingDirs</td>
<td>52</td>
<td>334</td>
<td>84</td>
<td>yes</td>
</tr>
<tr>
<td>Claros</td>
<td>constructMessage; parseStringToAddressArray; extendedTrim</td>
<td>76</td>
<td>409</td>
<td>88</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>MessageController.getBodyText; prepareInlineHTMLContent; getPartIdByContentId</td>
<td>58</td>
<td>329</td>
<td>74</td>
<td>yes</td>
</tr>
</tbody>
</table>
Table 5-4: Characteristics of the packages chosen for the study

<table>
<thead>
<tr>
<th>System</th>
<th>Name</th>
<th>LOC</th>
<th>Terms</th>
<th>Unique Terms</th>
<th># of classes</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>aTunes</td>
<td>Amazon</td>
<td>156</td>
<td>675</td>
<td>112</td>
<td>3</td>
<td>yes</td>
</tr>
<tr>
<td>Apex Text</td>
<td>Search</td>
<td>1252</td>
<td>5171</td>
<td>341</td>
<td>10</td>
<td>yes</td>
</tr>
<tr>
<td>Claros</td>
<td>base</td>
<td>455</td>
<td>1049</td>
<td>100</td>
<td>4</td>
<td>no</td>
</tr>
</tbody>
</table>

A particular characteristic of the source code artifacts is that some terms are repeated constantly in different identifiers. For example, the first method in table 5-1 (ripTracks) has 236 terms, but only 74 of those are unique terms. In fact, the ratio between unique terms and the total number of terms of an artifact is very low (in average, 0.33 for methods, 0.25 for sequences, 0.19 for classes, and 0.11 for packages). Furthermore, this ratio decreases considerably as the artifact length increases its size.

As a result of the repetition of terms, a short method, like ripTracks, with only 48 LOC, can have a comparable number of unique terms as a long package, such as base, with 455 LOC (it is almost 10 times longer than ripTracks). In this specific example, ripTracks has 74 unique terms, whilst base has only 100.

Subjects
The subjects from Bogotá are three M.Sc. and 15 senior undergraduate students. Two of the master students also work as professional Java developers. The subjects from Sannio are 15 first year M.Sc. students. All students have a good knowledge of Java, software engineering, and some experiences in dealing with existing large software systems. They have a good capability to read and write in English. The subjects from Bogotá summarized artifacts from aTunes and Apex Text, whereas the subjects from Sannio summarized artifacts from aTunes and Claros.

5.2.2 Research questions
The study aims at addressing the following research questions:

RQ1: How long are the source code summaries generated by developers? This research question aims at providing coarse-grained information about the summaries, with the aim
of understanding their conciseness or verbosity. Appropriate summary length might depend on the type of artifact to be summarized (i.e., methods, classes, packages, etc.) and the specific purpose for which the summary is generated. In automatic text summarization, the length of the summary is a parameter that considerably influences the results of both types of evaluations, intrinsic and extrinsic.

**RQ2:** What kind of terms do developers use in the source code summaries? This research question aims at analyzing the content of summaries, to investigate (i) whether terms contained in the source code are used or if, instead, developers tend to use other terms; (ii) what kind of code identifiers are often used in summaries (i.e., class names, method names, attributes, local variables, etc.); and (iii) what parts of speech (nouns, verbs, adjectives, others) are being used when summarizing packages, classes, methods, or method chains. Given that automatic summaries are meant to be read by developers primarily, the information related to this question is important to ensure that they satisfy their information needs.

**RQ3:** What parts of source code artifacts do developers perceive as relevant for describing artifacts? This research question complements RQ2 and aims at understanding what developers perceive important in source code artifacts (e.g., class or method names, local variables, etc.) when understanding and summarizing them. We are also interested in investigating if what developers perceive as important is different than what they actually highlight in the source code when performing term-based summarization.

### 5.2.3 Experiment layout and procedure

The study was conducted in four sessions (i) a pre-experiment test, (ii) a comprehension laboratory, and (iii) two experimental sessions, each replicated in Sannio and Bogotá.

**Pre-study test and subjects grouping**

Before the study, we performed a self-assessment pre-test with the aim of collecting information about: (i) the grades obtained in previous software engineering and programming courses; (ii) the knowledge of Java and the experience in software development and in evolving existing systems; and (iii) the knowledge of English
language (reading and writing), as participants were not English native speakers, yet the software was written with English comments and identifiers.

After collecting the data of the pre-test, we split subjects at each location into two groups, A and B. In Bogotá, the subjects in Group A performed summarization tasks on ATunes, while subjects in Group B on Apex Text. In Sannio, subjects in Group A had to perform summarization tasks on ATunes, while subjects in Group B on Claros. We made sure each group was composed of both experienced and inexperienced subjects, based on the information collected in the questionnaire.

The motivation for using these specific systems in the study is twofold. First, we wanted to have a more diverse set of systems, including a Web application, so that summaries do not depend on specific aspects of a system. Second, we preferred to keep one system (ATunes) fixed between the two experiments to check whether subjects of the two populations behave differently.

Comprehension session
Once we assigned subjects to systems, we performed a training lab with the objective of allowing the subjects to gain a first, coarse understanding of the systems to be used in the experiment. We asked the subjects to read documentation of the system, use it and browse the source code, to understand the main features and their implementation.

After the comprehension session, the study was organized in two laboratory sessions of three hours each, one performed several days after the other. Before starting the first laboratory, we explained to the subjects the detailed tasks to be performed. We also provided them with a checklist containing detailed instructions, and with the experiment material.

Summarization session I: sentence-based summaries in English
In the first session, the subjects received the source code and a printed form listing the names and path of seven artifacts (one package, two classes, two methods, and two method call sequences). For each artifact, subjects had to read and understand the source code and then write one or more sentences in English summarizing the artifact, in
the space available on the sheet. We provided half-a-page for each artifact summary, which turned out to be more than enough in each case.

Since summaries developed for different purposes could have completely different form, we explicitly instructed subjects to write summaries with the purpose of explaining source code artifacts to a project newcomer, who understands the domain of the software system.

**Summarization session II: term-based summaries and summaries in native language**

In the second session, subjects received again the source code and for each artifact they had to perform two tasks.

First, the subjects had to build a term-based summary by identifying in the source code the terms that they considered relevant for describing the artifact. A term could be any source code identifier or part of an identifier, which could be composed of multiple words (e.g., `createFileName`), or any word contained in comments. We asked subjects to mark the terms in the source code by means of XML tags. For instance, terms relevant for describing the first class to be summarized had to be enclosed between `<c1>` and `</c1>` tags, while terms relevant to describe the second method to be summarized had to be enclosed between `<m2>` and `</m2>` tags. This approach allowed us to automate the analysis of the results and to know the location of each marked term in the source code, e.g., whether it was a class name, a local variable, a constant, etc.

After completing the term-based summaries, the subjects were asked to summarize again the artifact with one or more sentences, this time written in their native language (Spanish or Italian). Although we did not use such summaries for our analyses, their purpose was to check, in cases when the English summaries were clearly wrong, whether this was due to an insufficient comprehension level about the artifact, or instead, it was due to difficulties in writing the summary in English.

The reason for having the summarization task performed in two sessions situated several days apart is three-fold: (i) avoid having a tiring or boredom effect by performing the experiment in a single day; (ii) limiting the learning effect, i.e., avoiding that term-based
summaries would contain words picked from sentence-based summaries in English or vice versa; (iii) avoiding the English based summary to be a simple translation of the Spanish/Italian, and vice versa.

**Post-study survey questionnaire**

After each experimental session, the subjects were asked to fill a post-study survey questionnaire. The questionnaire was composed of two sections: the first one had to be filled in both sessions, while the second only after the second session.

The first section contained questions on whether the session objectives were clear, whether the subjects had enough time to perform the tasks, whether they were able to examine the source code without experiencing particular problems, and whether they experienced difficulties in formulating the summaries. All question answers were on a four-point Likert scale (Oppenheim, 1992) (1: strongly disagree, 2: weakly disagree, 3: weakly agree, and 4: strongly agree).

The second part of the questionnaire contained questions aimed at collecting information about the perceived usefulness of the terms contained in different source code locations, i.e., class names, attribute names, method names, etc., when summarizing packages, classes, methods, and method sequences. The list of locations included in this survey was the following:

1. Class Names
2. Attribute Names
3. Attribute Types
4. Method Names
5. Method Returned Types
6. Returned Variable Names
7. Parameter Names
8. Parameter Types
9. Local Variable Names
10. Local Variable Types
11. Invoked Method Names
12. Source Code Comments
13. Package Names
14. Exception Names

**Validation of the collected summaries**

We excluded from our analysis summaries that were clearly wrong due to lack of understanding of the artifact by the subject, as we were interested in building our analyses on summaries which reflect a good understanding of the code.
To this aim, two judges independently rated each summary using a scale ranging between 1 (low) and 3 (high), and after discussing disagreements, they filtered out summaries with an average score less than 2. Overall, 15% of the summaries were discarded and not used in the analysis.

5.3 Results and discussion
This section reports the results of the study we performed.

5.3.1 How long are the summaries generated by developers?
To address RQ1, we counted the number of words composing each summary. We considered all words in the summaries, and similarly, for the source code, we considered identifiers, types, comment words, and also programming language keywords. The reason for using the latter is because sometimes such keywords (e.g., while, if, else, null) were used in the summaries, thus should be considered part of the source code text. Finally, we split compound identifiers using the Camel Case separator and the underscore, considering terms composing each identifier as separate words.

Table 5-5 reports descriptive statistics (1st and 3rd quartile, median, mean, and standard deviation $\sigma$) of the number of terms used in sentence-based and term-based summaries. The median length of sentence-based summaries varies between 33 for packages to 47 for method sequences. However, the Kruskal-Wallis test did not indicate any significant difference (p-value=0.11), in length, among sentence-based summaries for different kinds of artifacts. Subjects did not write more detailed summaries for larger artifacts (classes and packages). Method sequences tend to have slightly longer summaries, often written by shortly explaining the behavior of each method in the sequence. For term-based summaries, the length varies between 8 words for methods and 18 for packages. In this case, the difference, in length, among different kinds of artifacts is significant (p-value<0.001). A pairwise comparison indicated that package, class, and chains are summarized using a significantly higher (p-value<0.01) number of words than single methods.

According to these results, 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 5-5. 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.

Table 5-5: Descriptive statistics about the length in the sentence-based and term-based summaries (SB: sentence-based; TB: term-based)

<table>
<thead>
<tr>
<th>Type of Artifact</th>
<th>q1</th>
<th>median</th>
<th>q3</th>
<th>mean</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>20</td>
<td>10</td>
<td>33</td>
<td>18</td>
<td>52</td>
</tr>
<tr>
<td>Class</td>
<td>29</td>
<td>9</td>
<td>36</td>
<td>17</td>
<td>46</td>
</tr>
<tr>
<td>Sequence</td>
<td>29</td>
<td>9</td>
<td>47</td>
<td>14</td>
<td>65</td>
</tr>
<tr>
<td>Method</td>
<td>25</td>
<td>6</td>
<td>36</td>
<td>8</td>
<td>49</td>
</tr>
</tbody>
</table>

5.3.2 What kind of terms do developers include in the source code summaries?

Terms in common between summaries and artifacts

Table 5-6 reports statistics of the percentage of words used in sentence-based summaries that are also contained in the source code artifact to be summarized. As shown, around 40% of the words used by the subjects to describe an artifact are contained in its declaration (median between 35% for packages and 42% for classes). Surprisingly, the percentage is not significantly higher when summarizing larger artifacts (packages and classes) than for methods and method sequences. This happens partially because the vocabulary, i.e., the unique terms, of classes and packages is not significantly larger than one single method or a short method sequence (see tables in section 5.2.1), thus the likelihood of choosing a word contained in a large artifact is not much higher than in a short one.

In connection with this result, it is pertinent to remark that (Copeck and Szpakowicz, 2004) analyzed 9000 manually-written summaries of newswire stories and discovered that no more than 55% of the vocabulary they used occurs in the original document. A smaller experiment with 300 summaries of news articles, written by expert summarizers, revealed that “expert summarizers often reuse the text in the original document to
produce a summary” (Jing and McKeown, 1999). Thus, the researchers in this case reported that 78% of the summary sentences in this experiment were produced by copy and paste.

Our result cannot be considered low, if one considers that the original document is a mixture of syntactic constructions of the programming language, and words written in English that often do not form complete sentences.

<table>
<thead>
<tr>
<th>Table 5-6: Descriptive statistics of the percentage of sentence-based summaries terms present in the source code artifact</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type of Artifact</strong></td>
</tr>
<tr>
<td>Package</td>
</tr>
<tr>
<td>Class</td>
</tr>
<tr>
<td>Sequence</td>
</tr>
<tr>
<td>Method</td>
</tr>
</tbody>
</table>

Table 5-7 illustrates an example of sentence-based and term-based summaries, written by one of the subjects for the method `ripTracks` and the package `Amazon` in aTunes. The terms in italics and underlined in the sentence-based summaries are terms that exist also in the respective artifact.

<table>
<thead>
<tr>
<th>Table 5-7: Examples of sentence-based and term-based summaries</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type of summary</strong></td>
</tr>
<tr>
<td>Sentence-based summary</td>
</tr>
<tr>
<td>Term-based summary</td>
</tr>
</tbody>
</table>
Part of speech analysis (POS)
We also analyzed the distribution of POS in the term and sentence-based summaries. Figure 5-1 and Figure 5-2 show the median percentage of terms, belonging to different POS categories, which appear in sentence-based and term-based summaries for different kinds of artifacts. For sentence-based summaries (Figure 5-1), it can be noticed how, despite a high percentage of other words (articles, prepositions, etc.), nouns play an important role in the summaries, and they are being used in a percentage between 29% for methods and 39% for packages. Verbs are used in 16% of cases for packages, 17% for method sequences, and 19% for methods and classes. Adjectives and adverbs are rarely used. The Kruskal-Wallis test reported a significant difference between the percentages of nouns (p-value = 0.004) in kinds of artifacts, with a higher percentage in packages. No significant differences were found for the other POS.

When analyzing the term-based summaries (Figure 5-2), it can be noticed that, as expected, other words are not used, except in 2% of cases for method sequence summaries. The percentage of nouns increases with respect to verbs and ranges between a median value of 64%, for sequences, and 83% for packages. It is thus clear that, when subjects had to choose terms in the source code that best describe the artifacts, they mostly deemed nouns to be the most appropriate, very likely because they characterize the properties of the artifacts for packages/classes, or the input/output data and affected attributes for methods and method sequences. It is also surprising that, for both sentence-based and term-based summaries, verbs do not appear to be used often to summarize methods and method sequences, although verbs are usually used to describe actions/behaviors.
We then compared the POS proportion of summaries with respect to the POS proportions in the artifacts to be summarized. For sentence-based summaries, 36% of the summaries have a significantly lower proportion of nouns than their respective artifacts; the proportions were similar between code and summaries for verbs, adjectives, and adverbs.
and, obviously, almost all artifacts (94%) contained a lower proportion of other POS. For term-based summaries, we found that they contained the same proportions of POS as the code artifacts, with only a lower proportion of other POS in 10% of the cases.

Finally, when comparing summaries with user manuals, we found that (i) sentence-based summaries had similar proportions of all POS as manuals in 95% of the cases, and (ii) term-based summaries also had similar proportions, but with less other POS in 45% of the cases and more nouns in 17% of the cases.

Thus, term-based summaries have a similar POS distribution to the source code (sometimes with a few more nouns), while sentence-based summaries are written just as any other natural-language document from the same domain.

We then compared the POS proportion of summaries with respect to the POS proportions in the artifacts to be summarized. For sentence-based summaries, 36% of the summaries have a significantly lower proportion of nouns than their respective artifacts; the proportions were similar between code and summaries for verbs, adjectives, and adverbs and, obviously, almost all artifacts (94%) contained a lower proportion of other POS. For term-based summaries, we found that they contained the same proportions of POS as the code artifacts, with only a lower proportion of other POS in 10% of the cases. Finally, when comparing summaries with user manuals, we found that (i) sentence-based summaries had similar proportions of all POS as manuals in 95% of the cases, and (ii) term-based summaries also had similar proportions, but with less other POS in 45% of the cases and more nouns in 17% of the cases.

**Origin of the terms**

We further analyzed the location of the terms included in the term-based summaries of the various kinds of software artifacts. We were expecting that words from certain code locations would be used more frequently than others in summaries for various artifacts. For example, common naming and programming conventions recommend that the class name should reflect the role of the class. Hence, we expected that class names will appear in all class summaries. More specifically, we were expecting that:
• **Package summaries** would (almost) always contain the name of the package being summarized. In addition, we expected that most class names in the package and some method names and attributes would be used in moderation.

• **Class summaries** would (almost) always contain the name of the class being summarized. We also expected terms from attribute names, method names, attribute types, and comments to be used most frequently.

• **Method sequence summaries** would (almost) always include the method names. Parameter names, parameter types, method calls, and local variable names would also be frequent.

• **Method summaries** would (almost) always include the name of the method being summarized and the name of the class to which the method belongs to. Parameter types, parameter names, local variable names, and comments would also be used often in method summaries.

Table 5-8 reports percentages of summaries, for different kinds of artifacts, that contain at least one element from a given code location. The results reveal some expected, but also unexpected facts.

**Table 5-8: Percentages of term-based summaries containing at least one term belonging to a given code location**

<table>
<thead>
<tr>
<th>Package Location</th>
<th>%</th>
<th>Class Location</th>
<th>%</th>
<th>Sequence Location</th>
<th>%</th>
<th>Method Location</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>class name</td>
<td>71</td>
<td>met name</td>
<td>67</td>
<td>Call name</td>
<td>81</td>
<td>var name</td>
<td>81</td>
</tr>
<tr>
<td>met name</td>
<td>71</td>
<td>call name</td>
<td>60</td>
<td>Var name</td>
<td>81</td>
<td>met name</td>
<td>79</td>
</tr>
<tr>
<td>attr name</td>
<td>62</td>
<td>attr name</td>
<td>53</td>
<td>Met name</td>
<td>63</td>
<td>par name</td>
<td>60</td>
</tr>
<tr>
<td>Par name</td>
<td>52</td>
<td>class name</td>
<td>53</td>
<td>const</td>
<td>44</td>
<td>call name</td>
<td>57</td>
</tr>
<tr>
<td>Var name</td>
<td>43</td>
<td>var name</td>
<td>42</td>
<td>Attr name</td>
<td>42</td>
<td>Const</td>
<td>33</td>
</tr>
<tr>
<td>call name</td>
<td>38</td>
<td>Const</td>
<td>37</td>
<td>Par name</td>
<td>37</td>
<td>attr name</td>
<td>21</td>
</tr>
<tr>
<td>Const</td>
<td>29</td>
<td>attr type</td>
<td>21</td>
<td>Var type</td>
<td>12</td>
<td>var type</td>
<td>19</td>
</tr>
<tr>
<td>Par type</td>
<td>29</td>
<td>par name</td>
<td>19</td>
<td>Ret type</td>
<td>9</td>
<td>Comment</td>
<td>10</td>
</tr>
<tr>
<td>Ret type</td>
<td>29</td>
<td>Comment</td>
<td>7</td>
<td>class name</td>
<td>7</td>
<td>Java keyw</td>
<td>5</td>
</tr>
<tr>
<td>package</td>
<td>24</td>
<td>ret.var name</td>
<td>7</td>
<td>Java keyw</td>
<td>7</td>
<td>ret.var name</td>
<td>5</td>
</tr>
<tr>
<td>attr type</td>
<td>19</td>
<td>import</td>
<td>5</td>
<td>comment</td>
<td>5</td>
<td>class name</td>
<td>2</td>
</tr>
<tr>
<td>Var type</td>
<td>14</td>
<td>Java keyw</td>
<td>5</td>
<td>Ret.var name</td>
<td>5</td>
<td>par type</td>
<td>2</td>
</tr>
<tr>
<td>import</td>
<td>10</td>
<td>var type</td>
<td>5</td>
<td>Par type</td>
<td>2</td>
<td>ret type</td>
<td>2</td>
</tr>
<tr>
<td>comment</td>
<td>5</td>
<td>Other</td>
<td>2</td>
<td>attr type</td>
<td>0</td>
<td>attr type</td>
<td>0</td>
</tr>
<tr>
<td>Java keyw</td>
<td>5</td>
<td>par type</td>
<td>2</td>
<td>Exc name</td>
<td>0</td>
<td>exc name</td>
<td>0</td>
</tr>
<tr>
<td>Exc name</td>
<td>0</td>
<td>exc name</td>
<td>0</td>
<td>import</td>
<td>0</td>
<td>import</td>
<td>0</td>
</tr>
<tr>
<td>Other</td>
<td>0</td>
<td>package</td>
<td>0</td>
<td>other</td>
<td>0</td>
<td>Other</td>
<td>0</td>
</tr>
<tr>
<td>Ret.var</td>
<td>0</td>
<td>ret type</td>
<td>0</td>
<td>package</td>
<td>0</td>
<td>Package</td>
<td>0</td>
</tr>
</tbody>
</table>
• **Package summaries.** As expected, a very high percentage of summaries contain class names (71%) and attribute names (62%). Unexpected was that the percentage of summaries containing method names (71%) was equal to that of summaries containing class names. This reveals that method names are just as important as class names for package summaries. More surprisingly, only 24% of the summaries contained the package name. One can notice that, despite the fact that one may expect that package summaries are written at a higher level of abstraction, 52% of them contain parameter names, 43% variable names, and 38% method invocations.

• **Class summaries.** As expected, a high number of summaries contained method names (67%) and attribute names (53%). Unexpectedly, class names were found in only 53% of the class summaries, being chosen less than method invocations (60%). Another interesting finding is the moderate importance of local variable names (43%) and constants (37%).

• **Method sequence summaries.** As expected, they contain invocations very often (81%) as well as local variables (80%). Unexpectedly, subjects included the method names in only 63% of their method sequence summaries and only in 37% of the cases they referred to the formal parameters.

• **Method summaries.** Unexpectedly, the code location found in most summaries was local variables (81%). The method names were revealed to be important, even though less than expected, being included in only 79% of the summaries. As we expected, parameter names were included often (60%). However, comments and parameter types were included in only 10% and 2% of the summaries, respectively.

When analyzing the overall numbers, we made some interesting observations:

- Comment terms are rarely used across all artifact types, probably because the artifacts were sparsely commented.

- Exceptions are never used, as subjects – even for method and sequences – mainly focused on the regular behavior, without providing details about the exceptional behavior.
Text extracted from literals and named constants are used in about 35% of the summaries, suggesting that, when understanding and summarizing source code, constants are among the elements subjects looked at. This is because constants often encode valuable domain information.

5.3.3 Which source code parts do developers consider useful?

After the term-summarization task, the subjects were asked to rate the usefulness in creating summaries of 14 source code parts. The participants rated the usefulness of the 14 different parts of code with a scale from 1 to 4, where 1 indicates totally useless, 2 moderately useless, 3 moderately useful, and 4 very useful, for each kind of artifact summary. We did not use the 5-level scale in order to avoid the tendency to mark non-committal answers (i.e., neither useful nor useless). We collected 25 sets of answers, as some subjects did not attend this session of the experiment. In the results discussed below, the responses were grouped into only two categories: useless and useful, ignoring the adverbs that qualify their opinions (totally, moderately, and very). Additionally, the relevance of a part of source code was calculated as the percentage of participants that considered this particular part as useful for summarizing a type of artifact.

Which source code parts are useful for summarizing a method or a class?

Not surprisingly, through the questionnaire responses we found that a lot of parts are considered useful when summarizing methods and classes. Table 5-9 shows that their respective names were considered useful for almost all of the participants. Thus, the name of a method is unquestionably important to sums up its purpose, and the same applies in the case of a class. Also, almost all of the participants think that the names of the methods of a class are useful for describing its purpose.

Apart from that, most of the participants considered invoked method names and attribute names as relevant information sources for summarizing both types of artifacts. For a method specifically, the results showed that the name of the class where it is, the name and type of its parameters, and the names of its local variables should be considered for extracting relevant information. Another interesting aspect, in the case of methods, is that entity names were highlighted by more participants than their types. One might argue that this is because quite often the types of these entities are basic types of the programming
language, such as integer, float or string, and thus, they do not convey domain information.

Table 5-9: Ranking of parts of source code according to the importance for summarizing: (a) a method, and (b) a class

<table>
<thead>
<tr>
<th>Location</th>
<th>Percentage of usefulness</th>
<th>Location</th>
<th>Percentage of usefulness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method Names</td>
<td>96,0%</td>
<td>Class Names</td>
<td>100,0%</td>
</tr>
<tr>
<td>Invoked Method Names</td>
<td>96,0%</td>
<td>Method Names</td>
<td>96,0%</td>
</tr>
<tr>
<td>Attribute Names</td>
<td>92,0%</td>
<td>Invoked Method Names</td>
<td>92,0%</td>
</tr>
<tr>
<td>Class Names</td>
<td>84,0%</td>
<td>Attribute Names</td>
<td>88,0%</td>
</tr>
<tr>
<td>Returned Variable Names</td>
<td>84,0%</td>
<td>Attribute Types</td>
<td>72,0%</td>
</tr>
<tr>
<td>Parameter Names</td>
<td>84,0%</td>
<td>Package Names</td>
<td>68,0%</td>
</tr>
<tr>
<td>Parameter Types</td>
<td>84,0%</td>
<td>Returned Variable Names</td>
<td>64,0%</td>
</tr>
<tr>
<td>Local Variable Names</td>
<td>84,0%</td>
<td>Parameter Types</td>
<td>64,0%</td>
</tr>
<tr>
<td>Method Returned Types</td>
<td>80,0%</td>
<td>Local Variable Names</td>
<td>64,0%</td>
</tr>
<tr>
<td>Local Variable Types</td>
<td>80,0%</td>
<td>Parameter Names</td>
<td>60,0%</td>
</tr>
<tr>
<td>Attribute Types</td>
<td>76,0%</td>
<td>Method Returned Types</td>
<td>56,0%</td>
</tr>
<tr>
<td>Source Code Comments</td>
<td>52,0%</td>
<td>Local Variable Types</td>
<td>52,0%</td>
</tr>
<tr>
<td>Package Names</td>
<td>44,0%</td>
<td>Source Code Comments</td>
<td>52,0%</td>
</tr>
<tr>
<td>Exception Names</td>
<td>44,0%</td>
<td>Exception Names</td>
<td>40,0%</td>
</tr>
</tbody>
</table>

(a) A method                  (b) A class

What source code parts are useful for summarizing a method sequence or a package?

Table 5-10 shows that the more relevant places for summarizing methods are also important for describing method sequences. Thus, for most of the participants, *invoked method names, local variable names, parameter names and types, method names, attribute names* and *class names* are at the top of their preferences when summarizing both types of artifacts, with slightly different levels of relevance. Again, the names of variables and parameters are perceived as more important than their types.

On the other hand, packages seem to be more difficult to summarize than other artifacts, and only *package names, class names*, and in some degree, *methods names* were useful information; whereas the rest of the locations were marked as moderately or totally useless. This may explain the tendency to mark a greater amount of terms within this 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.

Table 5-10: Ranking of parts of source code according to the importance for summarizing (a) a method sequence, and (b) a package

<table>
<thead>
<tr>
<th>Location</th>
<th>Percentage of usefulness</th>
<th>Location</th>
<th>Percentage of usefulness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invoked Method Names</td>
<td>100,0%</td>
<td>Class Names</td>
<td>92,0%</td>
</tr>
<tr>
<td>Returned Variable Names</td>
<td>92,0%</td>
<td>Package Names</td>
<td>92,0%</td>
</tr>
<tr>
<td>Parameter Names</td>
<td>92,0%</td>
<td>Method Names</td>
<td>72,0%</td>
</tr>
<tr>
<td>Method Names</td>
<td>88,0%</td>
<td>Attribute Names</td>
<td>56,0%</td>
</tr>
<tr>
<td>Attribute Names</td>
<td>84,0%</td>
<td>Invoked Method Names</td>
<td>56,0%</td>
</tr>
<tr>
<td>Class Names</td>
<td>80,0%</td>
<td>Attribute Types</td>
<td>52,0%</td>
</tr>
<tr>
<td>Method Returned Types</td>
<td>80,0%</td>
<td>Returned Variable Names</td>
<td>52,0%</td>
</tr>
<tr>
<td>Local Variable Names</td>
<td>80,0%</td>
<td>Parameter Names</td>
<td>48,0%</td>
</tr>
<tr>
<td>Parameter Types</td>
<td>76,0%</td>
<td>Parameter Types</td>
<td>48,0%</td>
</tr>
<tr>
<td>Attribute Types</td>
<td>72,0%</td>
<td>Local Variable Names</td>
<td>48,0%</td>
</tr>
<tr>
<td>Local Variable Types</td>
<td>72,0%</td>
<td>Method Returned Types</td>
<td>40,0%</td>
</tr>
<tr>
<td>Package Names</td>
<td>52,0%</td>
<td>Local Variable Types</td>
<td>40,0%</td>
</tr>
<tr>
<td>Exception Names</td>
<td>44,0%</td>
<td>Source Code Comments</td>
<td>36,0%</td>
</tr>
<tr>
<td>Source Code Comments</td>
<td>40,0%</td>
<td>Exception Names</td>
<td>28,0%</td>
</tr>
</tbody>
</table>

(a) A method sequence  (b) A package

We also noticed two prominent results on the responses to the questionnaire:

- Source code comments were not considered particularly valuable, especially for describing packages and method sequences. The lack of comments in two of the systems considered, partially explains this result. Anyway, we expected higher percentages of relevance.

- The names of methods and classes were considered useful when summarizing all four types of artifacts.

5.3.4 Agreement between perceived usefulness and usage in term-based summaries

We compared what subjects perceived as usefulness with how much they used terms from these locations in the term-based summaries. In order to do that, we counted the
number of term-based summaries for which a code location was used or not used in the summary, and the number of subject that perceived that location useful or useless.

We discovered that there is a high agreement between usage and perception of usefulness for method names and invoked method names, and for variable names – the parts of code that turn out to be most widely used. There is also an agreement between low usage and perception of usefulness for exception names. There is a strong disagreement for parameter types, attribute types, and return variable names (perceived useful but not used that much).

These results are very interesting and deserve a detailed discussion we report below, separating cases that reflects what one would expect in summaries, and what, in our opinion, was rather unexpected. The reported percentages indicate the portion of subjects that agree on a specific code location in terms of perceived usefulness. Where needed, for convenience we also report the percentage of usage (in term-based summaries) from Table 5-8.

**Agreement between perception and usage**

- **Class names** were perceived as useful to describe classes by all the subjects in the study. Likewise, class names were considered useful for summarizing packages by 92% of the subjects. Class names actually appeared in 71% of package summaries and in 53% of class summaries.

- **Method names** were perceived as useful for summarizing methods (96% of subjects), classes (96%), method sequences (88%) and packages (72%). These perceptions were confirmed by the actual usage of terms from method names in a very large percentage of summaries for all types of artifacts.

- **Attributes names** were perceived as useful for summarizing methods (92% of subjects), classes (88%), and chains of methods (84%). Their usefulness is confirmed in the term-based summaries for classes, where 53% of summaries contained at least one term from this location, and method sequences (42% of term-based summaries contained at least one attribute name). In the case of packages, they were considered useful for around 56% the subjects and were also moderately used in the package summaries (62%)
• *Local variable names* were perceived as useful for describing methods (84%) and method sequences (80%). Based on their usage, variable names were indeed used often for summarizing methods (81%) and method sequences (81%). Also for the other kinds of artifacts, there was agreement between the perception that variable names are somewhat useful and their usage.

• *Invoked method names* were perceived as useful for describing method sequences by all the subjects and for summarizing methods by 96% of them. With regard to this location we can conclude that there is agreement between perception and usage.

• *Exception names* were considered useless for describing all kinds of artifacts by the majority of the subjects. This was confirmed also by the fact that no exception name was used in any summary.

A high percentage of participants perceived class and method names as useful for describing all kinds of artifacts. The practice confirmed this fact only for method names. Class names were often selected only within package (71%) and class (53%) summaries.

**Disagreement between perception and usage**

These cases are the most interesting, as often researchers try to gain insight into practices by asking developers on what they perceive to be important for a particular task. Our data indicate that there is often disagreement between what developers (all be it juniors) perceive as important and what they actually practice.

• *Attribute types* were not included in any term-based summaries for methods and method sequences, despite them being perceived as useful for these kinds of summaries by 76% and 72% of the subjects, respectively.

• *Method return types* were selected in a mere 2% of the method summaries and in 9% of the method sequence summaries. This significantly contradicts what developers perceived about this location, as they considered them as being very important for summarizing these kinds of artifacts (80%).

• *Names of returned variables* registered the highest disagreement between how people perceived their importance and how they actually used them. They were perceived as highly useful for methods and method sequences, yet only 5% of
summaries for methods and method sequences contain at least one term from the name of a returned variable.

- **Parameter types**, in contrast to parameter names, were only rarely chosen for term-based summaries of classes, methods, and method sequences, despite being perceived as useful by 64%, 84%, and 76% of the subjects, respectively. They were moderately used (29% of the cases) and perceived useful (48%) for summarizing packages.

- **Package names** were used rarely (24% of package summaries), but were considered useful by 92% of the subjects.

- **Comments** were perceived useful for around half of the subjects for summarizing classes and methods (52%). However, they were seldom actually used to summarize classes (7%) and were used only in 10% of method summaries. The general lack of comments in the artifacts is the likely explanation in this case.

As was mentioned above, the results of the survey showed that, roughly speaking, the names of entities are perceived slightly more relevant than their types when summarizing all types of artifacts. The analysis of the term-based summaries showed that this difference, in practice, is much higher since the type of variables, attributes, and parameters were barely used in most of the cases, albeit they were considered highly useful, when summarizing methods and sequences, and moderately useful, for the case of classes. Once again, a likely explanation is that often the types of these entities are basic types of the programming language, such as integer, float or string, and thus, they do not convey domain information.

### 5.4 Qualitative analysis

During the manual analysis of the term-based and sentence-based summaries, we identified some main pieces of evidence (POE), which we describe in this section.

#### 5.4.1 Sentence-based summaries

When analyzing the sentence-based summaries, we noticed that there were no important semantic differences between the summaries written in English and those written in the native language, and they generally conveyed the same message.
For method sequences, we noticed that sometimes they are summarized as a unit, i.e., the subjects described the functionality of the sequence as a whole, while other times each method in the sequence is summarized separately, as summaries follow the control flow in the sequence. For example, in the case of the sequence \textit{run} → \textit{loadRepository} → \textit{navigateDir}, in aTunes system, one subject summarized it as

\begin{quote}
“The sequence loads the music and the pictures from a repository (a directory) recursively and in background.”
\end{quote}

and another one as

\begin{quote}
“The first method alerts that it is reading the repositories and calls the second method; \textit{loadRepository} notifies to the listener how many repositories have to be read and calls the third method; this third method goes through the directory and loads all files and notifies these actions; after that, the first method shows the time spent in the entire process.”
\end{quote}

Both summaries are acceptable, yet follow very different discourse. With the available information, it is not yet possible to draw a clear conclusion about the reasons for these two types of summarization and the cases in which one is preferred over the other; we have seen the same subjects write different types of summaries for different method sequences, and also the same artifact getting the two different types of summaries. We observed the same situation described for method sequences also for packages. There were cases where a package was described as a whole, and other cases where each class in the package was described separately.

\textbf{POE I:} Developers summarize code artifacts by either describing the artifact globally, or by describing each of its components.

One observation refers to the use of synonyms when describing the same software artifact. We noticed that in some cases, the subjects used quite a variety of word choices to describe the same artifact. For example, the verbs \textit{rips, encodes, exports, creates,}
were all used by different subjects to summarize the actions performed by the method *ripTracks* in ATunes. This confirms the findings of Furnas *et al.* (Furnas, *et al.*, 1987), which found that the probability of two people choosing the same words in order to describe the same objects is very low. One interesting note is that the verbs described above act as synonyms in the context of the method *ripTracks*, but they are not synonyms in English. This reinforces another observation made by Sridhara *et al.* (Sridhara, *et al.*, 2010), which showed that the relationships between words in the context of source code are not the same as in the English language. Last, but not least, this explains, in part, the low overlap between sentence and term-based summaries we found in RQ2.

**POE II:** *It is likely that different developers will use different terminology to explain the same concept.*

### 5.4.2 Term-based summaries

While analyzing the term-based summaries, the first observation we made was that a handful of subjects chose some terms in their summaries from locations which we did not expect, and, thus, did not mention in the post-experiment questionnaire. For example, five subjects included Java keywords in their summaries, with some including up to three such keywords in a summary. In addition, some of the subjects included Java keywords in more than one of their summaries. One subject, for example, included a total of six keywords in three of his/her summaries, one class, one package, and one method chain summary. When looking at the keywords s/he selected, we found that she included *null* three times, *return* two times and *new* once. Analyzing the markings that she made in the source code, we discovered that this was due to the fact that s/he selected entire expressions and sometimes an entire line of code, which often were found in the *return statement* of a method. This was confirmed also in the cases of some of the other subjects and it suggests that sometimes developers might consider entire expressions as representative for a source code artifact, not only independent identifiers or words. We plan to investigate coarser levels of granularity for the constituent elements of a summary in future work.

Another interesting fact concerns another category, not included in our original questionnaire, i.e., constants. We observed that in more than one third of the term-based summaries, subjects included at least one term coming from either literal constants, such
as, string constants, or from Java variables declared using the \textit{public static final} modifiers. In the first case, we noticed that literal constants sometimes contained important domain information, which was leveraged by the subjects, which selected it. For example, in method \texttt{persistRepositoryForFuture}, in ATunes, one of the messages sent to the logger was "\textit{Storing repository information...}". This described well the functionality of the method, and, in consequence, it was included in the term-based summaries by some of the subjects. In the case of the \texttt{final} variables, we noticed that in some cases, these "constants" stored important relevant information about the artifact, which was used quite often in the source code. For example, all 15 term-based summaries available for the \texttt{LanguageTool} class, in ATunes, contained at least one term from either the name or the string value attributed to the two constants defined in the class, i.e., \texttt{TRANSLATIONS_DIR} and \texttt{DEFAULT_LANGUAGE_FILE}. These constants store the directory where the translations of the UI in different languages are stored and the default language file, respectively. These are important and representative concepts for the class, which manages the translations of the UI in the different languages supported in the program. The example presented here illustrates a situation where constants can play an important role in the summary of an artifact.

\textbf{POE III}: Summaries may contain entire (small) portions of source code, such as expressions, programming language keywords, or literals.

\section{5.5 Threats to validity}

This section discusses threats to validity that can affect the results of our study.

\textit{Construct validity} threats concern the relationship between theory and observation. Where measurements were performed automatically, we manually checked a sample of the summaries to assess the correctness of our scripts. The part-of-speech classification was performed with \textit{TreeTagger}, which has a precision of about 85\% for English text. Manual measurements – e.g., for determining code locations for term-based summaries – were performed by two different persons to limit mistakes. Finally, the survey questionnaires were prepared using a Likert scale to facilitate their evaluation.
This is an exploratory study, thus we do not discuss threats to internal validity, as we are interested in observing different variables from the summaries, rather than capturing relations between variables. Nevertheless, we limited the effect of lack of comprehension on the summary quality by removing from the sample incorrect summaries. Although an analysis of co-factors is not reported here, the analyses we made did not reveal substantial differences of interactions of the variables with co-factors.

Conclusion validity is not relevant for exploratory studies, even though wherever possible we support our claims with statistical tests, and we mainly use non-parametric tests, which do not make assumptions on the underlying data distribution.

External validity threats concern the generalization of our findings. Although our study involved three different systems belonging to different domains, further replications on more systems, especially on systems developed in languages different from Java are needed. This study was conducted with students. Although they are all graduate and senior undergraduate students, with experience level comparable to junior programmers, it would be useful to replicate the study with professionals. It is important to note that the difference between students and professionals is not always easy to identify.

### 5.6 Approximation to gold standard summaries

The set of term-based summaries provided by developers could be used to generate gold standard summaries. One way to do this is to define the relevance of a term as the number of developers that selected it in their summaries, and then, rank all the terms selected by developers according to their relevance. The top $n$ terms in this list would be the set of terms that convey the most important information about the purpose of the artifact. The data reported in Table 5-5, where the median length of term-based summaries for different types of artifacts ranges between 8 and 17 terms, is useful to choose a suitable value for $n$. Table 5-11 shows the gold standard summary for the method ripTracks in aTunes, with $n=10$. The last column of this table shows the intersection of the summary with the sentence-based summaries of this artifact. For example, the term tracks were selected by 90% of developers and it appears in 54% of the sentence-based summaries written by them.
Table 5-11: Gold standard summary for the method ripTracks

<table>
<thead>
<tr>
<th>Term</th>
<th>Relevance</th>
<th>% of SB-summaries where it appears</th>
</tr>
</thead>
<tbody>
<tr>
<td>rip</td>
<td>100</td>
<td>18</td>
</tr>
<tr>
<td>tracks</td>
<td>90</td>
<td>54</td>
</tr>
<tr>
<td>file</td>
<td>90</td>
<td>36</td>
</tr>
<tr>
<td>wav</td>
<td>60</td>
<td>54</td>
</tr>
<tr>
<td>result</td>
<td>60</td>
<td>27</td>
</tr>
<tr>
<td>folder</td>
<td>50</td>
<td>45</td>
</tr>
<tr>
<td>encode</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>encoder</td>
<td>40</td>
<td>9</td>
</tr>
<tr>
<td>titles</td>
<td>30</td>
<td>18</td>
</tr>
<tr>
<td>listener</td>
<td>30</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5-12 presents how relevant the terms in gold standard summaries are, for the case of Bogota group, with n=10. According to these results, the average relevance ranges from 53% to 98%, and in average, 73% of the terms in gold standard summaries appear in at least one sentence-based summary for the same artifact. This percentage could increase if we take into account the use of synonyms in the sentence-based summaries; as a case in point, the term encode found in the method ripTracks, was replaced by words such as transform and convert in some summaries.

Table 5-12: Relevance of terms in gold standard summaries and intersection with sentence-based summaries

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Minimum relevance</th>
<th>Maximum relevance</th>
<th>% of terms in at least 1 SB-summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>method1A</td>
<td>30</td>
<td>100</td>
<td>80</td>
</tr>
<tr>
<td>method1B</td>
<td>40</td>
<td>100</td>
<td>70</td>
</tr>
<tr>
<td>class1A</td>
<td>60</td>
<td>100</td>
<td>80</td>
</tr>
<tr>
<td>class1B</td>
<td>55</td>
<td>100</td>
<td>80</td>
</tr>
<tr>
<td>chain1A</td>
<td>70</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>chain1B</td>
<td>50</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>package1A</td>
<td>77</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>method2A</td>
<td>20</td>
<td>80</td>
<td>40</td>
</tr>
<tr>
<td>method2B</td>
<td>40</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>class2A</td>
<td>60</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>class2B</td>
<td>60</td>
<td>100</td>
<td>70</td>
</tr>
<tr>
<td>chain2A</td>
<td>40</td>
<td>100</td>
<td>70</td>
</tr>
<tr>
<td>chain2B</td>
<td>80</td>
<td>100</td>
<td>80</td>
</tr>
<tr>
<td>package2A</td>
<td>60</td>
<td>100</td>
<td>60</td>
</tr>
</tbody>
</table>
The level of relevance and the high intersection with sentence-based summaries indicate that formal intrinsic evaluation of summarization approaches can be performed using gold standard summaries obtained by merging summaries provided by multiple developers. Since these summaries represent the core of both types of abstracts, they would be the main target of the automatic summarizers based on extractive approaches.

5.7 Conclusions

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. When summarizing complex artifacts, such as packages or method sequences, subjects follow different rules of discourse, either by providing a comprehensive description of the whole or by describing individual component of the artifact. In regard to the used parts-of-speech (i.e., nouns, verbs, adjectives, adverbs), sentence-based summaries are written as plain English documents, while term-based summaries tend to mainly use nouns, representing entities, or verbs when an action needs to be described.

The percentage of intersection between sentence-based summaries and the artifacts is around 40%, and it is even lower in the case of packages. One likely explanation of this is that in source code often there are lots of duplicates (Much more than in natural language texts). Thus, the number of unique terms (the vocabulary) within an artifact does not increase with its length, i.e., there is no positive correlation between length and the number of unique terms.

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 relatively 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 analysis of the provenance of terms used in the summaries is the most important part of the study. We found that for summarizing a method the main sources of information are its name, call methods names and local variable names; for a class, its name, the names of its methods and the call methods; and for a method sequence, the names of the methods in the chain, the names of other invoked methods, local variable names, and also attribute names are moderately relevant. Finally, the results related to packages are less clear. Apparently, for summarizing a package the most useful sources of information are the class names. Local variable names and attribute names are also somewhat useful. 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.

The observations we made can be used as building blocks to define guidelines aimed at explaining to developers how to describe code artifacts when communicating with project newcomers, but, possibly, also heuristics to be used in the design or improvement of automatic code summary generators.

Finally, the results obtained also represent valuable information for evaluating automatic summarization tools. The gold-standard summaries obtained in this study contain the most relevant terms in the intersection between term-based and sentence-based summaries, and therefore, could be considered as building blocks to design formal evaluations of automatic summarization tools.
6. Automatic Summarization of Source Code Artifacts

6.1 Introduction

As we presented in chapter 2, automatic text summarization is concerned with the production of a brief but accurate representation, called a summary, of one or more source documents, with the help of a computer program. Summaries need to be significantly shorter than the original document, while preserving the most important information in the document. Summaries can be divided in two main categories: extractive and abstractive. Extractive summaries are obtained from the contents of a document by selecting the most important information in that document. Abstractive summaries, on the other hand, are meant to produce important information about the document in a new way, at a higher level of abstraction, and usually include information which is not explicitly present in the original document.

This chapter presents three completely different approaches to generate automatic summaries of software artifacts. In section 6.2 we investigate an extractive term-based summary, which contains the most relevant terms for describing the purpose of a source code artifact, i.e., methods and classes. In section 6.3 we propose a technique to generate an indicative sentence-based summary for classes, and finally, in section 6.4 we describe an approach to automatically extract method descriptions from communications in bug tracking systems and mailing lists.

6.2 Term-based summaries of methods and classes

A wide variety of techniques have been proposed for producing summaries in text summarization. Some of the most successful ones include techniques based on the position of words or sentences in the source document, and techniques based on text retrieval (TR). Among the techniques based on the position of terms, the lead summaries
are the most frequently used and most successful, being often selected as a baseline for assessing new techniques. Lead summaries are based on the idea that the first terms that appear in a document (i.e., the *leading terms*) are the most relevant to that document.

Statistical based text retrieval (TR) techniques have also been successfully used for text summarization [6-8]. In this section we assess the suitability of several summarization techniques, mostly based on TR methods, to capture source code semantics in a way similar to how developers understand it. We present a case study that aims at determining how each technique impacts the quality of the summaries produced. To achieve this goal, we developed a framework that allows us to use TR techniques to generate automated source code summaries. While obtaining the lead summaries is straight-forward, the framework for the generation of TR summaries is more complex, having two main components, with the following aims:

- Extract the text from the source code of the system and convert it into a corpus.
- Determine the most relevant terms for documents in the corpus and include them in the summary.

Each of the two components can be implemented in various ways, which we discuss here.

### 6.2.1 Automatic generation of Term-based summaries

#### Source code corpus creation

Source code contains a lot of text, yet it is not entirely natural text. In order to use text retrieval techniques on source code, we need to convert it into a document collection. Within OO software development, methods and classes are typical examples of documents, though one can think of files and packages as well. In each case, the identifiers and comments in the source code entities are extracted. The next step, which is optional, is splitting the identifiers according to common naming conventions. For example, “setValue”, “set_value”, and “SETvalue”, are all split into “set” and “value”. If identifiers are split, there is a choice whether to keep the original (unsplit) version of the identifier or not. The last step (typical in all TR applications) is using a stop-words list to filter out terms which do not carry specific meaning. Such terms are conjunctions,
prepositions, articles, common verbs, pronouns, etc. (e.g., “should”, “may”, “any”, etc.) and programming language keywords (e.g., if, else, for, while, float, char, void, etc.).

**Generating source code summaries using text retrieval**

Once a corpus is created, several TR techniques can be used to generate the summaries. We implemented two variants: one based on the Vector Space Model (VSM) (Salton, et al., 1974), and the second on Latent Semantic Indexing (LSI) (Deerwester, 1988). Both techniques have been shown to work well for the summarization of natural language documents (Gong and Liu, 2001) (Steinberger and Ježek, 2009) (Kireyev, 2008). They have also been used to index software corpora in a variety of comprehension applications (Antoniol, et al., 2000) (Gay, et al., 2009) (Marcus, et al., 2004).

While the two techniques generate summaries differently, the first step is the same in each case: represent the terms and documents in the corpus in a matrix where each row represents a term and each column represents a document. The content of a cell in this matrix represents the weight of a term (the row) with respect to a document (the column). These weights are generally a combination between a local weight and a global weight. The local weight captures the relationship between a term and a particular document and it is usually derived from the frequency of the term in the document. The global weight refers to the relationship of a term with the entire corpus and it is usually derived from the distribution of the term in the corpus. There are several options to choose from for both the local and global weights, resulting in numerous combinations. We identified the three combinations that perform the best in natural language summarization: log, tf-idf, and binary-entropy, for both LSI (Gong and Liu, 2001) (Steinberger and Ježek, 2009) and VSM (Liu, et al., 2009) and used them in our study.

For generating a summary using VSM, the terms in the document (i.e., class or method) are ordered according to the chosen weight and the top K terms are included in the summary. LSI does not operate on the original term by document matrix, but it projects it to a smaller one which approximates it, corresponding to the highest Eigen values in the original matrix. Thus, LSI allows the representation of terms and documents in the same coordinates. In consequence, it allows computing similarities between terms and documents. These similarities are used to obtain the list of the most similar, i.e., most relevant terms to a document, which are then included in the summary. First, the cosine
similarities between the vectors of each term in the corpus and the vector of the document to be summarized are computed. Then, the terms are ordered based on this similarity and the top \( K \) terms, having the highest similarity to the document are included in the summary. One particular feature of the top \( K \) list of terms generated in this manner with LSI is that it can contain also terms that do not appear in the summarized method or class, but appear somewhere else in the corpus. When a summary contains such terms, it can be interpreted as a lightweight abstractive summary, as it produces information about the original document in a new form, i.e., which is not contained in the document.

6.2.2 Quality assessment of term-based summaries: a case study

To evaluate the quality of the automatically produced source code summaries and to measure their capability to capture the developers’ understanding of the code, we performed a study in which developers judged the quality of a large set of summaries for methods and classes. The study was particularly aimed at assessing the impact of the various factors affecting the generation of code summaries (e.g., corpus creation, TR technique, term weight used, summary length, etc.) on the summary quality.

Subjects and objects of the study

In this study we asked four computer science students (coded as D1-D4) to evaluate code summaries automatically generated using VSM and LSI for methods and classes from two Java software systems. Each student has several years of programming experience.

The two Java systems we used in the case study are both available for download from Sourceforge. The first system, aTunes (version 1.6), is a full-featured media player and manager having 221 classes, 1,852 methods, and 25,000 non-blank lines of text in its source code (including 5,000 lines of comments). The second system, Art of Illusion\(^7\) (version 2.7.2), is a 3D modeling, rendering, and animation studio. It has 597 classes, 7,057 methods, and almost 100,000 non-blank lines of text (including 15,000 lines of comments).

\(^7\) http://www.artofillusion.org/
For each system, we chose a set of 10 methods and their corresponding classes from the source code. The methods were chosen in such a way that they form a heterogeneous set for each system, based on the following characteristics of their implementation: length in lines of text (we chose methods from three categories: less than 20 lines, between 20 and 50 lines, and more than 50 lines), presence of comments, presence of parameters, type of class they belong to (entity, control, boundary), the presence of a return statement, the number of actions the method implements (one or more than one), and the stereotype of the method (accessor, mutator, creational, other).

Types of summaries
We generated summaries for each class and method using four different summarization techniques: lead, VSM, LSI, and a baseline technique, i.e., random summaries. The random summaries are formed by randomly selecting K terms from the identifiers and comments of a method or class and including them in the summary. For each of these techniques, we varied a series of parameters and observed the effect that these variations have on the quality of the generated summaries.

For all techniques we varied the number of terms (K) included in the summary. We generated summaries with 5 and 10 terms for each technique, as we believe that fewer than 5 terms capture too little information and having more than 10 terms pushes our short term memory limits (Miller, 1956).

For VSM and LSI we used the three weighting schemes mentioned before (i.e., log, tf-idf, and binary-entropy) for the terms in the term-by-document matrix. The terms and order of these terms in the summaries depend on the weight used. For the lead and random summaries there is no need to use such weights.

We also used three different ways to generate the corpora: (1) splitting identifiers and discarding the original form; (2) splitting identifiers and keeping the originals; (3) not splitting the identifiers. Applying all these variations lead to the generation of 66 different summaries for each of the 40 methods and classes in the two systems, for a total of 2,640 summaries. These summaries were then analyzed and evaluated by the developers.
The evaluation procedure
To measure the quality of the automatic summaries, we performed an intrinsic online evaluation, which is one of the standard evaluation approaches used in the field of text summarization (Jones, 2007). This evaluation involves the active participation of human judges, who rate each of the automatic summaries based on their own perception of its internal quality.

In a first phase, the developers spent several days to familiarize themselves with the two software systems, focusing specifically on the methods and classes that were going to be summarized. Three of them had previous experience with adding features and fixing bugs in one of the two systems.

In the next phase, the four developers were presented with the 66 summaries for each method and each class and asked to answer the question “How much do you agree with this automated summary of the method/class?” In order to limit the possibility of a placebo effect on the participants, they were not presented with any information that would disclose the summarization technique or the parameters used to produce any of the summaries. The developers had four options to choose from to answer the question, each being assigned a score: strongly agree (assigned a score of 4), agree (score of 3), disagree (score of 2), and strongly disagree (score of 1). This set of possible answers represents a four level Likert scale.

In order to be able to understand the results better, in addition to choosing one of the above options for rating each summary, we also asked the developers to tag each of the terms that appear in a summary as either relevant or irrelevant for the method or class under analysis. The developers were provided with a form for each method or class, which contained the summaries, each followed by the list of terms in the summary. Next to each term in this list there was an empty box where developers were asked to write a 0 when they considered the term irrelevant, and a 1 when the term was relevant. The developers had access to the source code of both systems at all times and they were not given a time limit to finish the evaluation. To minimize the evaluation bias, developers were not given any other instructions on assessing the summaries or the relevance of the terms.
Follow-up questionnaire

In order to understand how the developers performed the evaluation and get more insight into the results of the study, we asked the developers to answer three follow-up questions. We report here on the answers we received from developers in the follow-up questionnaire in order to better understand the results that follow.

The first question we asked was: *What was the process you followed when rating the summaries?*

Two of the developers, D1 and D2, followed very similar processes when rating summaries. They started by extracting a list of relevant words from the source code, i.e., the words they considered should appear in the summary. Then, they rated all the summaries based on the terms they had in mind and at the end they marked the terms as relevant or irrelevant.

D3 and D4 did not extract the relevant terms from the source code. D3 analyzed each summary and its terms individually, rating the summary first and then marking each of the terms as relevant or irrelevant. D4, on the other hand, first compiled the list of unique terms found in all summaries of a class/method and marked in this list the ones that were relevant. Based on this list, D4 then marked all terms in all summaries as relevant or irrelevant and then rated the summaries. All developers reported changing the initial ratings of some of the summaries when they compared summaries between them.

We observed that there is no connection between the process followed by developers to rate summaries and the scores they gave. For example, the pairs of developers (D2, D4), and (D1, D3), who followed different processes, had more similar ratings than (D1, D2), who followed a similar process for rating summaries.

The second question was: *What criteria did you use for rating the summaries?*

D1 considered the number of irrelevant terms contained in a summary as an important factor in the evaluation, whereas D2, D3, and D4 did not. This is corroborated by the data we collected, which reveals that D1 gave higher scores more often to 5-term summaries than to 10-terms summaries (10 times more often than D2, 12 times more often than D3, and 4 times more often than D4), as the latter contained more irrelevant terms. The other developers were less influenced by the noise in the summaries and focused on the
relevant terms hence they rarely gave higher ratings to 5-terms summaries than to 10-terms summaries. The fact that D1 considered the irrelevant terms in his ratings led to a decrease in the average score, mode and sum of scores of his ratings for all summaries. For example, even if he gave the highest scores to the lead summaries among all summarization techniques, the highest average score for classes was 2.47, compared to 3, 3.06 and 3.59, as given by the other developers. In average, he did not consider any type of summary very good. For methods, his average rating of the lead summaries was 2.75, compared to 3.25, 3.10, and 3.70, as given by the other developers. More details on these scores are presented in the next section.

D4 reported that the order of the terms in the summary was important at times, as the summary was clearer when the relevant terms followed one another and when they were positioned at the beginning of the summary. D4 used the position of the relevant terms to differentiate between good and very good summaries. The other developers said the order of the terms in the summary did not impact their ratings.

When analyzing method summaries, all developers considered the method name to be the most relevant piece of information and ranked summaries having this idea in mind. When scoring class summaries, D2 and D4, considered that the class name and attribute names were the most relevant information for a class and therefore should be present in the summary of a class. The other two developers considered that the names of the methods defined in the class are more relevant. By analyzing the scores given by each developer to the methods and classes in the case study, we found that D2 and D4, agreed the most in their ratings among all pairs of developers, for both classes and methods.

All developers reported noticing changes in their rating criteria between different scoring sessions and attributed them to changes in mood and fatigue. Also, D2 reported that once during the same session, when he did not order the summaries according to their first terms, he sometimes assigned slightly different scores to the same summary, when this appeared more than once among provided summaries (this happens when two different techniques generate the same summary for a method or class). For example, he remembered scoring the same summary with both a score of 3 and a score of 4 before
ordering the summaries according to their content, but then realizing his inconsistency and synchronizing the scores.

The third question we asked was: What was missing (if anything) from the summaries you analyzed? This question helps us understand what should be included in a summary, besides what was already mentioned as important in Question 2. D1, D2, and D4 reported that they would always include good comments that appear before the class/method definition in the summaries. D3 thinks that if the identifiers are good, the comments are not needed. Even if this type of comments were mostly included in the lead summaries when available, they were generally not found in the other types of summaries. The fact that developers appreciate the leading comments could be one of the reasons for the good scores obtained by lead summaries, as discussed in the next sections.

All developers reported that they would always include at least a verb and an object in method summaries, which describe the action performed by the method and the object on which it is performed. Our future work on automatic summarization of source code will take these observations into account, as it will try to produce summaries as close as possible to what developers would like to see.

Results and discussion
The six lead summaries scored higher than any other individual technique (Table 6-1). This indicates that the terms found at the beginning of a class or method are generally relevant terms for the artifact. The maximum average score obtained by lead summaries when considering both methods and classes is 3.09, which is achieved for 10 term-summaries and unsplit identifiers. The minimum score obtained by lead summaries for both methods and classes is 2.68, and is achieved for summaries having 5 terms and where the identifiers were split and the originals were not kept.
Table 6-1: The 24 highest ranked types of summaries

The format of the technique name is AaNnBbbCc, where
Aa=technique (Le=Lead, Ls=LSI, Vs=VSM)
NN=number of terms (5 or 10)
Bbb=corpus splitting technique (Spn=Split No Originals, SpO=Split Keep originals, NSp=No Split)
Cc=weighting (Tf=tf-idf, Lg=log, Be = binaryEntropy)
M = methods, C= classes, MC = methods and classes

<table>
<thead>
<tr>
<th>#</th>
<th>Technique name</th>
<th>Average score</th>
<th>St. Deviation</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>M</td>
<td>C</td>
<td>MC</td>
</tr>
<tr>
<td>1</td>
<td>Le10Nsp</td>
<td>3.25</td>
<td>2.94</td>
<td>3.09</td>
</tr>
<tr>
<td>2</td>
<td>Le10Spn</td>
<td>3.28</td>
<td>2.90</td>
<td>3.09</td>
</tr>
<tr>
<td>3</td>
<td>Le10Spo</td>
<td>3.21</td>
<td>2.86</td>
<td>3.04</td>
</tr>
<tr>
<td>4</td>
<td>Le5Nsp</td>
<td>3.08</td>
<td>2.45</td>
<td>2.76</td>
</tr>
<tr>
<td>5</td>
<td>Le5Spn</td>
<td>2.98</td>
<td>2.40</td>
<td>2.69</td>
</tr>
<tr>
<td>6</td>
<td>Le5Spo</td>
<td>2.98</td>
<td>2.39</td>
<td>2.68</td>
</tr>
<tr>
<td>7</td>
<td>Vs10SpoBe</td>
<td>2.43</td>
<td>2.34</td>
<td>2.38</td>
</tr>
<tr>
<td>8</td>
<td>Vs10SpoLg</td>
<td>2.40</td>
<td>2.31</td>
<td>2.36</td>
</tr>
<tr>
<td>9</td>
<td>Vs10NspTf</td>
<td>2.35</td>
<td>2.36</td>
<td>2.36</td>
</tr>
<tr>
<td>10</td>
<td>Vs10SpnLg</td>
<td>2.35</td>
<td>2.31</td>
<td>2.33</td>
</tr>
<tr>
<td>11</td>
<td>Vs10NspLg</td>
<td>2.33</td>
<td>2.28</td>
<td>2.30</td>
</tr>
<tr>
<td>12</td>
<td>Vs10SpnBe</td>
<td>2.39</td>
<td>2.19</td>
<td>2.29</td>
</tr>
<tr>
<td>13</td>
<td>Vs10SpoTf</td>
<td>2.30</td>
<td>2.24</td>
<td>2.27</td>
</tr>
<tr>
<td>14</td>
<td>Vs10SpnTf</td>
<td>2.26</td>
<td>2.23</td>
<td>2.24</td>
</tr>
<tr>
<td>15</td>
<td>Vs10NspBe</td>
<td>2.25</td>
<td>2.19</td>
<td>2.22</td>
</tr>
<tr>
<td>16</td>
<td>Vs10SpoTf</td>
<td>2.38</td>
<td>2.06</td>
<td>2.22</td>
</tr>
<tr>
<td>17</td>
<td>Vs10NspLg</td>
<td>2.39</td>
<td>2.01</td>
<td>2.20</td>
</tr>
<tr>
<td>18</td>
<td>Vs10NspTf</td>
<td>2.33</td>
<td>1.96</td>
<td>2.14</td>
</tr>
<tr>
<td>19</td>
<td>Vs10SpnBe</td>
<td>2.29</td>
<td>1.95</td>
<td>2.12</td>
</tr>
<tr>
<td>20</td>
<td>Vs10NspLg</td>
<td>2.10</td>
<td>2.04</td>
<td>2.07</td>
</tr>
<tr>
<td>21</td>
<td>Vs10SpoLg</td>
<td>2.24</td>
<td>1.84</td>
<td>2.04</td>
</tr>
<tr>
<td>22</td>
<td>Vs10SpoTf</td>
<td>2.03</td>
<td>2.03</td>
<td>2.03</td>
</tr>
<tr>
<td>23</td>
<td>Vs10SpnLg</td>
<td>1.95</td>
<td>2.09</td>
<td>2.02</td>
</tr>
<tr>
<td>24</td>
<td>Vs10SpnTf</td>
<td>2.21</td>
<td>1.79</td>
<td>2.00</td>
</tr>
</tbody>
</table>

After analyzing the summaries and the developers’ answers to the post-experiment questionnaire, we concluded that the high scores of the lead summaries are due to the terms in the headers of methods, the names of classes, and sometimes good leading comments, which are included in these summaries and considered important by developers. Among the TR techniques, the top 9 average scores (for both methods and
classes) are obtained by the 10-term VSM generated summaries with a variety of options (see rows 7 to 15 in Table 6-1)

For analyzing the difference in performance of the techniques over the entire data, we averaged the scores over all artifacts, developers, and parameter configurations. Lead summaries got an average of 2.89, VSM summaries 2.11, LSI summaries 1.75, and the random summaries 1.56 (remember that 1 corresponds to the strongly disagree rating and 4 to the strongly agree rating). We applied a statistical test to compare the means of the different techniques. Based on the characteristics of the arrays of values for the different summarization techniques (normal distribution, dependent values), we used a two-tailed paired t-test for this purpose. The resulted p-values were 0.02 for the pair of LSI-random summaries and 0.00 for all the other pairs, indicating that the differences between summarization techniques are statistically significant.

We also looked at the differences in the ratings given by developers between the two software systems. We found that the differences in ratings were minor and that there was a very high correlation between the scores assigned to artifacts in the two systems (0.98).

1) Combining the lead and VSM summaries

While the scores of the lead summaries are good, they also indicate there is room for improvement, i.e., there is relevant information not captured by the lead summaries. The TR summarization technique with the highest scores, VSM, is also far from perfect. However, the data support the observation that the VSM summaries contain relevant information valued by the developers, even if not as much and not the same as lead summaries do. One obvious missing term, for example, is the method name. VSM favors terms with high frequency in the document and in most cases method names, for example, appear only once in a method. It is likely that a VSM based summary will not include them.

We analyzed the VSM and lead summaries to see if any of the relevant terms they capture are not included in the other type of summaries. More specifically, we computed the intersection and union of the relevant terms between each VSM configuration and each lead configuration (Table 6-2) across all the summarized methods and classes in the two systems. We found that the number of relevant terms in the union is always greater
than in each of the two techniques considered individually (in average, twice as many relevant terms as the lead summaries) and the intersection of relevant terms between VSM and lead summaries is empty in 1/3 of the cases. This fact strongly suggests that lead and VSM summaries capture different relevant information about the methods and classes.

<table>
<thead>
<tr>
<th></th>
<th>Lead</th>
<th>VSM</th>
<th>Lead ∪ VSM</th>
<th>Lean ∩ VSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>6.26</td>
<td>5.68</td>
<td>10.14</td>
<td>1.80</td>
</tr>
<tr>
<td>C</td>
<td>5.50</td>
<td>5.53</td>
<td>9.81</td>
<td>1.22</td>
</tr>
<tr>
<td>MC</td>
<td>5.88</td>
<td>5.60</td>
<td>9.98</td>
<td>1.51</td>
</tr>
</tbody>
</table>

As the number of relevant terms in summaries is strongly correlated with the scores obtained by the summaries (0.82 Spearman rank correlation), we expect that if the number of relevant terms in a summary increases, so does its score. Based on the observations we made, our hypothesis was that combining the lead and VSM summaries will lead to better summaries and higher rankings from the developers compared to the individual summaries.

In order to study if this is indeed the fact, we combined the lead and VSM summaries by concatenating to the lead summaries the terms in the VSM summaries which did not already exist in the lead one. We obtained 18 new summaries for each artifact, as we performed every combination between lead and VSM summaries having the same number of terms. The same four developers ranked each of the new summaries using the same procedure they used for ranking the individual summaries.

The results (Table 6-3) show that our expectations were met and that the combined summaries obtained higher scores than the individual summaries, some coming close to the maximum score. The highest average score across all methods and classes was 3.54, achieved for the union of the 10-term lead and VSM summaries, without splitting the identifiers and using tf-idf as the weight for VSM. The improvement in score was almost half a point, which made the transition from a good summary to an excellent summary.
Table 6-3: The Lead+VSM summaries and their scores

The format of the technique name is AaNnBbbCc, where:

Aa=technique (Le=Lead, Ls=LSI, Vs=VSM)

NN=number of terms (5 or 10)

Bbb=corpus splitting technique (Spn=Split No Originals, SpO=Split Keep originals, NSp=No Split)

Cc=weighting (Tf=tf-idf, Lg=log, Be = binaryEntropy)

M = methods, C= classes, MC = methods and classes

<table>
<thead>
<tr>
<th>#</th>
<th>Technique name</th>
<th>Average score</th>
<th>St. Deviation</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>M</td>
<td>C</td>
<td>MC</td>
</tr>
<tr>
<td>1</td>
<td>LeVs10NspTf</td>
<td>3.66</td>
<td>3.43</td>
<td>3.54</td>
</tr>
<tr>
<td>2</td>
<td>LeVs10NspLog</td>
<td>3.61</td>
<td>3.4</td>
<td>3.51</td>
</tr>
<tr>
<td>3</td>
<td>LeVs10NspBe</td>
<td>3.59</td>
<td>3.38</td>
<td>3.48</td>
</tr>
<tr>
<td>4</td>
<td>LeVs10SpoBe</td>
<td>3.55</td>
<td>3.28</td>
<td>3.41</td>
</tr>
<tr>
<td>5</td>
<td>LeVs10SpoTf</td>
<td>3.55</td>
<td>3.26</td>
<td>3.41</td>
</tr>
<tr>
<td>6</td>
<td>LeVs10SpoLg</td>
<td>3.48</td>
<td>3.29</td>
<td>3.38</td>
</tr>
<tr>
<td>7</td>
<td>LeVs5NspBe</td>
<td>3.35</td>
<td>3.04</td>
<td>3.19</td>
</tr>
<tr>
<td>8</td>
<td>LeVs10SpnLg</td>
<td>3.26</td>
<td>3.08</td>
<td>3.17</td>
</tr>
<tr>
<td>9</td>
<td>LeVs10SpnTf</td>
<td>3.25</td>
<td>3.08</td>
<td>3.16</td>
</tr>
<tr>
<td>10</td>
<td>LeVs10SpnBe</td>
<td>3.28</td>
<td>3.05</td>
<td>3.16</td>
</tr>
<tr>
<td>11</td>
<td>LeVs5NspTf</td>
<td>3.36</td>
<td>2.96</td>
<td>3.16</td>
</tr>
<tr>
<td>12</td>
<td>LeVs5NspLg</td>
<td>3.28</td>
<td>2.91</td>
<td>3.09</td>
</tr>
<tr>
<td>13</td>
<td>LeVs5SpoBe</td>
<td>3.19</td>
<td>2.81</td>
<td>3.00</td>
</tr>
<tr>
<td>14</td>
<td>LeVs5SpoTf</td>
<td>3.16</td>
<td>2.8</td>
<td>2.98</td>
</tr>
<tr>
<td>15</td>
<td>LeVs5SpoLg</td>
<td>3.1</td>
<td>2.8</td>
<td>2.95</td>
</tr>
<tr>
<td>16</td>
<td>LeVs5SpnLg</td>
<td>3.01</td>
<td>2.74</td>
<td>2.88</td>
</tr>
<tr>
<td>17</td>
<td>LeVs5SpnTf</td>
<td>3.01</td>
<td>2.71</td>
<td>2.86</td>
</tr>
<tr>
<td>18</td>
<td>LeVs5SpnBe</td>
<td>3.01</td>
<td>2.63</td>
<td>2.82</td>
</tr>
</tbody>
</table>

In all cases, the combined summaries obtained higher average scores than each of the individual summaries, and at times the improvement was substantial. The lead summaries benefitted from the addition of the terms from the VSM summaries and the improvement averaged 0.28 over all artifacts and all six types of lead summaries. The highest improvement was in the case of the 10-terms summaries without identifier splitting, which had previously scored the highest among the individual summaries. In this case, the score of the new, combined summary indicates that developers agree highly with the new summary, suggesting that lead+VSM summaries are a good baseline for the automatic summarization of software artifacts. The lead+VSM summaries gained most over the lead summaries in the cases of classes and less in the case of methods (Table 6-3). This can be explained by the fact that the method lead summaries already contain parts of the
header of a method and/or explanatory leading comments, which are all considered very relevant by developers. In the case of classes, on the other hand, the lead summaries contain the name of the class and/or leading comments and the first few attributes in the class, which are not always the most relevant for the class. Adding extra relevant terms by the addition of the VSM summaries increases the quality of the summaries.

Theoretically, the learning resulted from the first summarization session could lead in a difference in the judging and rating of summaries in the second summarization session and impact the results. However, the time passed between the session when they rated the individual summaries and the one when they rated the combined summaries was significant (almost four months), which was enough to mitigate the effect of learning. This was corroborated also by the fact that two of the developers reported having some difficulties understanding one class and one method, respectively, before the second summarization session, even though they had no problems understanding them before the first session. They eventually understood the class and method well, but they spent considerably more time understanding them in the second session than in the first one. This indicates that the learning effect, if any, was neutralized between the two sessions.

2) Methods vs. classes
Lead summaries received better scores for methods than for classes (see Table 6-1). This can be explained by the fact that for the lead method summaries, the four developers agreed more on the type of terms that should be included in the summary. In the case of classes, however, besides the name of the class, which was considered important by all four developers, there were different opinions on what should be in the summary. D2 and D4 put emphasis on the name of class attributes, while D1 and D3 considered the name of methods to be more important. The name of the methods, though, were rarely included in the class lead summaries, as opposed to the attribute names, which led to the lower rating of these summaries by D1 and D3. For example, for the class AmazonService in ATunes, we found that D2 and D4 gave a score of 3 to all the lead summaries having 5 terms, which contained the name of the class and one or two of the names of attributes in the class. On the other hand, developers D1 and D3 both assigned a score of 2 to all these summaries.
The other summarizing techniques also performed better on methods than on classes, even though the differences between methods and classes were not as high as in the case of lead summaries, where this difference was 0.47 on average. In the case of VSM, the difference was 0.08 on average, for LSI this difference was 0.17, and 0.34 for random summaries.

For the lead+VSM summaries we observed the same phenomenon: the summaries for methods received higher scores than the summaries for classes, for all combinations of lead and VSM summaries. This is not surprising, due to the higher scores for methods received by each of the techniques independently. We also noticed that the difference between the scores of the method summaries and the scores of the class summaries are higher for the union of the short summaries, i.e., 5-term summaries. This is because, in the case of methods, 5 terms are often enough for the lead summaries to include most of the header of a method, which contains relevant terms according to the developers. In the case of classes, however, the first 5 terms are often not enough to capture the most relevant information, hence the lower scores.

3) 5-term vs. 10-term summaries
By construction, the 10 terms summaries always contain the 5 terms summaries, followed by other 5 terms. When comparing two summaries obtained using the same technique and the same parameters, the 10 terms summaries are always ranked higher than the 5 terms summaries. Table 6-4 shows an aggregated view of these results, presenting the average scores for all techniques across all artifacts for 5 and 10 terms, respectively. The last column shows the results of the union between lead and VSM summaries, when each of them contains 5 or 10 terms.

<table>
<thead>
<tr>
<th></th>
<th>Lead</th>
<th>VSM</th>
<th>LSI</th>
<th>Random</th>
<th>Lead+VSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 terms</td>
<td>3.07</td>
<td>2.30</td>
<td>1.93</td>
<td>1.76</td>
<td>3.24</td>
</tr>
<tr>
<td>5 terms</td>
<td>2.71</td>
<td>1.91</td>
<td>1.57</td>
<td>1.37</td>
<td>3.11</td>
</tr>
</tbody>
</table>

One reason for the fact that 10-term summaries get higher scores is that they contain more relevant terms than the 5-term ones. Since three of the four developers reported in the post-experiment questionnaire that they put emphasis on the relevant terms rather
than the irrelevant ones when ranking summaries, the 10-term summaries get higher scores. This observation is supported also by the high correlation we found between the number of relevant terms contained in the summaries and the scores of they received (0.82 Spearman rank correlation).

According to these results, one could say that summaries should have as many terms as possible, since the scores of the summaries increase with the number of relevant terms, and the number of relevant terms increases with the total number of terms in the summary. However, the purpose of the summaries is to offer developers a short description of the software artifact, which would take less to read than the original artifact.

4) Splitting techniques
When analyzing the results for individual summaries, we observed that the versions containing only full identifiers obtained the highest score in 60% of the cases, followed by the summaries containing both the terms resulted after splitting the identifiers and the original identifiers, in 40% of the cases. The summaries containing only the terms by splitting the identifiers never had the highest score. Apart from that, in 80% of the cases they were assigned the lowest scores.

We found the same tendency for the lead+VSM summaries, where the version of the summaries having the full identifiers always received the highest scores (on average the score was 3.33), followed in all cases by the version containing both the full identifiers and the words resulted after splitting the identifiers (on average they got a score of 3.19). The summaries containing only the terms after splitting the identifiers were always the ones developers assigned the lowest score to, i.e., 3.01 on average.

These findings were confirmed also by the developers, which reported that they preferred the summaries containing the full identifiers. All developers reported that the summaries containing only the terms resulted after splitting the identifiers were last in their preferences. They explained their choice by the fact that it is harder to understand what the method or class is doing when the phrases appearing in identifiers are split, especially when these are phrases containing a verb and its direct object.

5) Weights
For both TR techniques we used three of the most popular weighting schemes used in text summarization, namely binary-entropy, tf-idf, and log (Salton and Buckley, 1988). In order to determine which weight is most appropriate for source code summarization, we computed the averages for each weighting scheme for methods, classes and over all the artifacts. The aggregated results for all summarization techniques using a particular weight are presented in Table 6-5. The weighting scheme with the lowest scores for VSM across all data is td-idf. This weight obtained the lowest score for both 5-term summaries and 10-terms summaries, in the case of methods, classes, and overall. Binary-entropy was the weight which resulted in the highest ratings for VSM over the entire data, and for methods as well. However, in the case of classes, it was surpassed by log.

For LSI, the situation was the opposite. Tf-idf obtained the highest scores among the weights for classes and over all artifacts. However, for methods, log performed better. One curious thing was that, even though tf-idf received the lowest scores for VSM summaries, the lead+VSM summaries got the highest scores when this weight was used. Sometimes, however, binary-entropy or log were on the same level as tf-idf.

Table 6-5: Average scores for different weights

<table>
<thead>
<tr>
<th>WEIGHTING SCHEME</th>
<th>VSM</th>
<th>LSI</th>
<th>LEAD+VSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tf-idf</td>
<td>2.07</td>
<td>1.81</td>
<td>3.19</td>
</tr>
<tr>
<td>Log</td>
<td>2.12</td>
<td>1.78</td>
<td>3.16</td>
</tr>
<tr>
<td>Binary-entropy</td>
<td>2.13</td>
<td>1.67</td>
<td>3.18</td>
</tr>
</tbody>
</table>

Even though there are slight differences between the performances of TR summarization techniques using different weighting schemes, this difference proved to be not statistically significant. It is hard to draw any clear conclusion on this issue at this point. Future studies involving larger and more varied corpora are most likely needed in order to reach a conclusion.

6) Agreement between developers

Subjectivity is unavoidable, especially in online evaluations. It is desirable to estimate its effect on the results and it is common to measure the agreement of the evaluators in order to understand the results better. Light variations in rating between developers are normal and expectable, for example one developer agreeing highly with a summary (score 4) and a second one still agreeing, but less intensely (score 3). However, situations
when a developer agrees or highly agrees with a summary, and another one disagrees, or highly disagrees with it can indicate misunderstandings of the class or method. In consequence, we focused on finding such cases and analyzing their causes.

The agreement measure we used for a pair of developers is defined as the number of times the two developers either both agreed with the summary to some degree (they assigned it a score of 3 or 4) or they both disagreed with it (they assigned it a score of 1 or 2). Note that if a developer gives a summary a score of 1 and a second developer gives the summary a score of 2, we consider the two developers to be in agreement, as they both disagreed, even though to a different degree. We found that the pairs of developers agreed on average in 76.8% of the cases. This is similar to the level of agreement found in studies involving the assessment of natural language summaries (Mani, 2001). The agreement for each type of summary and number of terms is shown in Table 6-6.

For all summaries not involving the lead summaries, the 5-term summaries had a higher agreement than the 10-term summaries. Among these, the highest agreement was in the case of the random summaries, which had a 92.8% agreement for the 5-term summaries. These observations are explained by the fact that in the case of VSM, LSI, and random summaries, the 5-term summaries were often of poor quality (see Table 6-4), containing few relevant terms, thus making it easy for developers to assign them a low score and to agree on this. One could also say that since the 5-term summaries contain less terms, there is less to disagree about.

Table 6-6: Average agreement between all pairs of developers

<table>
<thead>
<tr>
<th>Technique</th>
<th>5 terms</th>
<th>10 terms</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lead</td>
<td>63.8%</td>
<td>73.6%</td>
<td>68.7%</td>
</tr>
<tr>
<td>VSM</td>
<td>76.2%</td>
<td>61.8%</td>
<td>69.0%</td>
</tr>
<tr>
<td>LSI</td>
<td>86.6%</td>
<td>75.2%</td>
<td>80.9%</td>
</tr>
<tr>
<td>Random</td>
<td>92.8%</td>
<td>80.4%</td>
<td>86.6%</td>
</tr>
<tr>
<td>Lead+VSM</td>
<td>70.2%</td>
<td>81.6%</td>
<td>75.9%</td>
</tr>
</tbody>
</table>

The lead and lead+VSM summaries, on the other hand, had a higher agreement for the 10-term summaries. In this case the situation is exactly the opposite: the 10-term summaries were often very good, especially in the case of methods. It was easier for
developers to agree on assigning those good scores, as opposed to the lead 5-terms summaries, which contained fewer relevant terms.

Overall, however, for the individual summaries developers agreed more on rating the bad summaries than the good ones. The random summaries got the highest agreement, followed by the LSI summaries, then VSM, and lead as last. This is the reverse order of the types of summaries according to their score. In fact, we found a strong negative correlation (-0.75) between the scores received by a summary and the agreement of developers in scoring it.

7) Relevant terms

Table 6-7 presents the number and ratio of relevant terms (number of relevant terms/total number of terms in a summary) in the 5-term and 10-term summaries for the summaries with the highest scores, i.e., lead, VSM, and lead+VSM. As we can observe, the lead+VSM summaries contain always a higher number of relevant terms than the lead or the VSM summaries alone, even though the ratio of relevant terms decreases compared to the highest ratio in the two individual summaries. However, the lead+VSM summaries always received a higher rating from developers than any other summarization technique alone. This indicates that the number of relevant terms in a summary is more important than the ratio of relevant/irrelevant terms, i.e., relevant terms bare more weight than the irrelevant ones in rating a summary. This was confirmed by three of the developers, i.e., D2, D3, and D4 during the post-experiment questionnaire. D1 was the only developer which considered the number of irrelevant terms as an important factor in his ratings.

Table 6-7: The number and the ratio of relevant terms in Lead, VSM and Lead+VSM summaries

<table>
<thead>
<tr>
<th></th>
<th>Rel. terms</th>
<th>Methods</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Lead</td>
<td>VSM</td>
<td>Lead+VSM</td>
<td>Lead</td>
<td>VSM</td>
<td>Lead+VSM</td>
<td>Lead</td>
<td>VSM</td>
<td>Lead+VSM</td>
<td></td>
</tr>
<tr>
<td>5T</td>
<td>No.</td>
<td>4.70</td>
<td>4.07</td>
<td>7.58</td>
<td>3.82</td>
<td>4.02</td>
<td>7.14</td>
<td>4.19</td>
<td>4.04</td>
<td>7.36</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ratio</td>
<td>0.91</td>
<td>0.81</td>
<td>0.86</td>
<td>0.76</td>
<td>0.80</td>
<td>0.78</td>
<td>0.84</td>
<td>0.81</td>
<td>0.82</td>
<td></td>
</tr>
<tr>
<td>10T</td>
<td>No.</td>
<td>7.95</td>
<td>7.29</td>
<td>12.40</td>
<td>7.18</td>
<td>7.05</td>
<td>12.32</td>
<td>7.57</td>
<td>7.17</td>
<td>12.36</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ratio</td>
<td>0.80</td>
<td>0.73</td>
<td>0.77</td>
<td>0.72</td>
<td>0.71</td>
<td>0.70</td>
<td>0.76</td>
<td>0.72</td>
<td>0.74</td>
<td></td>
</tr>
</tbody>
</table>

From the same table we can notice that for methods the lead summaries always have a higher number and ratio of relevant terms than the VSM summaries. For classes,
however, the lead summaries have a lower number of relevant terms for the 5-term summaries, and a comparable number of terms for the 10-terms summaries. This is because often few of the first 5 or 10 terms in a class are relevant terms for the class. For methods, however, the lead summaries often contain the header of the method, among other things, which contains terms considered relevant by the developers. Even though in the case of classes the VSM summaries have more or almost the same number of relevant terms as the lead summaries, the lead always received a higher rating. This indicates that not only the number of relevant terms is important, but also which these terms are. For example, the lead summaries contain often the names of classes, considered very important by all developers, which tend to give them high ratings.

**Threats to validity**

As with any case study, generalization of the results has to be done with care. Several factors influence our ability to produce analytical generalizations.

Due to the high time demand, we were not able to involve more developers at this stage. With other developers providing the ratings the results may be somewhat different. We plan to perform studies involving significantly more developers in our future work.

The developers did not agree in some cases about the ratings of the summaries. However, the level of disagreement was not higher than observed during the assessment of natural language text summaries (Mani, 2001). We expect that similar agreement levels would be observed among any set of developers.

During the evaluation sessions, developers had to rate several summaries of the same artifacts. In consequence, the presence of a learning effect is possible. We did not try to measure it or mitigate it.

We selected only ten methods and ten classes from each of the systems. While we tried to vary their properties, they may not necessarily be the most representative of the two systems and even less so of other systems. More than that, the two systems have high quality, self-explanatory identifiers. It is hard for us to estimate what the results would be for systems with poor identifier naming.
6.2.3 Conclusions

In this work we investigated the use of automatic text summarization techniques to generate source code summaries. We found that a combination between techniques making use of the position of terms in software and TR techniques capture the meaning of methods and classes better than any other of the studied approaches. Developers generally agree with the summaries produced using this combination.

When analyzing the parameters that impact the production of source code summaries, we found that the weights used for the TR techniques do not impact the produced summaries considerably. On the other hand, longer summaries seem to be preferred by developers over short ones. However, the summaries have to remain short, as a developer should be able to read them faster than the source code. Developers also preferred the summaries containing the full identifiers, as opposed to the ones where the identifiers are split.

6.3 Sentence-based Summaries of classes

6.3.1 Introduction

We propose in premiere a technique to automatically generate structured natural language descriptions for Java classes, independent of their context and assuming that no documentation exists (i.e., if it exists, the comments are currently not used). The system takes a Java project as input, and for each class, it outputs a natural-language summary. The goal of the generated summaries is to support the quick understanding of a class by describing its intent and leaving aside its context and any algorithmic details. In this sense, the summaries are indicative (i.e., provide a brief description of the class content), abstractive (i.e., include information that is not explicit in the class), and generic (i.e., attempt to cover only the important information of the class).

The intended audience is any developer who is unfamiliar with the code and needs to quickly get the gist of the class to decide whether to peruse the source code or not. For example, the developer may be deciding whether to (re)use a class X and wondering whether it would serve her needs; or, while reading the code of another class, she encounters an attribute of type X and wonders what it means. Developers sometimes write comments that describe the main responsibility of a class, to help other developers,
regardless of their task. Our automatic summaries have the same goal. Although different maintenance tasks require different kinds of information from classes, our approach can serve as an initial step in the generation of specific-purpose summaries, which is outside the scope of this paper.

Our conjecture is that the type of methods and their distribution in a class are not accidental and denote some design intent, which reflect the main goal of the class. Thus, our summarization technique first determines the stereotypes of the class (Dragan, et al., 2010) and each one of its methods (Dragan, et al., 2006). The stereotype information is used in conjunction with predefined heuristics, to select the information that will be included in the summary. The summary of the class is then generated by combining the selected information following a set of predefined rules, and using techniques developed to generate natural language phrases for variables and program statements (Sridhara, et al., 2010). The technique is completely automated and very fast.

As mentioned, the summarization technique assumes that no comments are present (i.e., a worse-case scenario). The generated summaries include identifiers associated with data attributes and methods. Since no domain information is used (e.g., domain vocabularies or ontologies), the quality of the summaries depends on the quality of the identifiers. The summaries focus on class responsibilities and ignore class relationships. Clearly, to fully understand a class, its context should not be ignored. Future work will investigate augmenting the summaries with class relationship information.

This section presents the details of our technique and its implementation, concluding with an empirical evaluation where we asked 22 programmers to evaluate three aspects of summaries generated for 40 Java classes from two systems. They found that the summaries are expressive (i.e., readable and understandable) and concise (i.e., do not contain extraneous information), and that their content is informative enough (i.e., important information is not missing) in most cases.
6.3.2 Summary generation process

The main steps in the summary generation are: *stereotype identification*, *content selection*, and *text generation*. The remainder of this subsection describes in detail these main steps.

**Stereotype identification**

Class stereotypes are high level abstractions that describe the role or responsibility of classes in a system (Dragan, et al., 2010). We decided to extract and use class stereotype information to reflect the generic responsibility of a class. For example, an Entity class encapsulates data and behavior, and it is usually the keeper of the data model and business logic. This is domain independent information, which can be automatically extracted from the source code. The domain specific information is given by the identifiers of methods and fields. Note that we do not employ any domain analysis (e.g., word relations) or domain artifacts (e.g., domain vocabulary or ontology) to determine the quality of the identifiers, hence the quality of the summaries is implicitly dependent on the identifiers we include.

We identify the stereotype of a class by adapting the rules proposed in (Dragan, et al., 2010) (Dragan, et al., 2006). These rules consider the distribution of the methods and their stereotypes in the class.

1) **Method stereotypes.** Method stereotypes describe the responsibility of methods within a class. For example, a method that returns an attribute directly is commonly known as a *get* method, whereas a method that modifies one attribute is called a *set* method. Recent work (Dragan, et al., 2006) (Dragan, et al., 2009) defined 15 stereotypes for C++ methods.

Method stereotypes are classified as: *accessors*, if the method returns information about the object’s state directly, or through the parameters of the method; *mutators*, when the method changes the object’s state; *creational* methods, if the method creates or destroys objects; *collaborational* methods, when the method defines the communication between objects or how the objects are controlled in the system; and *degenerate* methods, in any other case. Each category includes several specific stereotypes. To determine the
method stereotypes we used \textit{JStereoCode}, an Eclipse plug-in for automatically identifying Java code stereotypes (Moreno and Marcus, 2012).

2) \textbf{Class stereotypes}. Previous work (Dragan, et al., 2010) defined a list of 13 class stereotypes (Table 6-8). The stereotypes are stated as rules based on the distribution of the class’s method stereotypes. For example, to determine whether a class is \textit{Boundary}, it has to contain more \textit{collaborators} than non-collaborator methods, some \textit{factory} methods, and a low number of \textit{controller} methods. In quantitative terms, these conditions are translated to:

\begin{align*}
|\text{collaborators}| & > |\text{methods}| - |\text{collaborators}| \quad \text{and} \\
|\text{factory}| & < 1/2 |\text{methods}| \quad \text{and} \\
|\text{controller}| & < 1/3 |\text{methods}|
\end{align*}

where $|\text{methods}|$ is the total number of methods, and $|\text{factory}|$, $|\text{controller}|$, and $|\text{collaborators}|$ represent the number of factory, controller, and collaborator methods in the class, respectively.

The rules defined in (Dragan, et al., 2010) were modified and extended in order to eliminate the overlap between some of the rules, which allowed classes to be classified as having too many stereotypes (Moreno and Marcus, 2012). The resulting class stereotypes are presented in Table 6-8.
Table 6-8: Class stereotype taxonomy (Adapted from (Dragan, et al., 2010))

<table>
<thead>
<tr>
<th>#</th>
<th>Class Stereotype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Entity</td>
<td>Encapsulates data and behavior. Keeper of data model and business logic.</td>
</tr>
<tr>
<td>2</td>
<td>Minimal Entity</td>
<td>Trivial Entity that consists entirely of accessor and mutator methods.</td>
</tr>
<tr>
<td>3</td>
<td>Data Provider</td>
<td>Entity that consists mostly of accessor methods.</td>
</tr>
<tr>
<td>4</td>
<td>Commander</td>
<td>Entity that consists mostly of mutator methods.</td>
</tr>
<tr>
<td>5</td>
<td>Boundary</td>
<td>Communicator that has a large percentage of collaboration methods, a low percentage of controller, and not many factory methods.</td>
</tr>
<tr>
<td>6</td>
<td>Factory</td>
<td>Consists mostly of factory methods.</td>
</tr>
<tr>
<td>7</td>
<td>Controller</td>
<td>Controls external objects - the majority of its methods are controllers and factories.</td>
</tr>
<tr>
<td>8</td>
<td>Pure Controller</td>
<td>Consists entirely of controller and factory methods.</td>
</tr>
<tr>
<td>9</td>
<td>Large Class</td>
<td>Combines multiple roles, i.e., it consists of accessor, mutators, collaborational, and factory methods.</td>
</tr>
<tr>
<td>10</td>
<td>Lazy class</td>
<td>Its functionality cannot be easily determined. It consists mostly of incidental, and get or set methods.</td>
</tr>
<tr>
<td>11</td>
<td>Degenerate</td>
<td>Degenerate. Very trivial class that does very little - it consists mostly of empty, and get or set methods.</td>
</tr>
<tr>
<td>12</td>
<td>Data Class</td>
<td>Degenerate behavior - it has only get and set methods.</td>
</tr>
<tr>
<td>13</td>
<td>Pool</td>
<td>Consists mostly of class constants and a few or no methods</td>
</tr>
<tr>
<td>14</td>
<td>Boundary+Data Provider</td>
<td>A boundary class whose main purpose is to provide access to its attributes.</td>
</tr>
<tr>
<td>15</td>
<td>Boundary+Commander</td>
<td>A boundary class whose main purpose is to allow modification of its attributes.</td>
</tr>
<tr>
<td>16</td>
<td>no-stereotype classes</td>
<td>classes that do not match the conditions of any stereotype</td>
</tr>
</tbody>
</table>

Content selection

Once the class stereotypes are determined, we need to identify which methods are to be included in the summary. These methods are determined through a filtering process, which starts with a set containing all the methods in the target class, except the ones overriding Object’s methods. Two filters are applied to this set:

1) **Stereotype-based filter.** The first filter removes the methods whose stereotypes are not relevant to the class stereotype according to its definition. In consequence, we apply one heuristic for each class stereotype. For example, *accessors* are the main methods in a *Data Provider* class, so every non-accessor method (i.e., every method that is not *accessor*) is filtered out when processing this kind of classes.

For classes with two stereotypes we combine the two heuristics such that we remove the methods that are not relevant to any (not both) of the definitions. As an illustration, every method that is not *accessor* or *collaborator* is removed when processing *Boundary+Data Provider* classes (Dragan, et al., 2006).
2) **Access-level filter.** The second filter is based on the access level permitted by the modifiers of the methods. Java provides four levels of access to method members (relevant in our context): *private*, *package-private*, *protected*, and *public*. Since we focus primarily on the most visible (i.e., system level) responsibility of a class, we remove private, package-protected, and protected methods (one category at a time). We remove these methods in order, from the least visible to the more visible, observing the stop rules defined below.

The filtering process ends when one of the following situations occurs: (i) the set has three or less methods; (ii) the set has been reduced to 50% or less than its initial size; or (iii) all the filters have been applied (occurs mostly for *Large Classes*).

**Text generation**

Once the information is selected, the summaries are constructed using techniques developed to generate natural language phrases for variables and program statements (Sridhara, et al., 2010). The summary consists of four parts:

1. A general description based on the interfaces, superclass and/or the stereotypes of the class.
2. The description of the structure based on its stereotype.
3. The description of the behavior based on enumerating its most relevant methods, grouped in blocks.
4. A list of the inner classes, if they exist

Based on Sridhara’s work for method summary comment generation (Sridhara, et al., 2010), we defined text generation templates for each of the four parts of class summaries.

1) **First part: General description.** The first sentence of the summary provides a generic idea of the type of objects represented by the class. This is achieved by using the names of the super classes and interfaces of classes (if any), as qualifiers of the represented object. For example:

<table>
<thead>
<tr>
<th>Class declaration:</th>
<th>public class AudioFile extends File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generated text:</td>
<td>A file extension for audio files.</td>
</tr>
</tbody>
</table>
If both a superclass and interfaces exist, then they are included in the description using the template:

\[ \text{A } <\text{interface}_1>,...,<\text{interface}_i> \text{ implementation, and } <\text{superclass}> \text{ extension for } <\text{represented object}>. \]

When classes do not extend or implement any type, the stereotype of the class is used as the qualifier. For example:

**Class declaration:** public class CdRipper  
**Stereotype:** Boundary  
**Generated text:** A boundary class for cd rippers

In the worst case, the general description only mentions the object represented by the class. The text generation component also considers whether the target class is an abstract declaration. For example:

**Class declaration:** abstract class Context  
**Generated text:** An abstract class for contexts

Considering the possible variations in class declarations and the class stereotype taxonomy, we defined 22 different templates to provide the general description.

2) **Second part: Stereotype description.** The generic responsibility of the class is described using the stereotype definitions, presented in Table 6-8. When it is possible, the definitions are enriched with specific information (i.e., some domain information), such as, the represented object, the classes used within the target class, or the existence of certain kind of methods. For example, when the class to summarize is a Data Provider, the generated text follows the template:

This entity class consists mostly of accessors to the <represented object>‘s state.

As mentioned before, the focus of the summaries is not on class relationships. The only (partial) exception is for **Boundary** classes, where it is important to mention (at least some of) the classes it communicates with. These classes can be selected in several ways, but
we decided to include the most frequently used ones within the class. For example, the stereotype description for a *Boundary+Commander* class is:

This boundary class communicates with <class\(_1\)>., ..., <class\(_k\)>., and other <remaining number of classes used> classes, and consists mostly of mutators to the <represented object>'s state.

Some of the class stereotypes do not depend on the presence of all the method stereotypes (see the description presented in Table 6-8). For example, a *Data Class* can consist entirely of get methods. In such a case, the fragment that refers to the existence of set methods is omitted from the description:

This data class consists only of getter methods.

When generating this part of the summary, we must consider whether the first part of the summary already contains the stereotype name or the word “class”. If this is the case then, the text generation component uses alternative starting words in order to reduce redundancy and gain readability. For example:

| Class declaration: Public class CdRipper | Stereotype: Entity | Generated text: It includes accessors and mutators to the cd ripper's state, and some business logic |

Taking into account the class stereotype taxonomy, the stereotypes of the methods in the class, and the content of the first part of the summary, we defined 40 different templates for the second part of the summaries.

3) **Third part:** *Behavior description.* For describing the behavior provided by the class, we use the relevant methods selected when applying the filtering process already described. Note that we could simply enumerate all of those methods. However, this would reduce the readability of the summary, especially in cases when a class has methods of many different stereotypes.
Consequently, we divide the behavior description into three blocks. The first two blocks represent the accessor and mutator methods, respectively, while the third block represents the rest of methods. Then, to generate the behavior description, each relevant method is assigned to a block according to its stereotype. If one of the blocks is empty, it is ignored when generating the description. In general, the resulting text follows the template:

It provides access to:
- <phrase for accessor method_1>;
- …; and
- <phrase for accessor method_k>.

It allows managing:
- <phrase for mutator method_1>;
- …; and
- <phrase for mutator method_j>.

It also allows:
- <phrase for method_1>;
- …; and
- <phrase for method_z>.

To create a readable description of the behavior, we need to express the methods’ action in natural-language form. We assume that the method signature provides enough information to describe the general functionality of a method in a readable form. This part of the summary is especially sensitive to the identifier quality. We used existing phrase generation tools (Sridhara, et al., 2010) to obtain natural-language fragments from method signatures.

4) Fourth part: Inner classes enumeration. The last part of the summary is optional. It is only used when the class declares inner classes. In such cases, the following template is used:

It declares the helper classes <inner class_1>, …, and <inner class_n>.

The final summary is created by concatenating the four parts described above. Figure 6-1 shows a complete summary for the class MPlayerHandler in aTunes, which consists of six fields and 23 methods. Nine of these methods are classified as mutators and one as
accessor. The rest of them are spread in the creational, collaborational, and degenerate categories. According to the stereotype identification rules, this class is a Commander.

Figure 6-1: automatic summary for the class MPlayerHandler and a fragment of its declaration

An AbstractPlayer extension for m player handlers.
This entity class consists mostly of mutators to the m player handler's state.
It allows managing:
- mute;
- volume; and
- next with no gap.
It also allows:
- finishing m player handler;
- handling next;
- playing audio file f;
- stopping m player handler;
- playing m player handler; and
- handling previous.

```java
public class MPlayerHandler extends AbstractPlayer {
    public static final boolean GAP = false;
    private static final String LINUX_COMMAND = "mplayer";
    private static final String WIN_COMMAND = "win_tools/mplayer.exe";
    private static final String QUIET = "-quiet";
    private static final String SLAVE = "-slave";
    private Process process;

    @stereotype CONSTRUCTOR
    public MPlayerHandler() {
    }

    @stereotype COLLABORATOR
    private static boolean testMPlayerAvailability() {
    }

    @stereotype SET
    private void play(AudioFile f) throws IOException {
    }

    @stereotype COMMAND
    public void finish() {
    }
```
6.3.3 Evaluation of automatic class summaries

The goal of the automatic summary of a class is to provide developers with a quick overview of its main responsibility, which can be easily read. Accordingly, we performed a study involving potential users, in a manner similar to previous work (Sridhara, et al., 2010), in order to evaluate the following properties of the generated summaries:

- Informativeness (Content adequacy): Is all important information about the class reflected in the summary?
- Conciseness: Is there extraneous information included in the summary?
- Expressiveness: How readable and understandable is the summary?

For this study we asked 22 programmers to judge the informativeness, conciseness, and expressiveness of automatically generated summaries for 40 Java classes.

Subjects and objects of the study

The study included 22 graduate students in computer science: 11 from the University of Delaware, 5 from Wayne State University, and 6 from Universidad Nacional de Colombia. We surveyed their programming knowledge and background. All of them reported good or very good knowledge of Java programming and some of them have industrial experience as Java developers.

We used two open source Java systems in the study, namely aTunes 1.6.0 and ArgoUML 0.22. aTunes is an audio player, consisting of 218 classes and 1,852 methods. ArgoUML is an UML modeling tool with 1,548 classes and 10,341 methods.

Since stereotypes are the foundation of our summarization technique, for both systems we selected one class representing each stereotype and two classes for the most frequent stereotypes in these two systems. The five most frequent stereotypes are: Entity, Boundary, Boundary+Data Provider, Boundary+Commander, and Controller. Moreover, we omitted Pure Controller classes from the study, since they are almost nonexistent in both systems. In total, 40 classes were selected. We selected 20 classes per system, by grouping the classes according to their stereotype, and randomly selecting two for the five most frequent stereotypes and one for each of the other 10. While we wanted to ensure
stereotype coverage, we also wanted to make sure each class summary is evaluated by at least three participants.

**Experimental design and procedure**

An expert with knowledge of a particular Java class (ideally its main developer) would give the most trustworthy evaluation of a summary of that class. Since we did not have access to the developers of the object systems, we ensured that the participants understood the summarized classes as much as possible. Therefore, we asked participants to familiarize themselves with the major functionality and structure of both systems by reading brief descriptions and, if needed, by executing the systems. After that, they were asked to (i) study and understand the entire class, (ii) write their own description for the class, (iii) read the provided automatic summary, and finally, (iv) evaluate the summary by answering the three questions presented in Table 6-9. We asked the participants to write their own summary (with no predefined format or content requirement) in order for us to assess later whether they really understood the class or not. Our assumption is that if a participant did not understand a class properly, their judgment of the summary may be useless. When evaluating the summaries for informativeness and conciseness, if they selected the third choice for informativeness or the second and third for conciseness (Table 6-9), then they were asked to write in a free form what is missing or what is extraneous in the summary.

**Table 6-9: Questions and possible answers used in the evaluation study**

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Question</th>
<th>Possible answers</th>
</tr>
</thead>
</table>
| Informativeness | Considering only the content of the description and not the way it is presented, do you think that the description? | 1. Is not missing any information  
2. Is missing some information but the missing information is not necessary to understand the class  
3. Is missing some very important information that can hinder the understanding of the class |
| Conciseness  | Considering only the content of the description and not the way it is presented, do you think that the description? | 1. Has no unnecessary information  
2. Has some unnecessary information  
3. Has a lot of unnecessary information |
| Expressiveness | Considering only the way the description is presented and not its content, do you think that the description? | 1. Is easy to read and understand  
2. Is somewhat readable and understandable  
3. Is hard to read and understand |
The assignment of the 40 selected classes and their summaries to participants was designed in such a way that:

- Each summary is reviewed by at least three participants;
- Each participant evaluates and equal number of classes from aTunes and from ArgoUML, and
- None of the participants receives summaries of classes with the same stereotype from the same system.

Thus, each participant was provided with six classes, three per system, and their corresponding automatic summaries. In order to achieve our goals, 20 subjects would have been enough (three judgments per summary and six summaries per participant). The distribution of classes was done before inviting the participants. We invited 25 participants (who were contacted individually) assuming that some will not be able to complete the study. The extra five evaluations were replicating five of the first 20 evaluations. Three invited participants did not complete the evaluation, so we collected data from 22. Hence, two of the summaries got only two judgments, 12 summaries received four judgments, one summary got five judgments, and the rest (25) of the summaries got three judgments. The subjects judged the classes in different order; and both systems were intermingled (i.e., some subject started with aTunes and some with ArgoUML).

We used a web page and an online survey tool to perform the study. This allowed us to provide evaluators with the necessary resources and instructions, controlling the steps that they had to perform, and collecting their answers. Additionally, they used an IDE to import the source code of both systems and study the Java classes assigned to them. The participants carried out the evaluation independently, i.e., at the time and space of their choice, without our supervision, and using the IDE of their choice. They could also stop in the middle of the evaluation and resume at any later time. We recorded the time they spent on studying each class to ensure they did spend enough time on each class and it did not take them too much time to complete the evaluation. We estimated that it would take on average between 90-120 minutes to complete the evaluation (15-20 minutes per class). On average, the participants accomplished this part of their task in 10 minutes, for either system. The two classes that took longest were one Commander and
one Boundary class from ArgoUML, yet still within our expectations (20 minutes on average). One of those classes represented a functionality not fully implemented in this version of the system (org.argouml.cognitive.ToDoList). In the case of simple classes such as Minimal Entities, the average was below five minutes. We concluded that the participants did not get too tired through the study and they spent an adequate amount of time understanding the code.

Results and discussion
In total, we collected 132 evaluations of the summaries of 40 classes. Before analyzing the results, two of the authors assessed separately each summary written by the participants. The authors spent two days studying and understanding the 40 classes. Even before the study, they were quite familiar with both systems. Each participant summary was assessed on a two-level scale: it was considered good when it captured the intent of the class, or bad, otherwise. Conciseness or clarity of the summaries was not considered at this stage. Clearly these evaluations are subjective; hence we ensured that the two authors agreed on their evaluations. The assessments given by the authors were compared to confirm the high or low quality of the descriptions. In most cases the authors agreed on the results. Where some disagreement existed, it was carefully discussed, until they reached an agreement.

In the end, based on this analysis, 24 out of the 132 participant descriptions indicated a poor understanding of the class. One of the classes was particularly difficult to understand by the participants (i.e., all of its evaluator descriptions were misleading or incorrect). This is a Factory class from ArgoUML (i.e., org.argouml.ui.explorer.rules.GoTransitionToGuard). The rest of the bad descriptions were spread uniformly across all the classes, stereotypes, and systems. They did not point at any specific participant, stereotype, or system. While analyzing the results, we duplicated each analysis with and without the responses corresponding to the 24 bad participant summaries. We expected that the participants who misunderstand a class would be prone to choose the first answer in each category, in order to avoid justifying their choice. However, these 24 cases did not significantly affect the results (see Table 6-10 for details).

Table 6-10 reports the percentage of automatic descriptions that were rated in each category for informativeness, conciseness, and expressiveness. The numbers in
parenthesis reflect the results after the removal of the evaluations corresponding to the 24 bad user summaries. Overall, the results are promising as for each property of the automated summaries the largest set of answers were the most positive choice. According to these numbers we can say that the automatic descriptions are mostly concise and readable, have little unnecessary information, but sometimes miss important information. Note that those cases (the 24 ones) where human summaries indicated deficient understanding of the classes do not influence much the overall results of the study.

Table 6-10: Distribution of participant responses
(The numbers in parenthesis reflect the answers after removing the 24 corresponding to the bad user summaries)

<table>
<thead>
<tr>
<th>Informativeness</th>
<th>Conciseness</th>
<th>Expressiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Response category</strong></td>
<td><strong>Percentage of Ratings</strong></td>
<td><strong>Response category</strong></td>
</tr>
<tr>
<td>Not missing any information</td>
<td>43% (45%)</td>
<td>Has no unnecessary information</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Is easy to read and understand</td>
</tr>
<tr>
<td>Missing some information</td>
<td>26% (26%)</td>
<td>Has some unnecessary information</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Is somewhat readable and understandable</td>
</tr>
<tr>
<td>Missing some very important</td>
<td>31% (29%)</td>
<td>Has a lot of unnecessary information</td>
</tr>
<tr>
<td>information</td>
<td></td>
<td>Is hard to read and understand</td>
</tr>
</tbody>
</table>

We also looked at the differences in the ratings given by evaluators between the two software systems. The two systems are quite different from each other – chosen on purpose. aTunes is smaller, its domain is much simpler (e.g., there are classes like Artist, Album, Song, AudioFile, etc.), and most of its classes have very descriptive identifiers. The results indicate that more of the summaries from ArgoUML miss important information than in aTunes (38% vs. 24%). On the other hand, 3% of the ArgoUML summaries are not concise vs. 5% of the aTunes summaries. Finally, 6% of the ArgoUML summaries are hard to read and understand vs. 2% of the aTunes summaries. These numbers include the answers corresponding to the 24 bad user summaries, also.

We present a more detailed analysis of the results for each of the three evaluated summary properties. The qualitative analysis is done using the free form text the participants provided when answering with the second or third choice, while evaluating the informativeness and the conciseness.
**Informativeness.** We consider this property as the most important, and at the same time, the hardest to ensure automatically. While it is the characteristic with lowest results, in 69% of the cases the evaluators considered that the automatic summaries miss little or no information relevant to the class. We analyzed the answer for each class stereotype to see whether some stereotypes prove to be more difficult to summarize than others. Table 6-11 summarizes these results and we discuss them in more detail.

Table 6-11: Distribution of participant responses per each stereotype
(Based on all 132 answers)

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Informativeness</th>
<th>Conciseness</th>
<th>Expressiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No info missed</td>
<td>Some info missed</td>
<td>Important info missed</td>
</tr>
<tr>
<td>Degenerate</td>
<td>17%</td>
<td>33%</td>
<td>50%</td>
</tr>
<tr>
<td>Minimal Entity</td>
<td>83%</td>
<td>17%</td>
<td>0%</td>
</tr>
<tr>
<td>Data Provider</td>
<td>43%</td>
<td>14%</td>
<td>43%</td>
</tr>
<tr>
<td>Commander</td>
<td>67%</td>
<td>33%</td>
<td>0%</td>
</tr>
<tr>
<td>Boundary</td>
<td>64%</td>
<td>18%</td>
<td>18%</td>
</tr>
<tr>
<td>Boundary+Data Provider</td>
<td>31%</td>
<td>38%</td>
<td>31%</td>
</tr>
<tr>
<td>Boundary+Commander</td>
<td>46%</td>
<td>15%</td>
<td>38%</td>
</tr>
<tr>
<td>Factory</td>
<td>14%</td>
<td>43%</td>
<td>43%</td>
</tr>
<tr>
<td>Controller</td>
<td>21%</td>
<td>50%</td>
<td>29%</td>
</tr>
<tr>
<td>Large Class</td>
<td>57%</td>
<td>43%</td>
<td>0%</td>
</tr>
<tr>
<td>Lazy Class</td>
<td>0%</td>
<td>13%</td>
<td>88%</td>
</tr>
<tr>
<td>Entity</td>
<td>64%</td>
<td>7%</td>
<td>29%</td>
</tr>
<tr>
<td>Data Class</td>
<td>83%</td>
<td>17%</td>
<td>0%</td>
</tr>
<tr>
<td>Pool</td>
<td>29%</td>
<td>29%</td>
<td>43%</td>
</tr>
<tr>
<td>No Stereotype</td>
<td>50%</td>
<td>17%</td>
<td>33%</td>
</tr>
</tbody>
</table>

The stereotypes that consistently got the best answer (i.e., the summaries include all the necessary information) were the following: **Minimal Entity, Commander, Data Class** and **Large Class** (100% of the cases), **Boundary** (82% of the cases), and **Entity** (71% of the cases). We did not expect that the summaries of **Large Classes** would obtain such good scores for informativeness. Such classes are usually complex and combine multiple roles, having methods with various different stereotypes.

We analyzed in detail the cases where the summary did not include important information. We found that 7 evaluators out of 8 considered that the descriptions for classes with **Lazy Class** stereotype missed important information. This was expected since this stereotype represents classes whose functionality cannot be easily determined (i.e., they consist mostly of *incidental* and *get* or *set* methods).
Another case where automatic descriptions miss important information corresponds to classes with Degenerate stereotype. This stereotype is assigned to classes with little functionality, i.e., those consisting mostly of empty methods, getters and setters. Factory classes proved to be difficult to describe also, especially in the case of the ArgoUML system.

In the study, we asked participants to indicate the missing information for summaries rated as low informativeness. We divided this information into three categories. The first group consists of relevant information reported as missed by several participants. Unmentioned methods and attributes fall in this category. The omitted methods are mainly caused by the filtering process: the stereotypes of the methods considered relevant by the developer are not necessarily relevant to the stereotype of the class (e.g., collaborational methods in Data Provider or Degenerate classes). This is an area where we can improve our technique, but more studies may be needed to lead us to ways to adjust our filtering process. Another cause for this situation is the loss of the method’s intent when generating text from its signature, in some cases. For example, while constructing the second block of the behavior description, the action of the mutator methods is sometimes removed. In the case of attributes reported as missing, we found that they usually correspond to attributes modified in mutator methods, when the name of the method did not reflect them. We plan to adjust our tools to include such attributes.

The second category consists of missed relevant information, reported only by one participant, yet we considered it important to analyze. We found that these reports provide important details that can help to improve stereotype descriptions. For example, in the summary of a Factory class we should include the classes with which it interacts. In the case of Pool classes, it is important to provide the data type of the constants declared by the class. The parameters of specific methods were also reported as missing information. We found that participants referred mainly to the parameters of static methods, which we currently ignore.

The missing information in the last group refers to elements that are beyond the scope of the summaries or indicate poor understanding of the evaluation task. For instance, evaluators who were unfamiliar with some concepts in the code reported their definition as missing. The same situation occurred with the usage of the classes, although we
specifically decided not to address this issue at the current stage. The generated summaries do not consider the class context and are not meant to be a dictionary of the domain concepts. Likewise, the lack of details about the inner classes (mentioned in the final part of the summary) was considered as missing information by some participants. The reason we chose not to include it is that if the developer has a particular interest for a member class, she can be referred to its corresponding summary.

Informativeness proves to be the hardest to judge by the subjects, also. When judging 14 of the 40 summaries, at least two of the subjects answered with option #1 and #3 (see Table 6-9), respectively; this indicates a significant disagreement in those cases. In contrast, when judging the conciseness and expressiveness of the summaries, such level of disagreement occurred only for two summaries, in each case. This is another indication of how difficult it is to automate the process of selecting the relevant information to be included in the summary of a class.

**Conciseness.** It is remarkable that only in five cases out of 132, the evaluators considered that the automatic descriptions had a lot of unnecessary information. These cases correspond to classes with *Boundary* (27%), *no-stereotype* (17%), and *Boundary+Data Provider* (15%) stereotype. For 11 out of 15 stereotypes of evaluated classes, the summaries have no unnecessary information, according to the evaluators.

Reviewing the cases where the evaluators said that the descriptions had some unnecessary information, we found that most of them correspond to classes in the *no-stereotype* category, i.e., the classes do not match with any set of rules defined for the stereotypes. This is certainly an expected result, since there is little we can say about such classes at this point.

In the study, we asked the developers to provide the unnecessary information existing in the summaries. We found that most of this information corresponds to the descriptions of the stereotypes. Specifically, when evaluating *Lazy* and *Controller* classes, developers mentioned that the summaries were satisfactory when ignoring such descriptions. In the case of *Boundary* classes, developers tended to consider the interaction classes as useful, except when those classes belong to Java libraries. A quick improvement for this situation is to filter out such classes. Another problem regarding this property referred to
specifying the number of communicating classes, which are not included in the summary. Evaluators suggested that this number was not useful if the classes were not mentioned. We plan to work on a better way of presenting such information, without affecting the conciseness of the summary.

Some specific methods were reported as unnecessary information. We found that these methods implement the Singleton pattern, which according to the method stereotype rules are classified as mutators. This is an issue when including them in the behavior description of a class. We consider that including this kind of method in the creational category of the taxonomy could improve the summaries.

Finally, we found that some unnecessary information was caused by the poor quality of some of the identifiers. For example, methods with identifiers similar to minutes2second are described as “getting minutes to seconds”, but a better choice would be “converting minutes to seconds.” We also found that redundant information in the natural-language form of the methods was reported as unnecessary, particularly when the name of the method and the type and name of parameters overlap. We plan to improve the heuristics used in the lexicalization process based on this feedback.

**Expressiveness.** This is the characteristic best evaluated by the participants. Similar to conciseness, only in five out of 132 cases did the evaluators consider that the automatic descriptions were difficult to read and understand. In contrast, in 88 of the cases they considered that the descriptions were easy to read and understand, and somewhat readable and understandable, in 39 cases. In our interpretation, these results reflect the fact that the text generation tools we used (developed previously by Sridhara (Sridhara, et al., 2010)) produce readable and understandable natural language text, and also, that the organization of the information is good. In many cases, all the subjects answered with option #1, i.e., all evaluations indicated that the descriptions are easy to read and understand. Such is the case for classes with Minimal Entity, Data Provider, Data Class, and Pool stereotypes.

The only case in which the results can be considered somewhat deficient is the Factory stereotype, where 29% of the summaries were deemed as hard to read and understand.
This result suggests that our heuristics need to be improved when a class consists mostly of factory methods.

### 6.3.4 Agreement among evaluators

In the evaluation study each participant chose one option among three available choices. To assess the agreement among evaluators when they answer a question with \(N\) options, (Mizzaro, 1999) proposed a procedure where the available options are sorted from the best to the worst, and then, enumerated them from 1 to \(N\). Then, he defines the function \(v(k)\) which gives the number assigned to option \(k\). Using these numbers, a *distance* between two judgments is defined as the absolute difference between the indices assigned to the selected options. Thus, if the selected options were \(p\) and \(q\), the distance between them is \(|v(p)-v(q)|\).

Based on that, we defined the *disagreement*, \(d(p,q)\), between two judgments that selected options \(p\) and \(q\), respectively, when assessing a specified property of a generated summary. The definition is:

\[
d(p,q) = \frac{|v(p) - v(q)|}{N - 1}
\]

In addition, we defined the function \(f(k)\) as the number of times that option \(k\) was selected.

Based on these definitions, and following (Mizzaro, 1999), we calculated the *overall agreement* among \(M\) evaluators that answered a particular question as the average agreement between all pairs of judgments, with the following formula:

\[
\sum_{i=1}^{N-1} \sum_{j=i+1}^{N} (1 - d(i,j)) \times f(i) \times f(j) + \sum_{i=1}^{N} \binom{f(i)}{2} \binom{M}{2}
\]

In other words, the overall agreement is calculated as the average of the agreement of each judgment with the other ones.

Table 6-12 shows the overall agreement among evaluators when they judged the three characteristics of the automatic summaries.
Table 6-12 Average agreement among evaluators

<table>
<thead>
<tr>
<th>Informativeness</th>
<th>Conciseness</th>
<th>Expressiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.61</td>
<td>0.74</td>
<td>0.82</td>
</tr>
</tbody>
</table>

The average agreement among all evaluators is 0.61, when they evaluated Informativeness (Content Adequacy); 0.74 for Conciseness; and 0.82 for Expressiveness. Informativeness was the most controversial characteristic of three considered, since the agreement was not high. Thus, it seems that judging whether a summary has the most relevant information is not easy for human evaluators. It also implies that determining what is the most relevant information for describing the purpose of a Java class is a hard question that often has more than one answer. In summary, this property is the most important, the hardest to ensure automatically and the more difficult to judge.

(Lin and Hovy, 2002) conducted an evaluation experiment of natural language summaries using data from the Document Understanding Conference 2001. Although one of the evaluations conducted by them is an online evaluation that used model summaries to score the automatic ones, the metrics used is similar to ours. Thus, it is worth to mention that the agreement within this experiment was around 0.4 and was considered as low by the researchers. This result may indicate that the inter-human agreement in our experiment is high for all the properties evaluated. That is why we consider that our approach for class summarization is promising, and also, the evaluation analysis revealed some clues for improving its performance, especially with regard to informativeness.

6.3.5 Threats to validity

Several factors influence the results of our evaluation. We only evaluated between 2-4 summaries for classes with a given stereotype. These classes were from two systems only. While we tried to maximize the number of evaluations given the number of participants we had, it is difficult to generalize the results. The participants were all graduate students with good knowledge of Java. It is conceivable that using professional developers, the results may have been different. Given the time the participants spent on understanding each class (10 minutes in average) and based on evaluating their own summaries, we concluded that (in most cases) they properly understood the classes. However, we can hardly call them experts on that code. Evaluations with the developers
of the code may lead to different results. With all such studies, future replications involving more data and more subjects would help in generalizing the results.

We believe some learning effect occurred while evaluating the summaries. For example, once the participants evaluated the first summary, they knew what would be asked for the second summary and it is possible that while comprehending the subsequent classes, they were already thinking ahead to answer the three questions. One option we considered was to ask the participants to evaluate the summaries after they understood all (or three) of the six classes. Given the rather short time of the study (to avoid fatigue), we realized that short term memory would be involved in remembering fast about the classes they studied, so we elicited their evaluation immediately, rather than later. With the same goal of limiting the learning effect, we assigned the classes to the participants in a counterbalanced way: none of them received two classes with the same stereotype, and they evaluated the classes and systems in different order. We also analyzed the answers to the first question and compared them with the rest – we observed no significant differences. We did the same with the last answer, in order to assess the effect of fatigue (if any), and we observed no significant difference.

6.3.6 Conclusions

We presented an approach to automatically generate structured natural language summaries for Java classes. The approach leverages information about the class stereotypes and uses existing text generation tools to compose summaries based on a set of heuristics we developed. While the techniques used to generate the summaries are adapted from prior work, this is a first technique that generates automatically natural language summaries for classes.

Twenty-two programmers evaluated 40 summaries generated for classes from two Java systems. According to their evaluations, 69% of the generated summaries do not miss important information about the classes, 96% of the summaries are concise, and 96% are readable and understandable. The level of agreement among them was high.

These results are more than promising, and we are convinced that our approach can be used to generate summaries that would help developers when browsing and reading the
code. The generated summaries are not targeted to any specific development task, but they could be used as a starting point for the generation of task-specific summaries. When interpreting the results, one must bear in mind that our summarization technique is completely automated and does not use existing documentation or external domain knowledge.

The study also revealed several areas where the summarization process can be improved. We plan to carry out such improvements and hope that they will result in higher quality summaries. For example, for the stereotype identification we plan to enrich both taxonomies — method and class stereotypes. This will also help in refining the heuristics to select the relevant content of a class. We also plan to consider the context of the class (i.e., the relationships to other classes) to enrich the summaries. When comments are present, they may also be used for generating the summaries. We will investigate this problem.

### 6.4 Automatic extraction of method summaries

This chapter presents an approach to automatically mine source code descriptions—in particular method descriptions—from developer communications, such as, emails and bug reports. It also presents evidence to support our assumption that developer communications are rich in useful code descriptions.

Our approach traces emails to classes, identifies affirmative textual paragraphs in these emails, and traces such paragraphs to specific methods of the classes. Then, it uses heuristics — based on textual similarity between paragraphs and methods, and on matching method parameters and other keywords to paragraphs—to filter out candidate method descriptions.

The filtering technique results in a set of one or more paragraphs describing each method (for which a description was found). These paragraphs may overlap, in terms of content, or they could describe different aspects of the method behavior, e.g., one describes the method interface and return value; another, the behavior in terms of calls to other methods; another, the exceptional behavior, etc. The technique retrieves all these
paragraphs and combines them into a method description. Such descriptions can have multiple uses:

- They can be used as such to help developers understanding the code.
- In perspective, an automatic tool can further process the descriptions and automatically generate method documentation, e.g., API descriptions or comments.

We have applied the proposed approach to 26,796 bug reports from the Eclipse project, and 18,046 emails and 3,690 bug reports from the Apache Lucene project. Results indicate that emails and bug reports contain descriptions for about 7% of the Eclipse methods and 36% of the Apache Lucene methods. The proposed filtering approach is able to correctly identify method descriptions in 79% of the cases for Eclipse and 87% of the cases for Lucene. Finally, we report several examples describing how methods are likely described in the developers’ communication, discussing the linguistic patterns we found in Eclipse and Lucene for different kinds of method descriptions.

6.4.1 Mining method descriptions from communications

The proposed approach for mining method descriptions is depicted in Figure 6-2. This section briefly explains each of the steps of the mining process.

Figure 6-2: Steps of the proposed mining approach
Step 1: Downloading emails and bug reports
First, we download mailing list archives and all bug reports concerning the analyzed time period of the investigated projects. Then, we extract the body from emails using a Perl mailbox parser (Mail::MboxParser). Bug reports (downloaded in HTML) are first rendered using a textual browser lynx and then the text is extracted using a Perl script.

Step 2: Tracing emails onto source code classes
We trace emails onto source code classes (referring to the system release before the email date). For this purpose, we use two heuristics similar to (Bacchelli, et al., 2009) (Bacchelli, et al., 2010):

- There is an explicit traceability link between a class and an email whenever (i) the email contains a fully-qualified class name or (ii) the email contains a file name (e.g., MappingCharFilter.java)—provided that there are no other files with the same name, or that the file name is also qualified with its path.
- For bug reports, we complement the above heuristic by matching the bug ID of each closed bug to the commit notes, therefore tracing the bug report to the files changed in that commit (if any are found).

Step 3: Extracting paragraphs
During a preliminary investigation we determined—by inspecting emails from out case studies—that paragraphs describing different aspects of the email topic are separated by one or more white lines. Therefore, we use such heuristics to split each email into paragraphs. For bug reports, different posts related to the same bug report are treated as separated paragraphs.

To avoid code fragments and stack traces within descriptions, we used an approach inspired to the work of (Bacchelli, et al., 2010). We computed, for each paragraph, the number and percentage of programming language keywords and operators/special characters. Paragraphs containing a percentage of keywords and special characters/operators higher than 10% are discarded.
A further processing, performed using the English Stanford Parser\(^8\) (Klein and Manning, 2003), aims at preserving only paragraphs in the affirmative form, removing those in interrogative forms — because we assume that method description should not contain interrogative sentences — as well as pruning out sequences of words that the parser was not able to analyze, i.e., sequences of words that cannot be considered valid English sentences.

Step 4: Tracing paragraphs onto methods
To trace paragraphs onto methods, we first extract signatures for all methods in a system version released before the email being analyzed. This is done using the Java reflection API. Then, we identify the paragraphs referring a method. These paragraphs shall meet the following two conditions:

- They must contain the keyword “method”. This is because we are searching sentences like “The method foo() performs...”. Indeed, there could be cases where the method is referred and described in a sentence not containing the keyword “method” (e.g., “foo() performs...”). However, we observed in a preliminary analysis that such cases occur mostly when the method is mentioned in other contexts (e.g., describing a fault) rather than when communicating a method description to other people.
- They must contain a method name, among the methods of classes traced to the email in Step 2. We also require that such a name must be followed by a open parenthesis— i.e., we match “foo()” while we do not consider “foo”. This is to avoid cases when a word matches a method name, while it is not intended to refer to the method. For example, we found several paragraphs like that e.g., “Method patch”, where “patch()” was actually a method of a class traced onto the email.

It is important to note that such a process can be subject to ambiguities. First, an email can be traced onto multiple classes, having one or more method with the same name (and maybe even the same signature). In such cases, the paragraph is assigned to all of these classes. Second, there may be overloaded methods. Where possible, this is resolved by comparing the list of parameter names mentioned in the paragraph with the

---

\(^8\) http://www-nlp.stanford.edu
list of parameters in the method signature as extracted from the source code. When this is not sufficient to resolve the ambiguity, we conservatively assign the paragraph to all matched methods. As explained later (Step 6) both ambiguities can be mitigated by computing the textual similarity between the paragraph and the method.

Step 5: Filtering the paragraphs
We defined—based on the manual inspection of hundreds of emails and the results of our previous projects—a set of heuristics to further filter the paragraphs associated with methods.

- It is required that, if a method has parameters, at least some of them are mentioned in the method description. We count the number and percentage of method parameter names mentioned in the paragraph. We define a score, \( s_1 \) as the ratio between the number of parameters mentioned and the total number of parameters in the method. We consider \( s_1 = 1 \) if the method does not have parameters.
- If a method is not void, we check whether the paragraph contains the keyword “return”. We define a score \( s_2 = 1 \) if the method is void, or if is not void and the paragraph contains “return”, zero otherwise. We argue that these paragraphs are often descriptions of how a method performs its task by invoking other methods.
- Keywords such as “overload” or “override” are likely to be contained in some paragraphs describing methods. This, in particular, happens when a paragraph describes the additional behavior with respect to the overridden/overloaded method. We define a score \( s_3 = 1 \) if any of the “overload” or “override” keywords appears in the paragraph, zero otherwise.
- When a paragraph describes a method, often it describes it in terms of invocation of other methods. Therefore, we mine the paragraphs containing for the words “call”, “execute”, “invoke” (or their plurals/conjugations). We define a score \( s_4 = 1 \) if any of the “call”, “execute”, or “invoke” keywords appears in the paragraph at least once, zero otherwise.

We apply the above described heuristics by constraining the set of selected paragraphs such that \( s_1 \geq \text{th}_p \) and \( s_2 + s_3 + s_4 \geq \text{th}_h \), where \( \text{th}_p \) is a threshold we set for the parameter
heuristic and $t_h$ is a threshold for the other heuristics. Details about the two thresholds are reported in the next section.

Step 6: Computing textual similarities between paragraphs and methods

After having filtered paragraphs using the heuristics, we rank them based on their textual similarity with the method they are likely describing. The rationale is that, other than the method name, parameter names, and other keywords identified in Step 5, such paragraphs would likely contain other words (e.g., names of invoked methods, variable names, local variables, etc.) contained in the method body. Also, as mentioned above, computing such a similarity would help mitigating ambiguities when tracing paragraphs onto methods.

To this aim, we extract the method's text from the system source code (again, referring to a version before the email). This is done using the srcml analyzer (Collard, et al., 2003). Then, we normalize the method text removing special characters, English stop words, and programming language keywords, and splitting the remaining words using the camel case heuristics. A similar text pruning is performed on paragraphs. After that, we index the paragraphs and the methods using a Vector Space Model implemented with the R\(^9\) Isa package.

We compute the textual similarity between each paragraph $P_k$ and the text of each traced method $M_i$ using the cosine similarity. For each method $M_i$, we rank its relevant paragraphs $P_k$ by the similarity measure. Finally, we consider only the paragraphs that have a similarity measure higher than a threshold $th_T$. These are the paragraphs that are presented to the user as containing a description to the method $M_i$. As it will be shown in the next section, varying $th_T$ will produce different results in terms of precisions and of retrieved candidate method descriptions.

Our proposed approach— to the best of our knowledge— represents the first attempt to mine method descriptions from developers' communication. As with any work that addresses a problem in premiere, limitations exist, which we hope to address in future

\(^9\) http://www.r-project.org
work. We highlight here those that should be kept in mind while interpreting the results of our empirical evaluation from the next section: (i) it cannot mine descriptions that span multiple paragraphs, (ii) it mines paragraphs that describe only partial or exceptional behavior of a method, and (iii) it is mostly an extractive approach, and therefore, is unable to mine abstractive descriptions, i.e., paragraphs that provide high level descriptions of methods.

6.4.2 Empirical evaluation of the approach

The goal of this study is to evaluate the proposed approach for extracting method descriptions from developers’ communications. The quality focus is the ability of the proposed approach to cover methods from the analyzed systems, as well as the precision of the proposed approach. The perspective is of researchers who want to evaluate to what extent mining developers’ communications can be used to support code understanding and to what extent the proposed approach is able to identify method summaries with a reasonable precision. The context consists of bug reports from the Eclipse project and both mailing lists and bug reports from the Lucene project. Eclipse\textsuperscript{10} is an open-source integrated development environment, written in Java. Lucene\textsuperscript{11} is a text retrieval library developed in Java. Table 6-13 reports some relevant characteristics of the two systems and the data we used. While Eclipse can be considered as a large system, Lucene is a small-medium system.

Table 6-13 Characteristics of the two subject systems

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Eclipse</th>
<th>Lucene</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyzed period</td>
<td>2001-2010</td>
<td>2001-2011</td>
</tr>
<tr>
<td>KLOC range</td>
<td>283-2486</td>
<td>6-345</td>
</tr>
<tr>
<td># of classes (range)</td>
<td>4829-18834</td>
<td>427-528</td>
</tr>
<tr>
<td># of methods (range)</td>
<td>31132-117654</td>
<td>2432-2952</td>
</tr>
<tr>
<td># of bug reports</td>
<td>26796</td>
<td>3690</td>
</tr>
<tr>
<td># of emails</td>
<td>-</td>
<td>18045</td>
</tr>
<tr>
<td># of paragraphs from bug reports</td>
<td>202539</td>
<td>115504</td>
</tr>
<tr>
<td># of paragraphs from emails</td>
<td>-</td>
<td>91408</td>
</tr>
</tbody>
</table>

\textsuperscript{10} http://www.eclipse.org

\textsuperscript{11} http://lucene.apache.org
The empirical study reported in this section aims at addressing the following research questions:

- **RQ1** How many methods from the analyzed software systems are described by the paragraphs identified by the proposed approach? While we do not expect to find descriptions for all, or nearly all of the methods, we believe that the approach would be useful in the practice only if finding descriptions for a given method would not be an extremely rare event.

- **RQ2** How precise is the proposed approach in identifying method descriptions? This research question aims at determine whether the mined description are meaningful method descriptions, or whether they are, instead, false positives. While some false positives are unavoidable, too many of them would make the approach unpractical.

- **RQ3** How many potentially good method descriptions are missed by the approach? This research question aims at providing an idea of how the proposed approach is affected by false negatives, i.e., filtering out good method descriptions.

**Threshold Calibration**
Step 5 of the proposed approach relies on two thresholds, \(th_P\) and \(th_H\). To calibrate \(th_P\), we analyze the distribution parameters referred to in the paragraphs traced onto methods. For Eclipse, the percentage of parameters had a minimum and first quartile equal to zero, a median=50%, and a third quartile and maximum equal to 100%. We selected the median as threshold and analyzed the performance with different settings for \(th_P\) varying it between 0% and 100% in 10% steps. We confirmed that the median choice works equally well for Lucene. We realize that such a rule for selecting \(th_P\) cannot be easily generalized, but it worked for these two systems and for proof of concept purpose. Investigating the generality of this rule is subject of future work.

Regarding \(th_H\), we set it to 1, i.e., accepting all cases where \(s_2+s_3+s_4 \geq 1\), in order to select paragraphs containing at least one keyword able to characterize the paragraph with respect to the different kinds of method descriptions outlined in Step 5 of the approach.
Once again, identifying alternative rules for calibration, which generalize better, is subject of future work.

The effect of the third threshold, $th_T$, on the precision of the approach is analyzed in detail in the following subsection.

**Evaluation Procedure**

First, we extracted, using Steps 1-4 of the proposed approach, a set of candidate paragraphs that are traced onto methods. We refer to them as the subset of *traced paragraphs*. After that, we performed a first pruning using the heuristics from Step 5, i.e., selecting all paragraphs—referred to as *candidate descriptions* from here on — having $th_p \geq 0.5$ and $th_H \geq 1$.

Subsequently, we computed the cosine similarities, with the aim of investigating how the performance of the approach varies by considering only paragraphs having a cosine similarity greater than a given threshold.

Then, we built the oracle against which to validate our results. The oracle was done by manually validating a sample of the *candidate descriptions*. Since it was not possible to manually validate all descriptions, we sampled 250 descriptions for each project. Such a sample allows us to achieve estimations with a confidence interval of ±5% assuming a significance level of 95% (Sheskin 2011). We decided not to perform a random sampling of the descriptions: since our aim is to analyze how the precision and the method coverage vary with different thresholds of the cosine similarity, we wanted to include in the sample enough data points representative of different cosine ranges. Therefore, the most appropriate way to proceed was to apply a stratified sampling. We divided our population of candidate descriptions in sets according to five classes of cosine range: 0%-20%, 20%-40%, 40%-60%, 60%-80%, and 80%-100%. Then, based on the distribution of descriptions over the different classes, we randomly sampled 25, 50, 100, 50, 25 descriptions for the five classes, respectively.

Then, we asked three reviewers to analyze the sampled descriptions and decide whether they were, or not, reasonable paragraph descriptions. To rate a description, the three reviewers had the system source code available and checked whether the description is,
indeed, one possible way a method could be described, either in terms of its syntax, as
extension of other methods, or in terms of a method invocation chain. If all three
reviewers agreed that a paragraph is a specific kind of description for a method, then the
paragraph was classified as true positive. If all three reviewers agreed that a paragraph is
not a good description for a method, then the paragraph was classified as false positive.
When they disagreed, they discussed until they reached consensus. In the end, 500
paragraphs were included in the oracle.

To address RQ1 we considered all the methods in the analyzed systems, whereas for
RQ2 we only used the paragraphs in the oracle, in order to analyze how the method
coverage (RQ1) and the precision (RQ2) change when increasing the cosine threshold.
We define the method coverage for a given cosine threshold $\theta_T$ as the percentage of the
methods in the system for which there exists at least one candidate description traced
onto it and such that $\cos(m_i, P_j) > \theta_T$. We define the precision for a given cosine
threshold $\theta_T$ as the percentage of true positives in the oracle for which $\cos(m_i, P_j) > \theta_T$.

Addressing RQ3 is more difficult. We are aware that precisely assessing false negatives
would be impossible (it would require analyzing the entire body of emails). Instead, we
extracted a small sample (100 paragraphs for each project, thus in total further 200
paragraphs) from the set of paragraphs pruned after applying the Step 4 heuristics, i.e.,
all paragraphs that can be mapped onto a method (and not all possible paragraphs,
because we assume that a paragraph describing a method at least mentions it), but do
not satisfy our similarity-based filtering. We manually validated the sample similarly to
how we did it for the oracle, in order to compute the percentage of false negatives in the
sample.

Results
Table 6-14 reports results about the number of paragraphs obtained after applying steps
2, 3, and 4 of the approach, i.e. (2) the initial set of paragraphs extracted from the emails
after pruning out source code and irrelevant sentences (short and interrogative ones), (3)
the number of paragraphs traced to methods, and (4) the number of paragraphs traced to
methods that satisfy the filtering according to Step 5 heuristics i.e., paragraphs with
$th_p \geq 0.5$ and $th_u \geq 1$. The table also reports, for the last two cases, the percentage of
covered methods. As it can be noticed, about 20% of the Eclipse paragraphs and 5% of the Lucene paragraphs can be traced to methods (Step 4), which ensures a coverage of 22% of the Eclipse methods and 65% of the Lucene methods. However, such paragraphs do not satisfy the heuristics of Step 5, nor they are constrained by any textual similarity threshold.

Table 6-14: Number of paragraphs and method coverage after applying filtering from steps 3, 4, and 5 of the approach

<table>
<thead>
<tr>
<th>Filtering Step</th>
<th>Eclipse</th>
<th></th>
<th>Lucene</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of paragraphs</td>
<td>Method coverage</td>
<td># of paragraphs</td>
<td>Method coverage</td>
</tr>
<tr>
<td>Step 3</td>
<td>202539</td>
<td>-</td>
<td>206912</td>
<td>-</td>
</tr>
<tr>
<td>Step 4</td>
<td>42095</td>
<td>22%</td>
<td>12417</td>
<td>65%</td>
</tr>
<tr>
<td>Step 5</td>
<td>3111</td>
<td>7%</td>
<td>3707</td>
<td>36%</td>
</tr>
</tbody>
</table>

When applying the heuristics of Step 5, the number of paragraphs is reduced to 3111 for Eclipse and 3707 for Lucene, which results in 7% method coverage for Eclipse and 36% for Lucene.

Figure 6-3 reports the achieved precision and the method coverage for both Eclipse and Lucene. The x-axis shows the increasing cosine similarity threshold, $th_T$, while the y-axis shows both the precision and the method coverage. Note that the precision is on a 0-100% scale, whereas maximum coverage for Eclipse is 22% and for Lucene is 65%.

Figure 6-3: Precision and method coverage for different levels of similarity

Note that the maximum possible coverage would be (see Step 4 of Table II) 22% for Eclipse and 65% for Lucene.
The results for both systems correspond to our expectations, i.e., increasing the cosine threshold results in an increase in precision and it comes at the cost of reduced method coverage. An interesting phenomenon is that the increase in precision peaks (79% for Eclipse and 87% for Lucene) at a threshold of approximately 0.5 for both systems, which means that maximum precision can be achieved without complete loss of method coverage. In both cases, the difference between the minimum and maximum precision is higher than the difference between the minimum and maximum method coverage (proportionally). In other words, the precision gain increases slower than the loss in method coverage. Method coverage in Eclipse for highest precision is about 3% (which means covering between 933 and 3,530 methods, depending on the version), and for Lucene is about 15% (which means covering between 365 and 442 methods).

We can summarize the results related to RQ1 (method coverage) and RQ2 (precision) stating that, on the one hand, the proposed approach is precise enough to mine method descriptions, thus reducing the developers’ burden to go through a wide number of false positives. On the other hand, the percentage of covered methods could appear as relatively low, thus it is useful to pursue a compromise between coverage and precision. However, it is important to note that (i) we cannot really expect to find descriptions for all methods, especially for large systems like Eclipse (for which, by the way, we do not have emails, but bug reports only), and (ii) the coverage depends a lot on the quality — with respect to our goal of mining descriptions — of the project discussion, which in our case seems to be better for Lucene than for Eclipse.

Regarding RQ3, i.e., the presence of false negatives, the analysis of a sample of 100 paragraphs traced to methods, but not satisfying the Step 5 heuristic, indicates that, for Eclipse, 78 out 100 paragraphs have been classified as true negatives, leaving 22 paragraphs that could represent good method descriptions, but that were discarded by our heuristics. For Lucene, 67 paragraphs were classified as true negatives, leaving a relatively large (33%) number of false negatives. Although this can be seen as a limitation of the proposed approach for capturing good method descriptions, this can be explained by the peculiar characteristics of the Lucene mailing lists and bug reports, which contain many very good method descriptions, as it has also been noticed from the high precision obtained for RQ2. As stated before, heuristics that result in a better balance between a
low number false negatives and a low number false positives will be investigated in the future. The current results are encouraging enough to motivate future research.

**Threats to Validity**

This section describes the main threats to validity that can affect the evaluation of our results. Given the kind of validation performed, it is worthwhile to mainly discuss threats to construct and external validity.

Threats to construct validity mainly concern, in this context, the measurements used in the evaluation. First, we are aware that, for assessing precision, we sampled only a subset of the extracted descriptions. However, (i) the sample size limits the estimation imprecision to ±5% for a confidence level of 95%, and (ii) to limit the subjectiveness and the bias in the evaluation, three evaluators (one not involved in the paper and one not knowing the details of the approach) manually analyzed the sample. Another threat to construct validity concerns RQ3. As explained in above, it is always difficult to perform a thorough assessment of false negatives. To deal with such a threat we evaluated a sample of 100 paragraphs not detected by the proposed heuristics. The actual number of false negatives in the entire system may be different than in the random sample.

Threats to external validity concern the generalization of results. We must remember that the main aim of this research is to investigate whether mailing lists and bug tracking systems are a useful source of information for understanding and potentially re-documenting source code, and to propose a novel approach to mine such descriptions (at proof of concept level), rather than to perform a thorough evaluation. The empirical evaluation here is limited to mailing lists/bug reports from two systems only. Clearly, it is important to point out that variables such as the project domain, the availability of mailing lists and bug reports (as well as their quality) could influence the performance of the proposed approach. Therefore, a more extensive evaluation with data sets from further systems is highly desirable. Last but not least, the generalization of the heuristics calibration cannot be guaranteed by our evaluation.
6.4.3 Qualitative analysis

This section provides a qualitative analysis of some exemplar paragraphs, identified during the manual validation. The aim is to: (i) show examples of the various kinds of descriptions that the approach is able to mine; (ii) explain why the approach, in some cases, detected false positives; and (iii), explain why the approach missed some good descriptions, i.e., false negatives.

The examples reveal several discourse patterns that characterize true positive, false positive, and false negative method descriptions. Regarding true positives, these paragraphs are always composed of sentences in affirmative form, directly explaining a method’s syntax or behavior. For Lucene (Table 6-16), the first true positive example is a clear description of the `addAttributeImpl` method from the `AttributeSource` class. In this case, the developer initially informs others about the introduction of a new method and after that he explains what the method does: “finds all interfaces that the class or superclasses implement and that extend the Attribute interface” and “adds the interface or instance mappings to the attribute map for each of the found interfaces”. This paragraph was extracted from a list of candidate descriptions with highest score (cosine=0.74), where each of these paragraph refer to 100% of the method parameters, in this case `addAttributeImpl`. We can find only phrases in affirmative form without sentences in dubitative form. In same way, for Eclipse (Table 6-15) if we observe the first true positive example—describing the constructor `ServiceLoader`—we can find a paragraph in affirmative form without sentences in dubitative form. It is important to note that this paragraph, with respect to the first example paragraph of Lucene, obtained highest rank because it refers to 100% of the parameters of `ServiceLoader` and it contains the keywords “call” and “return” and thus it describes the method in terms of invocation of other methods and in terms of its returned value (syntactic descriptions).
Table 6-15: Examples of true positive paragraphs for Eclipse

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Paragraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>ServiceLoader</td>
<td>ServiceLoader</td>
<td>Similarly to osgi services, the java serviceloader takes the name of the class for which you want a service. In the present case, we want an instance of the JavaCompiler service, so the actual call being made is: ServiceLoader.load(javax.tool.JavaCompiler) This method returns an iterator on all the services available.</td>
</tr>
<tr>
<td>Wizard</td>
<td>addPages</td>
<td>In the particular case of the NewLocationWizard, you should be able to get around it by creating a protected createMainPage method which you can override in the subclass. You can then call super.addPages() from the subclass (Wizard) add pages to avoid the duplication of the setting of the properties. I still don’t think that “alternative” is the proper term to use everywhere.</td>
</tr>
<tr>
<td>GC</td>
<td>drawString</td>
<td>The -1 value for bidiLevel is correct since it indicates that you’re not using bi-directional text. As Randy mentioned, this might be a GDI+ issue that got introduced in 3.5. Create an SWT GC, invoke setAdvanced(true), and then use its drawString() method to draw some text in your language. Also try drawText().</td>
</tr>
</tbody>
</table>

Table 6-16: Examples of true positive paragraphs for Lucene

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Paragraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>AttributeSource</td>
<td>addAttributeImpl</td>
<td>New method added to AttributeSource: addAttributeImpl(AttributeImpl). Using reflection it walks up in the class hierarchy of the passed in object and finds all interfaces that the class or superclasses implement and that extend the Attribute interface. It then adds the interface-instance mappings to the attribute map for each of the found interfaces AttributeImpl now has a default implementation of toString that uses reflection to print out the values of the attributes in a default formatting.</td>
</tr>
<tr>
<td>Scorer</td>
<td>Score</td>
<td>This proposes to expose appropriate API on Scorer such that one can create an optimized Collector based on a given Scorer’s doc-id ordenness and vice versa. QueryWeight implements Weight, while score(reader) calls score(reader, false /* out-of-order */) and scorer(reader, scoreDocsInOrder) is defined abstract. One other optimization is to expose a topScorer() API (on Weight) which returns a Scorer that its score(Collector) will be called, and additionally add a start() method to DISI. That will allow Scorers to initialize either on start() or score(Collector).</td>
</tr>
<tr>
<td>Query</td>
<td>weight</td>
<td>The method Query.weight() was left in Query for backwards reasons in Lucene 2.9 when we changed Weight class. This method is only to be called on top-level queries - and this is done by IndexSearcher. This method is just a utility method, that has nothing to do with the query itself (it just combines the createWeight method and calls the normalization afterwards). For 3.3 I will make Query.weight() simply delegate to IndexSearcher’s replacement method with a big deprecation warning, so user sees this. In IndexSearcher itself the method will be protected to only be called by itself or subclasses of IndexSearcher.</td>
</tr>
</tbody>
</table>
If we look at false positives, for Eclipse (Table 6-17) we can notice examples of descriptions that are too specific (e.g., for the releaseWidget method), hence not particularly useful to properly understand the entire method. Other examples are related to faulty behavior (dispose) and about a possible bug fixing (constructor of OperationCanceledException). For Lucene (Table 6-18), the candidate description of the isOptimized method from the MultiReader class consists, actually, in a proposal of bug fixing for several methods. Regarding the termDocs method from the SegmentReader class, the paragraph mainly describes dependencies among methods rather than describing method behavior. In some sense, this could also be considered a true positive (e.g., useful to understand method dependencies), although our evaluators classified it as a false positive because the paragraph did not clearly describe the method behavior. The last case (the topDocs method from the TopDocsCollector class) is a paragraph where people suggest to deprecate such a method and then integrate the behavior elsewhere. Also in this case, the description could be, in principle, considered a useful one, although it was not considered as such because the paragraph described the behavior to be refactored. In conclusion, false positives either concern borderline cases—which could be useful in some cases and hence increase the amount of useful material a developer has to comprehend the source code—or cases such as faulty or future behavior which would not easy to discern automatically. This also suggests that the results strongly depend on the data source we use (i.e., the content of the emails and, above all, of the bug reports), indicating that some sources, such as bug reports, in some case contain descriptions that are not appropriate for describing the correct, current behavior of a method.

### Table 6-17: Examples of false positive paragraphs for Eclipse

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Paragraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>releaseWidget</td>
<td>Similar (and related) NPE is on Table class, on releaseWidget() method call - the last element in columns[] array is null.</td>
</tr>
<tr>
<td>WorkbenchPart</td>
<td>dispose</td>
<td>It must be the last method called on the contribution item. After calling dispose(), it is a bug to continue using the contribution item</td>
</tr>
<tr>
<td>OperationCanceledException</td>
<td>OperationCanceledException</td>
<td>On thinking about it, throwing OperationCanceledException would be unusual since the method does not take a progress monitor parameter. Returning a CANCEL status seems like the best approach.</td>
</tr>
</tbody>
</table>
Table 6-18: Examples of false positive paragraphs for Lucene

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Paragraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiReader</td>
<td>isOptimized</td>
<td>These 3 methods should probably be fixed: isOptimized() would fail - similar to isCurrent(); setNorm(int, String, float) would fail too, similar reason. directory() would not fail, but fail to return the directory of reader[0]. This is because MultiReader() (constructor) calls super with reader[0] again. I am not sure.</td>
</tr>
<tr>
<td>SegmentReader</td>
<td>termDocs</td>
<td>Yes, but this class is package private and unused! AllTermDocs is used by SegmentReader to support termDocs(null), but not AllDocsEnum. The matchAllDocs was just an example, there are more use cases, e.g. a TermsFilter (that is the non-scoring TermQuery variant): Just use the DocsEnum of this term as the DclidSettlerator</td>
</tr>
<tr>
<td>TopDocsCollector</td>
<td>topDocs</td>
<td>We might also consider deprecating the topDocs() methods that take in parameters and think about how the paging collector might be integrated at a lower level in the other collectors, such that one doesn't even have to think about calling a diff. collector</td>
</tr>
</tbody>
</table>

Finally, concerning the false negatives (Table 6-19 and Table 6-20 for Eclipse and Lucene, respectively), many of them were descriptions discarded because they describe the methods without containing keywords (such as, “return”, “override”, “invoke”, etc.) we used for filtering. For example, in the case of Lucene, the paragraph referring to the optimize method from the IndexWriter class contains the sentence “I found that IndexWriter.optimize(int) method does...” containing the class name and method name, yet it does not contain any of the above keywords. A similar situation occurs for the parse method from the TrecFTParser class. Similar examples can be found in Eclipse, where the doubleClicked method from the JavaStringDoubleClickSelector class is, again, described properly, yet none of the filtering keywords is mentioned. In conclusion, this suggests that some false negatives could have been avoided by weakening the filtering criteria, however this would also have reduced the precision and hence would have increased the amount of (possibly useless) descriptions a developer has to browse.

Table 6-19: Examples of false negative paragraphs for Eclipse

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Paragraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaStringDoubleClickSelector</td>
<td>doubleClicked</td>
<td>What it does is: - change the behavior of the doubleClicked() methods to also consider the endpoint of the mouse selection for its calculation of the text selection.</td>
</tr>
</tbody>
</table>


If I understood TypeDescriptor.initialize() method correctly, it is not interested in the method code, so you could use classReader.accept(visitor, ClassReader.SKIP_CODE) to completely skip all methods code from visiting. Same applies to implementation of SearchEngine.getExtraction(..) and TagScanner.Visitor.getMethods(..) methods, where you also can add ClassReader.SKIP_CODE to avoid visiting method code.

<table>
<thead>
<tr>
<th>Engine</th>
<th>accept</th>
</tr>
</thead>
<tbody>
<tr>
<td>If I understood TypeDescriptor.initialize() method correctly, it is not interested in the method code, so you could use classReader.accept(visitor, ClassReader.SKIP_CODE) to completely skip all methods code from visiting. Same applies to implementation of SearchEngine.getExtraction(..) and TagScanner.Visitor.getMethods(..) methods, where you also can add ClassReader.SKIP_CODE to avoid visiting method code.</td>
<td></td>
</tr>
</tbody>
</table>

Table 6-20: Examples of false negative paragraphs for Lucene

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Paragraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>IndexWriter</td>
<td>optimize</td>
<td>I found that IndexWriter.optimize(int) method does not pick up large segments with a lot of deletes even when most of the docs are deleted. And the existence of such segments affected the query performance significantly. I created an index with 1 million docs, then went over all docs and updated a few thousand at a time. I ran optimize(20) occasionally. What saw were large segments with most of docs deleted. Although these segments did not have valid docs they remained in the directory for a very long time until more segments with comparable or bigger sizes were created.</td>
</tr>
<tr>
<td>TrecFTParser</td>
<td>parse</td>
<td>In TrecFTParser.parse(), you can extract the logic which finds the date and title into a common method which receives the strings to look for as parameters (e.g. find(String str, String start, int startlen, String end)).</td>
</tr>
</tbody>
</table>

6.4.4 Conclusions

We verified in this work our hypothesis that developer communications, such as, mailing lists and bug reports, contain textual information that can be extracted automatically and used to describe methods from Java source code. We found that at least 22% of the methods in Eclipse and 65% of the methods in Lucene are specifically referenced in emails and bug reports. Only a part of these references are included in paragraphs that describe the methods and can be automatically retrieved by our approach. Our approach to mine method descriptions from developers’ communication, and specifically from mailing lists and bug tracking systems, first traces emails/bug reports to classes, and then, after extracting paragraphs, traces them to methods. After that, it relies on a set of heuristics to extract different kinds of descriptions, namely: (i) descriptions explaining methods in terms of their parameters and return values; (ii) descriptions explaining how a method overloads/overrides another method; and (iii) descriptions of how a method works
by invoking other methods. Finally, a further pruning is performed by computing the textual similarity between the paragraphs and the method body.

Our empirical evaluation indicates that the proposed approach is able to identify descriptions with a precision up to 79% for Eclipse and up to 87% for Lucene. The method coverage of these descriptions is low for Eclipse, ranging between 7% and 2%, and higher for Lucene, ranging between 36% and 15%. The low method coverage is the result of two factors: (i) only part of the methods are described properly in these communications, and (ii) our approach is rather conservative as we focused on achieving high precision, given the envision usage scenario (i.e., a developer trying to understand quickly what a method does). Our investigation revealed the presence of linguistic patterns that, at least for the two analyzed systems, characterize different kinds of method descriptions.

There are several directions for future work. First, this is a first approach aimed at mining method descriptions from external unstructured artifacts, therefore there is still a lot of space for improvement with the aim of increasing the precision while keeping the method coverage as high as possible, as well as reducing the percentage of false positives. Furthermore, we would aim at further validating the proposed approach on a larger data set, including mailing lists and bug reports from more systems. It is important to establish whether the values we used for the heuristics of our approach work equally well on other data sets. Finally, we would also investigate approaches for mining descriptions of software artifacts at a higher level of abstraction, such as classes and packages.
7. Conclusions and Future Work

7.1 Conclusions

Current software systems must be continually changed to meet new requirements and adapt them to changing conditions in their operating environment. Additionally, it is widely accepted that effort and time spent understanding parts of a software system are a significant proportion of the resources needed to maintain and evolve existing code (Corbi, 1989). Developers responsible for maintenance tasks are faced every day with software systems with thousands or millions lines of code. This situation is particularly problematic when developers have to deal with large systems developed by others and the code is the only source of information that is available and up to date.

Different maintenance tasks require different levels of code comprehension. Most developers strive to understand as much as it is needed to perform a given task – no more and no less. The amount of code understood at one time is often limited by the ability to memorize, recognize, and recall textual tokens from the source code and their semantics, which makes tool support essential for most software comprehension tasks. Modern IDEs, together with searching and navigation tools, recommendation systems and data mining tools, all help developers minimize their effort in identifying parts of code relevant to their task. A common feature of these tools is that they provide a list of source code elements (such as methods) to the user, which he still has to read and understand in order to make final decision on their relevancy to the task at hand. When the code is well documented internally (for example, a method has good leading comments, meaningful name and parameters), it is frequently sufficient to see such comments and the method header to determine if it is relevant or not (Butler, 2009). However, more often than not, comments are missing, or out of date, and method headers contain words that the developer is not familiar with. In such cases, developers have little choice but to read the implementation and sometimes more than that.
We propose to provide help to developers in such situations. The goal is to supply them with a description of the code (such as the abstract of an article), which is more informative than the header and the leading comments, yet much shorter than the implementation, while capturing the essential information from it. Such descriptions will not replace reading the code when it needs to be understood, but they can save unnecessary effort spent reading irrelevant parts of it.

This dissertation explored the use of text summarization technology for automatically generating such descriptions of source code artifacts. Specifically, we studied how people describe code artifacts using term-based and sentence-based approaches, we adapted extractive and abstractive techniques for automatic code summarization, and also, we used intrinsic evaluation approaches to assess some properties of automatic summaries.

Regarding human summarization, the thesis reports the results of empirical studies aimed at investigating how developers summarize code artifacts when understanding them. Thus, we provided coarse-grained information about the conciseness of the summaries; we characterized the summary terms contained in source code artifacts and the ones that appear only in free form summaries; we categorized the kind of code identifiers often used in the summaries; we determined the parts of speech used by developers when summarizing various types of artifacts; and finally, we found out what developers perceive as important in source code artifacts when understanding and summarizing them. These results are useful in explaining how developers abstract source code, in gaining insights about the comprehension process, in explaining the role of code elements in such a process, and for the development of automatic source code summarization tools.

With regard to automatic generation of summaries, the dissertation describes several approaches for creating short and accurate textual descriptions for various types of code entities. Firstly, we proposed and validated an approach for summarizing methods and classes that uses a combination between techniques making use of the position of terms in code artifacts and text retrieval techniques. Secondly, we presented an approach to automatically generate structured natural language summaries for Java classes. The approach leverages information about the class stereotypes and uses existing text generation tools to compose summaries based on a set of heuristics we developed. While
the techniques used to generate the summaries are adapted from prior work, this is a first
technique that generates natural language summaries for classes automatically. Lastly,
we proposed and validated an approach to automatically extract method descriptions from
communications in bug tracking systems and mailing lists. In this way, we verified in this
work our hypothesis that developer communications, such as, mailing lists and bug
reports, contain textual information that can be extracted automatically and used to
describe methods from Java source code. The results of the evaluations all these
approaches indicate that text summarization techniques are suitable for automatic source
code summarization since developers generally agree with the summaries produced by
the proposed methods. Thus, these summaries can be useful for improving software
comprehension processes, which usually occur during software maintenance tasks.

7.2 Work in progress and future work

The research described in this dissertation is a promising foundation for the future work.
First of all, regarding human summarization, we are replicating the study with other
subjects, mainly native speakers, to analyze the influence of the native language in
drafting descriptions of source code. Also, our future work will aim at performing deeper
analyses on the data set, for instance, analyzing semantic relations among words and
considering factors not investigated in this study; and finally, further exploiting what we
learned to build and improve automatic summarization tools.

With regard to summarization tools, the evaluation study revealed several areas where
the class summarization tool can be improved. We plan to carry out such improvements
and hope that they will result in higher quality summaries. For example, for the stereotype
identification we plan to enrich both taxonomies – method and class stereotypes. This will
also help in refining the heuristics to select the relevant content of a class. We also plan to
consider the context of the class (i.e., the relationships to other classes) to enrich the
summaries. When comments are present, they may also be used for generating the
summaries. We will investigate this problem. Since the goal of our automated
summarization tools is to approximate developers’ reasoning, we plan to analyze how
automatic summaries compare to free-form natural language summaries generated by
developers, using formal text summarization measures.
Our first approach aimed at mining method descriptions from external unstructured artifacts can be extended and improved in many ways with the aim of increasing the precision while keeping the method coverage as high as possible, as well as reducing the percentage of false positives. Furthermore, we would aim at further validating the proposed approach on a larger data set, including mailing lists and bug reports from more systems. It is important to establish whether the values we used for the heuristics of our approach work equally well on other data sets. Finally, we would also investigate approaches for mining descriptions of software artifacts at a higher level of abstraction, such as classes and packages.

Finally, we will perform extrinsic evaluations of the automatic summaries by using them to support various software engineering tasks, such as: concept location, impact analysis, ontology extraction, recovering traceability links between software artifacts, etc. In (Aponte and Marcus, 2011) we already proposed the use of summaries as aids for improving traceability link recovery methods. Many solutions to the traceability link recovery problem are based on the use of text retrieval techniques (Antoniol, et al., 2000) (Marcus and Maletic, 2003), under the assumption that extracting and analyzing textual information contained in the software artifacts is an effective way to determine whether they are related. Most such methods generate a list of candidate links that must be examined by software engineers. They have to review each candidate link in order to determine those that are correct links and discard the false positives. This is a laborious activity as it requires a detailed study of a long list of software artifacts of various kinds. Therefore, (Aponte and Marcus, 2011) proposes automatically generating summaries of software artifacts, and offering developers these summaries as a first tool during candidate link analysis. Small artifacts (e.g., a method with less than ten lines of code; a short section in a document; etc.) do not require such summaries as they are easy to read by the developers. Large artifacts (e.g., a class with hundreds of methods; a long chapter in a document; etc.), on the other hand, can benefit from summarization. We expect that developers would be able to make proper decisions on many candidate links, without reading in details the original artifacts, but only their summaries instead. Some of the summaries will not be informative enough to help in these decisions, so developers would still have to read the original artifact. The overhead in such cases is minimal (i.e., reading the summary in addition to the artifact) and it is outweighed significantly by the potential benefits in the other cases.
In the long term, we expect that the use of source code summaries will reduce the developers’ cognitive effort during comprehension activities. This should positively impact the time of development, as well as the quality of the software produced. More than that, the summaries could be consumed not just by developers, but also by other tools. For example, we envision using the source code summaries to support tools for automatic reverse engineering of legacy code, software ontologies extraction, re-documentation, etc. We expect the summaries to be used by existing searching and navigation tools.
References


on Information and Knowledge Management. ACM, Atlanta, Georgia, USA, pp. 365-370.


Miller, G.A. (1956) The Magical Number Seven, Plus or Minus Two Some Limits on Our Capacity for Processing Information, Psychological Review, 63, 81-97.


