AUTOMATIC GENERATION AND ASSESSMENT OF SOURCE CODE METHOD SUMMARIES

A dissertation submitted
to Kent State University in partial
fulfillment of the requirements for the
degree of Doctor of Philosophy

by
Nahla Abid
April 2017
Dissertation written by

Nahla Abid

B.Sc., King Abdulaziz University, Jeddah, Saudi Arabia, 2008

M.S., Bowling Green State University, USA, 2012

Ph.D., Kent State University, USA, 2017

Approved by

_______________________________________
Dr. Jonathan Maletic, Doctoral Advisor, Department of Computer Science

_______________________________________
Dr. Mikhail Nesterenko, Department of Computer Science

_______________________________________
Dr. Gwenn L. Volkert, Department of Computer Science

_______________________________________
Dr. Bonita Sharif, Department of Computer Science and Information Systems, Youngstown State University

_______________________________________
Dr. Jocelyn R. Folk, Department of Psychological Sciences

_______________________________________
Dr. Christopher Was, Department of Psychological Sciences

Accepted by

_______________________________________
Dr. Javed I. Khan, Professor, Chair, Department of Computer Science

_______________________________________
Dr. James L. Blank, Professor, Dean, College of Arts and Sciences
# TABLE OF CONTENTS

TABLE OF CONTENTS ........................................................................................................... III

LIST OF FIGURES ................................................................................................................ II

LIST OF TABLES .................................................................................................................. XI

ACKNOWLEDGEMENTS ........................................................................................................ XV

CHAPTER 1 INTRODUCTION ............................................................................................... 1

1.1 Research Focus .................................................................................................................. 4

1.2 Research Goal .................................................................................................................... 5

1.3 Contributions ..................................................................................................................... 5

1.4 Organization ..................................................................................................................... 6

1.5 Publication Notes ............................................................................................................. 7

CHAPTER 2 BACKGROUND AND RELATED WORK .................................................... 8

2.1 Stereotype Description and Identification ........................................................................... 8

2.2 Source-Code Summarization .......................................................................................... 10

2.2.1 Natural-Language Processing (NLP) ......................................................................... 11

2.2.2 Generating Summaries for Java Methods ................................................................. 12

2.2.3 Describing High-Level Action of Java Methods ....................................................... 15

2.2.4 Describing the Formal Parameters of Java Methods ................................................. 18

2.2.5 Including Context of Methods Using Call Graph .................................................... 20

2.2.6 Determining Loop Actions ......................................................................................... 22

2.2.7 Text Retrieval (TR) ..................................................................................................... 25

2.2.7.1 VSM and LSI ........................................................................................................... 25
4.1 Study Participants ................................................................. 64
4.2 Experimental Design and Procedure ........................................... 64
4.3 Data Collected ........................................................................... 66
4.4 Results and Discussion ................................................................. 67
  4.4.1 Per System ........................................................................ 68
  4.4.2 Accuracy .......................................................................... 69
  4.4.3 Missing/Containing Important Information ................................. 71
  4.4.4 Including Unnecessary Information ........................................ 74
4.5 Threats to Validity ...................................................................... 75
4.6 Conclusions ............................................................................. 76

CHAPTER 5 IMPROVING AND EVALUATING THE IMPORTANCE OF THE
FOUR DOCUMENTATION SECTIONS ........................................... 77

5.1 Improving The Automated Approach ........................................... 77
5.2 Second Study ........................................................................... 78
  5.2.1 Participants ........................................................................ 78
  5.2.2 Experimental Design and Procedure .................................. 79
5.3 Results and Discussion ................................................................. 80
  5.3.1 No short description .......................................................... 82
  5.3.2 No calls and stereotypes ..................................................... 83
  5.3.3 No data members read ........................................................ 84
  5.3.4 No list of objects ................................................................ 85
5.4 Threats to Validity ..................................................................... 86
5.5 Conclusions ................................................................................................................. 86

CHAPTER 6 EYE TRACKING STUDY OF SUMMARIZING SOURCE CODE. 87

6.1 Study Participants ........................................................................................................ 89
6.2 Study Systems and Method Selection ......................................................................... 89
6.3 Study Procedure and Instrumentation ....................................................................... 91
6.4 Eye Tracking Apparatus and Measures .................................................................... 92
6.5 Correctness .................................................................................................................. 93
6.6 Gaze Behavior Inside the Method ............................................................................ 93
   6.6.1 Term locations Identification .............................................................................. 94
   6.6.2 Adjusted gaze time and visits ............................................................................ 96
   6.6.3 RQ1: Three Source-code Locations .................................................................... 97
      6.6.3.1 Signature ..................................................................................................... 97
      6.6.3.2 Calls .......................................................................................................... 98
      6.6.3.3 Control Flow ............................................................................................... 100
   6.6.4 RQ2: Analysis Based on Methods’ Sizes ............................................................ 101
   6.6.5 RQ3: Analysis Based on Method Stereotypes .................................................... 103
      6.6.5.1 Predicate ..................................................................................................... 104
      6.6.5.2 Property ..................................................................................................... 105
      6.6.5.3 Void-accessor .............................................................................................. 106
      6.6.5.4 Command .................................................................................................. 108
   6.6.6 Discussion ............................................................................................................. 109
   6.6.7 Comparison to Rodeghero et al.’s Study ............................................................. 110
6.6.8 Threats to Validity ................................................................. 111

6.7 Analyzing Eye-tracking data at Line Level................................. 113
6.7.1 Gaze time of source-code lines ............................................. 114
6.7.2 Heat Map .............................................................................. 116
6.7.3 RQ4: Gaze Time VS. Line’s Length ....................................... 118
6.7.4 RQ5: Gaze Time and Line Usage .......................................... 121
   6.7.4.1 The Similarity Between Lines Used ................................. 125
6.7.5 RQ6: Type of Lines Used ...................................................... 126
6.7.6 Discussion ............................................................................. 127
6.7.7 Threats to Validity ................................................................. 129

6.8 Conclusions ............................................................................ 129

CHAPTER 7 CODE COMPREHENSION DURING SUMMARIZING TASK .... 131

7.1 Gaze Behavior Outside the Method vs. Inside the Method ............. 132
7.1.1 RQ1: Time spend Outside the method scope .......................... 133
7.1.2 RQ2: Locations examined outside the scope of the assigned method .... 134

7.2 Reading Pattern ..................................................................... 137
7.2.1 Top-to-bottom vs Bottom-to-top .......................................... 137
7.2.2 Skimming vs Through ......................................................... 138
7.2.3 Disorderly vs Sectionally ..................................................... 138
7.2.4 Compare Experts’ Patterns to Rodeghero’s Patterns .................. 139

7.3 The Comprehension Process of Developers ............................... 140
7.3.1 Top-down vs. Bottom-up ..................................................... 141
7.3.2 Timeline of Participants’ Mental Models .................................................. 143

7.4 Conclusion ........................................................................................................ 147

CHAPTER 8 CONCLUSIONS ................................................................................ 149

8.1 Contributions .................................................................................................. 151

8.2 Future work ...................................................................................................... 152

APPENDIX A EXAMPLES OF METHODS REDOCUMENTED BY THE SUMCODE ......................................................... 154

A.1 Accessor Methods ......................................................................................... 154

A.2 Mutator Methods ............................................................................................ 160

A.3 Collaborational Methods ............................................................................... 163

A.4 Factory Methods ............................................................................................ 164

APPENDIX B CLUSTERING THE DURATION OF LINES THAT DEVELOPERS READ DURING SUMMARIZATION ............ 165

B.1 Experts’ Clusters ............................................................................................ 165

B.2 Novices’ Clusters ........................................................................................... 167

REFERENCES ....................................................................................................... 169
LIST OF FIGURES

Figure 1. A method of the class DataSource in the C++ system HippoDraw after re-documentation with the method stereotype ................................................................. 10

Figure 2. An example of ending summary and data-facilitating summary units .............. 13

Figure 3. An example of a sequence of method calls as single action .......................... 16

Figure 4. An example of generating a synthesized description for a conditional expression ........................................................................................................ 16

Figure 5. An example of generating a synthesized description for max loop .............. 17

Figure 6. The five item/locations of a loop-if that used in the action identification model ........................................................................................................... 23

Figure 7. An example of for-loop .................................................................................. 24

Figure 8. An example of a property method and its generated documentation summary. The documentation is inserted into the code as a method comment ................ 47

Figure 9. The overview of systems used to facilitate generating the documentation summaries ........................................................................................................ 48

Figure 10. The template of the four sections of the documentation summary .............. 51

Figure 11. The documentation summary of a property method with multiple returns.... 53

Figure 12. The documentation summary of a predicate method ................................. 53

Figure 13. An example of a void-accessor method and its generated documentation summary. ........................................................................................................... 54

Figure 14. A generated documentation summary for command method ..................... 57

Figure 15. A non-void-command collaborator method and its documentation .......... 58
Figure 16. A *factory* method and its documentation.................................................. 59
Figure 17. Overall ratings of graduate and undergraduate ........................................ 68
Figure 18. Ratings for the three systems........................................................................ 68
Figure 19. The short description generated for a *command collaborator* method. ....... 71
Figure 20. An example of a void-accessor method and its improved summary. .......... 77
Figure 21. A comparison of the first and second evaluation. ....................................... 81
Figure 22. Heat map generated for one method and one participant ......................... 117
Figure 23. The three steps performed to answer RQ6. ............................................... 122
Figure 24. The four chunks of one of the method used in the study........................... 142
Figure 25. The timeline of an expert summarizing the method in Figure 22 ............... 144
Figure 26. The timeline of one novice summarizing the method in Figure 22............ 145
Figure 27. The timeline of an expert during one summarization task ....................... 146
Figure 28. The timeline of a novice during one summarization task......................... 146
**LIST OF TABLES**

Table 1. Taxonomy of method stereotypes as given in (N. Dragan et al. 2006) ............ 9
Table 2. Text generation for single method call .............................................................. 14
Table 3. Text generation for Assignment ......................................................................... 14
Table 4. Text generation for conditional expression that is a method call .................... 14
Table 5. An example of generating summary for a parameter .................................... 19
Table 6. An example of summaries generated from method signature ....................... 21
Table 7. The eight Features and their Values ................................................................. 24
Table 8. The value of the eight features of “find” stereotype ..................................... 24
Table 9. Attributes used to generate different types of summaries............................ 27
Table 10. The different weight assigns to terms based on their locations .................... 29
Table 11. Example of extracting the comment from Q&A text ..................................... 31
Table 12. The summary phrase of a set method ............................................................. 33
Table 13. The summary phrase of a command method............................................... 33
Table 14. The list of templates used to generate documentation summaries for *Accessor* methods.................................................................................................................. 56
Table 15. The list of templates used to generate documentation summaries for *mutator* methods.................................................................................................................. 60
Table 16. Questions used in the study. The First four questions are answerable as

“Strongly Agree”, “Agree”, “Disagree”, and “Strongly Disagree.” The Last question is open-ended.................................................................................................................. 63
Table 17. An overview of the software systems used in the study. .............................. 64
Table 18. The stereotype of the methods used in the study ......................................................... 65
Table 19. The distribution of the participants' responses ................................................................. 66
Table 20. The distribution of the participants' responses per stereotype. Note that clb is short for collaborator. .................................................................................................................. 69
Table 21. Mann-Whitney Test for doc. without short description .................................................. 83
Table 22. Mann-Whitney Test for doc. without list of calls ............................................................ 83
Table 23. Mann-Whitney Test for doc. without list of DM read ................................................... 85
Table 24. Mann-Whitney Test for doc. without list of objects ..................................................... 85
Table 25. An overview of the Java systems used in the study .......................................................... 90
Table 26. Statistical summary of Wilcoxon Test of experts and novices. n is the number of samples. A sample is result from one programmer and one method. (*) indicates that p is significant .................................................................................................................. 99
Table 27. Wilcoxon Test Summary of experts for large, medium, and small methods. (*) indicates that p is significant ................................................................................................................. 101
Table 28. Wilcoxon Test Summary of novices for large, medium, and small methods. (*) indicates that p is significant ............................................................................................................. 102
Table 29. Wilcoxon Test Summary of novices for predicate methods. (*) indicates that p is significant ................................................................................................................................. 104
Table 30. Wilcoxon Test Summary of experts and novices for property methods. (*) indicates that p is significant ......................................................................................................................... 106
Table 31. Wilcoxon Test Summary of experts and novices for void-accessor methods. (*) indicates that p is significant ................................................................................................................. 107
Table 32. Wilcoxon Test Summary of experts and novices for command methods. (*) indicates that p is significant ................................................................. 108

Table 33. A comparison of our result to Rodeghero et al. ................................................. 111

Table 34. Wilcoxon Test Summary of experts and novices After adding the non-identified records to non-call terms. (*) indicates that p is significant .................. 112

Table 35. K-mean clusters of experts’ lines where k = 2. ................................................. 115

Table 36. K-mean clusters of experts’ lines where k = 3. ................................................. 115

Table 37. K-mean clusters of novices’ lines where k = 2 .................................................. 116

Table 38. K-mean clusters of novices’ lines where k = 3 .................................................. 116

Table 39. The average number of lines on each Heat map category for experts and novices ........................................................................................................ 118

Table 40. A summary of the Kendall’s tau comparing the set of ranks based on the gaze time and the set of ranks based on the number of identifiers. ......................... 120

Table 41. A summary of the Kendall’s tau comparing the set of ranks of the actual gaze time and the set of ranks of the adjusted gaze time based on the number of identifiers ........................................................................................................ 121

Table 42. A summary of number of lines used by experts that have the top efforts of 181 lines ............................................................................................................. 124

Table 43. A summary of number of lines used by novices that have the top efforts of 634 used lines ................................................................................................. 124

Table 44. The Overlap between lines used by all experts to write their summaries....... 125

Table 45. The Overlap between lines used by all novices to write their summaries..... 125
Table 46. Lines used by experts that fall into one of the summary units proposed by the literature automatic summarization/documentation. ......................................................... 127

Table 47. The average time spent by each expert reading different locations outside the assigned method. ........................................................................................................ 135

Table 48. The average time spent by each novice reading different locations outside the assigned method. ........................................................................................................ 135

Table 49. Statistical summary of Wilcoxon Test of experts and novices that compares the three reading patterns........................................................................................................ 139

Table 50. The average time of reading bottom-up or top-down by experts and novices. ........................................................................................................................................ 142

Table 51. Statistical summary of Wilcoxon Test of experts and novices that compares Top-down and Bottom-up ........................................................................................................ 143

Table 52. The Average actual duration and percentage of duration spent in each phase by experts........................................................................................................................................ 147

Table 53. The Average actual duration and percentage of duration spent in each phase by novices........................................................................................................................................ 147
ACKNOWLEDGEMENTS

I am very grateful to my advisor, Prof. Jonathan Maletic, for helping and supporting me. The door to Prof. Maletic office was always open whenever I ran into a trouble spot or had a question about my research.

I am also thankful to Dr. Natalia Dragan and Hend Alrasheed for supporting the conducting of some these studies.

A special thank to Dr. Bonita Sharif who ran the eye-tracking experiment. The eye-tracking study could not have been done without her assistance. I would like to individually thank every single person who participated in the studies presented here.

I also want to thank my Dissertation Committee for the valuable feedback and everyone in the Department of Computer Science at Kent State University who contributed towards the completion of the dissertation.

I am grateful to my father (Jamal Abid) and my mother (Mahdiah Abid), my brothers, and my sister, who have provided me through moral and emotional support in my life. I am also grateful to my friends who have supported me along the way.

Nahla Abid

March 2017, Kent, Ohio
CHAPTER 1
INTRODUCTION

During the evolution of large software systems, developers spend a great deal of time trying to locate and understand the parts of the system that are related to their current task (Fluri, Wursch, & Gall 2007; Ko, Myers, Coblenz, & Aung 2006; LaToza, Venolia, & DeLine 2006). A typical practice for developers is to alternate between the source code and the associated internal documentation (aka comments) to understand the intent of a method or function. Furthermore, comments are found to be critical in assisting developers to understand code faster and more deeply. Unfortunately, comments are oftentimes outdated or incomplete (Fluri et al. 2007) due to lack of either proper maintenance or adherence to proper documentation standards.

One solution is the use of meaningful and descriptive identifiers to describe the intent of a method or a class. However, long identifiers are often found confusing, reducing readability (Sridhara, Hill, Muppaneni, Pollock, & Vijay-Shanker 2010) and sometimes misleading (Ko et al. 2006). Comments are a better indicative of the purpose and intent of a source code artifact if they are written well and kept up to date (Ko et al. 2006).

Recently, several techniques have been proposed to automatically generate summaries directly from source code methods (Eddy, Robinson, Kraft, & Carver 2013; Haiduc, Aponte, Moreno, & Marcus 2010; McBurney & McMillan 2014; Sridhara et al. 2010; Sridhara, Pollock, & Vijay-Shanker 2011a; Vassallo, Panichella, Di Penta, & Canfora 2014; Wong, Yang, & Tan 2013) and classes (Eddy et al. 2013; Haiduc et al. 2010; L.
Moreno et al. 2013). Natural language processing (NLP) techniques have been used to generate English descriptions for Java methods (Sridhara et al. 2010; Sridhara et al. 2011a). These approaches analyze the part-of-speech tag of identifiers (method or variable names) to capture the intent of methods. While these approaches produce some reasonable results, its performance depends upon high-quality identifiers and method names. For example, when grouping method calls, NLP requires that all method names start with the same verb (Sridhara et al. 2011a). Furthermore, it is observed that the technique often selects lines of code that are considered essential for the method, but does not generate English descriptions for these lines (McBurney & McMillan 2014). Consequently, if identifiers and methods are poorly named, the approach may fail to generate accurate comments or any reasonable comments at all.

In the approach proposed here, we do not depend on NLP methods. Instead, the documentation (for methods) are generated using predefined fill-in-the-blank sentence phrases, which we refer to as templates. We constructed the templates specifically for different method stereotypes (N. Dragan, Collard, & Maletic 2006). Stereotypes reflect the basic meaning and behavior of a method and include concepts such as predicate, set, and factory. The stereotype information is leveraged to construct a custom template phrase for each stereotype to form the summaries. After the appropriate templates are matched to the method, they are filled with data to form the complete documentation. To fill in the templates, static analysis and fact extraction on the method is done using the srcML (Collard, Decker, & Maletic 2011) infrastructure (www.srcML.org).
Two studies involving 139 programmers are conducted. They use five C++ software systems to evaluate the accuracy and perceived value of the generated documentation. The result of the first study reveals that the generated documentation is accurate, does not include unnecessary information, and does a reasonable job describing what the method does. The second study shows that the most important part of the documentation is the short description followed by the list of calls and their stereotypes.

To further improve the automatic source-code summarization, we conduct an eye-tracking study of programmers summarize a set of Java methods. The previous literature suggested that the words/terms and lines/locations that programmers spend more time reading are more important for comprehension (Bednarik & Tukiainen 2006; Crosby & Stelovsky 1990) and can be used to enhance automatic summarization (Rodeghero, McMillan, McBurney, Bosch, & D’Mello 2014).

Our study expands the study done by (Rodeghero et al. 2014). We use a tool, iTrace, that is an Eclipse plugin to incorporate implicit eye tracking (Shaffer et al. 2015) that supports scrolling/switching between files. Therefore, the size of source code presented on the screen is not limited.

Data collected from 18 developers includes eye gazes on source code, written summaries, and time to complete each summary. When data is analyzed at terms level, eye-movement behavior of a developer is closely related to their level of expertise. Experts tend to revisit control flow terms rather than focusing on them for a long period. Novices tend to spend a significant amount of gaze time and visits when they read call and control flow terms. At line level, we found strong evidence that gaze time can predict about 70% of
lines used by experts to write their summaries by considering the lines with the top 30% gaze time effort. Furthermore, through manual inspection, we found that 95% of lines used by experts to write their summaries fall into of the summary units (lines) proposed by the literature on source-code summarization.

Finally, we propose using eye-movements of a developer to speculate the cognition model applied during the comprehension process. The cognition models examine how programmers understand source code by describing the temporary information structures in the programmer’s head (Von Mayrhauser & Vans 1995). The two type of models that we are interested in are top-down and bottom-up. The top-down model is usually applied as-needed (i.e., the domain of the system is familiar). The bottom-up model is applied when a developer is not familiar with the domain of the source code. On average, experts and novices read the assigned method more closely (using bottom-up mental model) than bouncing (using top-down). However, on average, novices spend a longer gaze time preforming bottom-up (66780ms) compare to experts (43913ms).

1.1 Research Focus

This work examines two closely related problems which are: source-code summarization and source-code comprehension. The first goal is to utilize the stereotypes of methods to generate automated documentation. The second goal is to improve source-code-summarization via eye-tracking study of developers summarizing source-code methods. The eye-tracking collected is also analyzed to study the type of activities that a developer performs during the comprehension process. The main questions of the research are as follows.
1) To what extent can method stereotypes be efficiently used to generate documentation for object-oriented methods?

2) When developers write summaries for object-oriented methods:
   a. What part of the code do they focus on the most?
   b. Do locations out-the-scope of the assigned method are examined?
   c. Can the eye-tracking data be used to predict the type of the mental models applied during the comprehension process?

1.2 Research Goal

The goal is to examine and develop the use of method stereotypes to generate documentation for OO methods. This can assist in improving program comprehension during maintenance. Additionally, the result from eye-tracking study about what developers focus on during summarization can be used to improve automated summarization.

1.3 Contributions

The main contributions of this work is to study and experiment ways to improve automatic documentation. Furthermore, we analyze the comprehension process of developers during summarization tasks. Considering source-code summarization and comprehension, we make the following contributions:

1) An automatic approach that generates documentation summaries for C++ methods using predefined templates.
2) A user study included 64 programmers to evaluate the accuracy and content adequacy of the automated approach.

3) A second user study included 75 programmers to evaluate the importance of different sections of our automated generated summaries.

4) An eye-tracking study of 18 developers performing summarizing tasks.

5) A comparison of the findings of our study to the related literature (Rodeghero et al. 2014).

6) A comparison of eye gaze time and number of visits on source-code entities with respect to level of expertise (novices or experts), method size, and method stereotypes.

7) Examine whether the most viewed lines approximate lines that people used in summary.

8) Mapping lines used by experts to write their summaries to the existing literature in summarization.

9) An analysis of locations outside the method scope that developers target during method summarization.

10) An approach to predict the mental model of a developer by analyzing the set of activities performed during the compression process (e.g., jumping around the code).

1.4 Organization

The dissertation is organized as follows. First, Chapter 2 presents the related work on source-code summarization and eye-tracking in software engineering. Then, an overview about the approach and proposed templates are given in Chapter 3. Chapter 4 and Chapter 5 demonstrate the result of the two studies to evaluate the automated summarization.
approach. The eye-tracking study of developers summarizing source code is presented in Chapter 6 and Chapter 7. Finally, Chapter 8 is the conclusion and the future directions work.

1.5 Publication Notes

Partial results presented in Chapter 3 have been published at the 31th International Conference on Software Maintenance and evolution (ICSME 2015).
CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter discusses background and related work relevant to the dissertation research. First an overview and definition is given for stereotypes as this is a key component of the research. Next a literature survey on research concerning source code summarization is presented. Lastly, an overview of the eye tracking literature is given.

2.1 Stereotype Description and Identification

Method stereotypes are a concise abstraction of a method’s role and responsibility in a class and system. They are widely used to informally describe methods. A taxonomy of method stereotypes (see Table 1) and techniques to automatically reverse-engineer stereotypes for existing methods was presented by Dragan et al. in (N. Dragan et al. 2006). This work was further extended to support the automatic identification of class stereotypes (Natalia Dragan, Collard, & Maletic 2010). We refer the readers to those papers for complete details on computing method stereotypes; however, the main points will be presented.

The taxonomy is developed primarily for C++ but it can be easily applied to other OO programming languages. Based on this taxonomy, static-program analysis is used to determine the stereotype for each method in an existing system. The taxonomy is organized by the main role of a method while emphasizing the creational, structural, and collaborational aspects with respect to a class’s design. Structural methods provide and
support the structure of the class. For example, *accessors* read an object’s state, while *mutators* change the state. *Creational* methods create or destroy objects of the class. *Collaborational* methods characterize the communication between objects and how objects are controlled in the system. *Degenerate* are methods where the *structural* or *collaborational* stereotypes are limited. Methods can be labeled with one or more stereotypes. That is, methods may have a single stereotype from any category and may also have secondary stereotypes from the *collaborational* and *degenerate* categories. For example, a two-stereotype *get collaborator* method returns a data member that is an object or uses an object as a parameter or a local variable.

**Table 1. Taxonomy of method stereotypes as given in (N. Dragan et al. 2006)**

<table>
<thead>
<tr>
<th>Stereotype Category</th>
<th>stereotype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Structural Accessor</strong></td>
<td>get</td>
<td>Returns a data member.</td>
</tr>
<tr>
<td></td>
<td>predicate</td>
<td>Returns Boolean value which is not a data member.</td>
</tr>
<tr>
<td></td>
<td>property</td>
<td>Returns information about data members.</td>
</tr>
<tr>
<td></td>
<td>void-accessor</td>
<td>Returns information through a parameter.</td>
</tr>
<tr>
<td><strong>Structural Mutator</strong></td>
<td>set</td>
<td>Sets a data member.</td>
</tr>
<tr>
<td></td>
<td>command</td>
<td>Performs a complex change to the object’s state (this).</td>
</tr>
<tr>
<td></td>
<td>Non-void-command</td>
<td></td>
</tr>
<tr>
<td><strong>Creational</strong></td>
<td>constructor, copy-const, destructor, factory</td>
<td>Creates and/or destroys objects.</td>
</tr>
<tr>
<td><strong>Collaborational</strong></td>
<td>collaborator</td>
<td>Works with objects (parameter, local variable and return object).</td>
</tr>
<tr>
<td></td>
<td>controller</td>
<td>Changes an external object’s state (not this).</td>
</tr>
<tr>
<td><strong>Degenerate</strong></td>
<td>incidental</td>
<td>Does not read/change the object’s state.</td>
</tr>
<tr>
<td></td>
<td>empty</td>
<td>Has no statements.</td>
</tr>
</tbody>
</table>
A tool (N. Dragan et al. 2006), *StereoCode*, was developed that analyzes and re-documents C++ source code with the stereotype information for each method. Redocumenting the source code is based on srcML (Source Code Markup Language) (Collard et al. 2011), an XML representation of source code that supports straight forward static analysis of the code.

In order to provide the method-stereotype identification, the source code is translated srcML, and then, *StereoCode* takes over by leveraging XPath, an XML standard for addressing locations in XML. For details about the rules for identifying each method stereotype, we refer the readers to (N. Dragan et al. 2006). Adding the comments (annotations) to source code is quite efficient in the context of srcML. The XPath query provides a location of the method and then a simple transformation is performed within the srcML document to add the necessary comments. This process is fully automated and very efficient/scalable. Figure 1 shows an example of a method that stereotyped as predicate.

```cpp
/** @stereotype predicate */
bool isValidLabel (const string& label) const;
```

**Figure 1.** A method of the class DataSource in the C++ system HippoDraw after re-documentation with the method stereotype

### 2.2 Source-Code Summarization

The relevant research on source code summarization is now presented. Several approaches are proposed to generate automatic summaries using Natural Language Processing (NLP) (McBurney & McMillan 2014; Sridhara et al. 2010), text retrieval (Eddy et al. 2013; Haiduc et al. 2010), and static analysis (Abid, Dragan, Collard, & Maletic 2015;...
L. Moreno et al. 2013). Furthermore, generating summarization for source-code-related artifacts such as cross-cutting concerns (Rastkar, Murphy, & Bradley 2011), bug reports (Rastkar, Murphy, & Murray 2010), and class diagrams (Burden & Heldal 2011.) are proposed and used effectively for maintenance tasks. This section explores different summarization techniques. The techniques are group based on the type of the underlying approach used such as: natural language processing (NLP), text retrieval (TR), mining excising comments, or static analysis.

2.2.1 Natural-Language Processing (NLP)

Sridhara et al. proposed techniques to automatically generate natural-language comments for Java methods (Sridhara et al. 2010), sequences of statements (Sridhara et al. 2011a), and formal parameters (Sridhara, Pollock, & Vijay-Shanker 2011b) using NLP. All these techniques follow three main steps: preprocessing (extracting and identifying the linguistic elements), content selection (selecting a group statements to form the automated summary), and summary generation (creating English phrases from the selected statements).

Preprocessing involves such things as camel case splitting. It is used to split identifiers such as: method names, parameter names, or local variables based on capital letters, underscores, and numbers. For example, “XYLine3DRenderer” is split into “xy line 3 d renderer”.

Abbreviations can also be expanded to better match the corresponding linguistic term. Abbreviations in variable and type names can reduce the readability of generated description (e.g., Button butSelectAll, MouseEvent evt). Techniques from prior work are
used to automatically identify and expand abbreviations in code (Hill, Muppaneni, Pollock, & Vijay-Shanker 2009).

Software Word Usage Model (SWUM). SWUM, proposed by Hill (Hill et al. 2009), is used to identify linguistic elements of a method. It used to capture the action, theme, and secondary arguments for a given method along with its program structure. Based on common Java naming conventions, it is assumed that most method names start with a verb. Therefore, SWUM assigns this verb as the action and looks for the theme in the rest of the name, the formal parameters, and then the class. If the method does not start with a verb, SWUM infers an action for the method, such as “get” or “convert”. In general, the position of a word within a name (e.g., beginning, end) or its structural location (e.g., formal name, formal type, class) is used to determine the appropriate verb. For example, the action in the method void saveImage() is “save”, whereas in Image savedImage(), the action is “get”, and in void image-Saved() the action is “reacting to” the event of the image being saved.

2.2.2 Generating Summaries for Java Methods

The two steps of generating summaries for Java methods are: content selection and summary generation. Based on structural and linguistic clues in the method, the important lines of code from the method are identified (Sridhara et al. 2010). Then, using part-of-speech tags, phrases from each chosen line of code are created and concatenated to form the final summary.

Summary unit (or S_unit) selection. The primary goal in s_unit selection is to choose the important or central lines of code to be included in the summary. Based on studying comments from popular open source Java programs and surveying experienced
Java programmers, five different s_units are identified: ending, void-return, same-action, data-facilitating, and controlling s_units (Sridhara et al. 2010). For illustration, ending and data-facilitating are discussed here.

Ending s_unit is the last statement in a method. Such a statement could be a candidate to be s_unit since a method often performs series of actions to achieve a final action. A single-entry single exit Control Flow Graph (CFG) is used to extract the last statement. On the other hand, data-facilitating s_units assign data to variables used in s_units identified by ending s_unit or other types of s_units. In Figure 2, the s_unit on line 6 is an ending s_unit and line 5 data facilitator. Data facilitating s_units are found by the def-use chains for the method.

```java
1 void returnPressed() {
2    Shell s = getShell();
3    String input = s.getEnteredText();
4    history.addElement(input);
5    String result = evaluate(input);
6    s.append(result);
7 }
```

**Figure 2. An example of ending summary and data-facilitating summary units**

**Text Generation.** The last step is to generate the automated summaries from the selected s_unit by applying pre-defined templates. Templates are created based on the statement categories which are: single method call, return statement, assignment statement, conditional expressions, and loop expressions. One category may have several templates. For illustration, a set of examples are presented here.
**Single method call.** Using SWUM (Hill et al. 2009), action (verb), theme (noun phrase), and secondary arguments (prepositional phrases) are identified. An example is shown in Table 2.

<table>
<thead>
<tr>
<th>Template</th>
<th>action theme secondary-args</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_unit</td>
<td>os.print(msg);</td>
</tr>
<tr>
<td>Summary</td>
<td>/* Print message to output stream */</td>
</tr>
</tbody>
</table>

**Assignment.** Consider the general form of an assignment $s_{\text{unit}}$, $\text{lhs} = M(\ldots)$. The general template is shown in Table 3.

<table>
<thead>
<tr>
<th>Template</th>
<th>action theme secondary-args and get lhs</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_unit</td>
<td>fileName = showFileDialog(\ldots);</td>
</tr>
<tr>
<td>Summary</td>
<td>/* Show file dialog and get file name */</td>
</tr>
</tbody>
</table>

**Conditional Expressions.** The if-expression could be a single variable, method call, or comparison statements. When if-expression has a method call that begins with an auxiliary verb (e.g., is), the subject is in the method name or is a receiver object or parameter of the method call. In Table 4, the subject is the parameter.

<table>
<thead>
<tr>
<th>If expression</th>
<th>A method call begins with an auxiliary verb</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_unit</td>
<td>if (AbstractDoc.hasOnlySpaces(prefix))</td>
</tr>
<tr>
<td>Summary</td>
<td>/* If prefix has only spaces */</td>
</tr>
</tbody>
</table>

For the full list of templates please refer to (Sridhara et al. 2010). Their proposed approach is evaluated by 12 human subjects. The selected $s_{\text{units}}$ and the generated
summaries are concluded to be accurate, missing some important information and concise. In term of accuracy, two cases require SWUM to more precisely identify the action and theme. In term of content adequacy, in 7 comments out of 48 the majority said there was some information missing.

2.2.3 Describing High-Level Action of Java Methods

The goal is to define and describe segments of code at an abstract level (Sridhara et al. 2011a). Three types of code fragments are detected and summarized which are: sequence of single action, conditional statements, and tracble loop pattern.

Sequence as single action. A sequence of statements with similar actions is indicated by similar method calls. The identification process considers information from method names, parameters, or receiver objects. Same action fragments are built incrementally by determining if pairs of statements can be integrated. Several variations are used to identify sequences. One case is explained through the example in Figure 3.

1) For each method call, the verb, noun phrase(NP), and optional prepositional phrases (PP) are extracted. In Line 15 and Line 16, the verb phrases “add ended panel to content panel” and “add bid panel to content panel” are generated using SWUM (Hill et al. 2009).

2) The verbs in the verb phrases are the same (add)

3) The PPs are the same (to content panel)

4) The NPs have the same head word (panel)

5) Thus, lines 15 and 16 can be integrated. Similarly, lines 16 and 17 can be integrated.
Considering the previous example, the summary description starts with the common verb. The noun phrases are synthesized by using the plural form of the common head of the noun phrases. For the prepositional phrase, the common preposition is used. The high-level abstraction summary is “add panels to content panel”.

```
15 contentPane.add(endedPanel);
16 contentPane.add(bidPanel);
17 contentPane.add(endingPanel);
18 contentPane.add(buttonOK);
```

**Figure 3. An example of a sequence of method calls as single action**

**Abstracting Conditionals.** In conditional statements two locations are abstracted: the set of conditions and then-blocks. Integrating statements of “if . . . else if . . . else . . . “is managed by integrating the statements of the second if-else and then recursively integrating with the parent if. To build a fragment, each statement in the then-block is compared to

```
4 if (os.startwith("mac") {
5    app= new DefaultOSXApplication();
6 } else if (os.startwith("win") {
7    app= new DefaultMDIApplication();
8 } else {
9    app= new DefaultSDIApplication();
10 }
```

**Figure 4. An example of generating a synthesized description for a conditional expression**

Consider the if-expression presented in Figure 4. Clauses of the form “subject predicate object” are generated for the conditional expressions. The subject of the two calls in if and if-else is “os” and the predicate is the method name “start with”. Since the
subject and predicate of each clause is the same, then the template: “based on what subject predicate” is used. The synthesized abstraction of if-expression in Figure 4 is “based on what os name starts with”.

**Traceable Patterns in Loops.** Five common high-level actions performed by loops are considered here which are: max-min, count, contain, find, and copy. A template is developed to represent the general code structure of each loop pattern. For a given loop within a method, if the loop map to any of the algorithm templates, an abstract description is generated using the corresponding template for the pattern. The template form max-min loop is the following:

*Get <item> (in <collection>) with <adjective> <criteria>*

```java
9   for ( int x = 0 ; x < vAttacks.size( ) ; x++) {
10       WeaponAttackAction waa = vAttacks.elementAt( x ) ;
11       float fDanger = getExpectedDamage ( g , waa) ;
12       if ( fDanger > fHighest ) {
13           fHighest = fDanger ;
14           waaHighest = waa ;
15       }
16   }
17   return waaHighest ;
```

**Synthesized Description:** Get weapon attack action object (in vectorAttacks) with highest expected damage

**Figure 5. An example of generating a synthesized description for max loop**

For the max-min pattern, the extraction rules are explained through the example in Figure 5:
1) <item> and <collection> are extracted from the actual loop. For item, the class name of variables is used “weaponAttackAction object”. For collection, the variable name vAttacks is expanded to vectorAttacks.

2) To extract <adjective>, scan the individual words in the variable name in the actual loop that corresponds to maxItem in the algorithm template. The adjective is highest (line 12 and 13).

3) To extract <criteria>, scan the verb phrase generated for the statement that assigns the current value be checked (current is fDanger in line 11) in the algorithm template. <criteria> is expected damage.

According to a study on 15 programmers, the approach was judged to successfully present the high-level action of Java methods. In some cases, programmers do not agree with the propositions used.

2.2.4 Describing the Formal Parameters of Java Methods

A technique to automatically generate comments for parameters of a method is proposed by Sridhara et al (Sridhara et al. 2011b). The generated comments can be stand-alone parameter comments or can be integrated with the method summary generated from the previous approach (Sridhara et al. 2010). Method descriptions (Sridhara et al. 2010) are used as input to express the intent of the method. The generated summaries along with the s_units are used to find a context link between the summary and the parameter comments.

The five steps for generating parameter comments are:
1) Def-use information is used to identify all uses of the parameter.

2) The most important s_units of a parameter are identified by pruning irrelevant s_units to the parameter’s major role. For example, if a parameter is used to check against null or zero values are less likely to be executed. Consider an if-expression of the form if (x! = null) ... else ..., the then block is more likely to execute than the else-block.

3) Estimating closeness of the remaining uses to the method summary and the closest s_unit is selected. Eight heuristics are defined to identify linking context and one of them is presented in Table 5.

4) Integrating the parameter to the method summary using the linking context information.

5) Creating stand-alone parameter comment. Five parameter comments templates are created. An example of a parameter template is shown in table 4. “mb.add(m)” is identified as a major use for parameter “m”. It is also selected original to be the summary unit for the method. Because the parameter “m” appears in the s_unit of the method, the parameter is already connected to the summary of the method.

<table>
<thead>
<tr>
<th>S_unit</th>
<th>mb.add(m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The verb phrase</td>
<td>“add the given menu to menu bar”</td>
</tr>
<tr>
<td>The context Link</td>
<td>m correspond to the theme menu in the verb phrase.</td>
</tr>
<tr>
<td>Template</td>
<td>Which is &lt;past participle(action)&gt; &lt;secondary argument&gt;</td>
</tr>
<tr>
<td>Parameter comment</td>
<td>“@param m: which is added to menu bar”.</td>
</tr>
</tbody>
</table>

According to 6 experienced programmers, the generated parameter comments generally provide critical information about a parameter. They provide information about
the role of a parameter. However, the integration to the method summary is felt to be unnecessary.

2.2.5 Including Context of Methods Using Call Graph

McBurney and McMillan proposed generating documentation summaries for Java methods that include information about the context of methods (McBurney & McMillan 2014). The approach uses a call graph to find other methods that rely on the output of the method being summarized. The short description of each method was generated from the method signature using part-of-speech tags (Hill et al. 2009).

The steps of generating documentation summaries are:

1) The technique uses SWUM (Hill et al. 2009) (explained earlier) to specify the verb and the direct object of all methods in a software system.

2) The call graph of an entire software system is created. In such a graph, the nodes are methods, and the edges are call relationships among the methods.

3) PageRank that calculates importance of nodes based on the number of edges is used. Methods that are called many times or that are called by other important methods are ranked as more important than methods that are called rarely. When a method M being summarized, the summary of X methods that call M and have the highest ranks are included in the summary of M.

4) Finally, a natural language generation system is build. The system takes the lexicalization extracted from SWUM as well as the rank of all methods. The output is a natural language summary.
To generate a documentation summary for the method “read”, the verb, direct object and propositional phrase of the method are extracted. Moreover, the same step is applied on methods that call “read” and have highest ranks. In this example, four methods are selected. Table 1 presents the method signatures and the corresponding short summaries.

Table 6. An example of summaries generated from method signature

<table>
<thead>
<tr>
<th>Method to be summarized</th>
<th>Method signature</th>
<th>Short Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>StdXMLReader.read()</td>
<td></td>
<td>Reads a character</td>
</tr>
<tr>
<td>Methods that calls “read”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMLUnit.skipWhitespace()</td>
<td></td>
<td>Skips whitespace from the reader</td>
</tr>
<tr>
<td>XMLElement.addChild()</td>
<td></td>
<td>Adds a child element</td>
</tr>
<tr>
<td>StdXMLBuilder.startElement()</td>
<td></td>
<td>This method is called when a new XML element is encountered</td>
</tr>
<tr>
<td>StdXMLBuilder.addAttribute()</td>
<td></td>
<td>This method is called when a new attribute of an XML element is encountered.</td>
</tr>
</tbody>
</table>

Using the natural language generator system, all the generated short summaries for the 5 methods are connected. The summary of “StdXMLReader.read()” is: “This method reads a character. That character is used in methods that add child XML elements and attributes of XML elements. Called from method that skips whitespace”.

In addition to the context information, a use message is also included. In such a message, an example of code fragment that uses the method being summarized is included. For example, for the method “getResutl()”, the use message can be something like: “The method can be used in an iteration statement; for example: while (!this.reader.atEOF()) && (this.builder.getResult() == null)) { ".

To evaluate the approach, a case study using 12 participants was conducted. Participants were asked to compare summaries generated from this approach with summaries generated using Natural language approach (Sridhara et al. 2010). However,
for a single method, participants examine one type of summaries only. The results reveal that summaries generated from this approach are better in term of accuracy, conciseness, and including information about why the method exist. However, in some cases, participants consider summaries generated using this approach to contain a lot of unnecessary information.

2.2.6 Determining Loop Actions

Wang et al. propose a model that defines the high level action of loops (Wang, Pollock, & Vijay-Shanker 2015) and object-related statements (Wang, Pollock, & Vijay-Shanker 2017). At first, they mined open source Java systems to define action stereotypes for a specific type of loop that they refer as loop-if. Loop-if is a single loop that has only one if statement. Based on their mining process, 12 action stereotypes defined which are: count, determine, max/min, find, copy, ensure, compare, remove, get, add, Set_one, and Set_all. The proposed action identification model is then used to assign action stereotypes to loops. Their model considers five locations/items of loop-ifs that can determine the action of a loop. Figure 6 presents locations that are extracted and used to determine the action stereotypes.

1. If-condition. The if-condition refers to the conditional expression in the if-statement of the loop.
2. Loop exit statement such as a break or return. As they affect the number of iterations that are executed, they are considered in the analysis.
3. Ending statement of if block. It is the last statement inside a loop-if.
4. Loop control variable. The loop condition determines the maximum number of iterations that will be executed.

5. Result variable. The intent of the result variable is to capture the resulting value of the loop’s action (if one exists). They are defined as an assignment or method invocation found as the ending statement of if block.

![Figure 6. The five item/locations of a loop-if that used in the action identification model](image)

The proposed model includes eight features to determine the action stereotype of a loop. Table 7 presents features related to ending statements (F1 to F5) and features related to the if-condition (F6 to F8) and the values for each feature.

Consider the for-loop in Figure 7. The type of the ending statement (line 4) is an assignment and the type of loop exist statement (line 6) is a return. Finally, the usage of loop control variable(s) in ending statement (line 4) is indirectly through data flow (line 2). These clues map to “find” stereotype on the identification model as shown in Table 8. Thus, the action is identified as "find an element that satisfies some condition.” For the complete model, we direct the reader to (Wang et al. 2015).
Table 7. The eight Features and their Values

<table>
<thead>
<tr>
<th>Label</th>
<th>Feature</th>
<th>Possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Type of ending statement</td>
<td>0: none 1: assignment 2:increment 3:decrement 4:method invocation 5:object method invocation 6: boolean assignment</td>
</tr>
<tr>
<td>F2</td>
<td>Method name of ending statement method call</td>
<td>0: none 1: add 2: addX 3: put 4: setX 5: remove</td>
</tr>
<tr>
<td>F3</td>
<td>Elements in collection get updated</td>
<td>0: false 1: true</td>
</tr>
<tr>
<td>F4</td>
<td>Usage of loop control variable in ending statement</td>
<td>0: not used 1: directly used 2: used indirectly through data flow</td>
</tr>
<tr>
<td>F5</td>
<td>Type of loop exit statement</td>
<td>0: none 1: break 2: return 3: return boolean 4: return object 5: throw</td>
</tr>
<tr>
<td>F6</td>
<td>Multiple collections in if condition</td>
<td>0: false 1: true</td>
</tr>
<tr>
<td>F7</td>
<td>Result variable used in if condition</td>
<td>0: false 1: true</td>
</tr>
<tr>
<td>F8</td>
<td>Type of if condition</td>
<td>1: &gt;/&lt;/&gt;=/&lt;= 2: others</td>
</tr>
</tbody>
</table>

1 for(subunit s : subunits) {
2     if (s instanceof Department){
3         Department subDepartment = (Department)s;
4         Department result = subDepartment.findDepartment(id);
5         if (result != null){
6             return result;
7     }
8 }
9 }

Figure 7. An example of for-loop

Table 8. The value of the eight features of “find” stereotype

<table>
<thead>
<tr>
<th>Feature</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
<th>F8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1: assignment</td>
<td>0</td>
<td>0</td>
<td>2: used indirectly through data flow</td>
<td>2: return</td>
<td>0</td>
<td>0</td>
<td>2: others</td>
</tr>
</tbody>
</table>

To evaluate their model, 15 developers were asked to indicate how much do they agree that a loop code implements a given action. To reduce possible bias, developers were told the action descriptions were identified from different systems, and that the goal was to
know which system is better. 60 loops were randomly selected, 5 for each action stereotypes. Each participant evaluated 12 loops. In 93.9% (169 out of 180) of responses, judges strongly agree or agree that the identified actions represent the high level actions of the loops, within which 61.9% correspond to strong agreement. Therefore, it was concluded that the identification model accurately identifies the high-level action of loop-if.

2.2.7 Text Retrieval (TR)

This section describes the usage of text retrieval approach to generate term-based summaries by selecting the most relevant terms of a class or method (Haiduc et al. 2010) (Eddy et al. 2013) (Rodeghero et al. 2014).

2.2.7.1 VSM and LSI

Haiduc et al. investigated the suitability of several text summarization techniques (TR) to automatically generate term-based summaries for methods and classes (Haiduc et al. 2010). This was further extended by Eddy et al. using a new TR technique named Hierarchical PAM (Eddy et al. 2013).

Three techniques were applied to generate summaries which are: Vector Space Model (VSM), Latent Semantic Indexing (LSI), and lead summary. VSM and LSI are statistical based text retrieval techniques that have been successfully applied for text summarization. Lead summaries is based on the position of terms. Lead summaries are based on the idea that the first terms that appear in a document are the most relevant to that document.
Generating term-based summaries using TR follow two steps: preprocessing and summary generation.

**Preprocessing.** Preprocessing is the process of extracting the text from the source code of the system and convert it into a corpus. First is to convert source code into a document collection. Documents reflect source code entities such as methods and classes. In each case, the identifiers and comments in the source code entities are extracted. Second, terms which do not carry specific meaning are removed using a stop-words list. Examples of stop words 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.). Finally, identifiers (e.g. method name) are divided using camel case splitting (see section 2.2.1).

**Generating summaries.** To determine the most relevant terms for documents in the corpus VSM and LSI are used. First, terms and documents in the corpus are presented in a matrix. 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 document that derived from the frequency of the term in the document. The global weight refers to the relationship of a term with the entire corpus that derived from the distribution of the term in the corpus.

When VSM is used, 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. On the
other hand, LSI projects the original term by document matrix to a smaller one which approximates it, corresponding to the highest Eigen values in the original matrix. LSI allows the representation of terms and documents in the same coordinates. To determine what to include the summary, 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 highest K terms are included in the summary. Different types of summaries are generated. Table 9 shows the various attributes of these summaries.

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of terms</td>
<td>5-term and 10-term</td>
</tr>
<tr>
<td>TR weighting schemes</td>
<td>log, tf-idf, and binary-entropy.</td>
</tr>
<tr>
<td>Identifiers state</td>
<td>splitting identifiers or not splitting the identifiers</td>
</tr>
</tbody>
</table>

To evaluate TR techniques, a user study that involves four developers was performed. Developers are presented with summaries for methods and classes and asked to evaluate the automated summaries. This work was replicated and expanded by Eddy et al. by including higher number of experienced participants and investigating a new topic model technique, Hierarchical PAM (Eddy et al. 2013). The pachinko allocation model (PAM), a family of generative topic models that build on Latent Dirichlet allocation (LDA), connects words and topics with a directed acyclic graph (DAG), where interior nodes represent topics and leaves represent words. Hierarchical PAM (hPAM) allows higher-level topics to share lower-level topics. Therefore, a “methodological” topic (e.g., a topic that includes “points” and “players”) can be shared among distinct topics (e.g., five topics that represent different sports). To generate summaries, the most likely subtopic for the
document (method or class) is identified and the top K terms for that subtopic are included in the summaries.

Similar conclusions and findings derived from the two user studies (Eddy et al. 2013; Haiduc et al. 2010). It is found that leading summaries outperforms all other individual techniques and combining Lead+VSM provides the best result. With the same technique and the same parameters, the 10 terms summaries are always ranked higher than the 5 terms summaries. One reason for the fact that 10-term summaries get higher scores is that they contain more relevant terms than the 5-term ones as it contains the same 5 terms plus more.

In addition, it is observed that the versions containing full identifiers obtained the highest score in 60% of the cases (Eddy et al. 2013). These findings were confirmed also by the developers. They reported that they preferred the summaries containing the full identifiers. However, the differences between the performances of the all techniques using different weighting schemes are not statistically significant.

The agreement between developers is higher when rating bad summaries than when rating good summaries (Eddy et al. 2013). Overall, the ratings assigned by the 14 subjects in (Eddy et al. 2013) were lower than the ratings assigned by the 4 subjects in the original study (Haiduc et al. 2010). A possible explanation is that 10 of the 14 subjects in Eddy et al.’s study reported having work experience in the software industry and their experience provided them with better insight on what should be included in the summary.

2.2.7.2 Improving VSM Approach by Eye Tracking Study

Rodeghero et al. conducted an eye-tracking study to determine the statements and terms that programmers view as important when they summarize a method (Rodeghero et
al. 2014). 10 experienced programmers are presented with Java methods and their task is to write their own summaries. Based on the computed gaze and fixation times, it is concluded that programmers consider method signatures as the most important section of code followed by invocation terms then control flow terms. It is also found that the list of relevant terms generated by (TR) (Eddy et al. 2013; Haiduc et al. 2010) approximated what the user focus on when writing their summaries.

Using the previous result, a modified version of Vector Space Mode (VSM) is implemented. The frequency of terms in VSM is replaced by the term positions. Therefore, terms located in method signature have weight higher than method invocation and method invocation has higher weight than control flow statements. An example of the assigned weight is shown in Table 10.

<table>
<thead>
<tr>
<th></th>
<th>Method signature</th>
<th>Method call</th>
<th>Control flow</th>
<th>All Other Areas</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original VSM</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Modified VSM</td>
<td>4.2</td>
<td>1.4</td>
<td>0.6</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Nine participants are provided with 24 methods and their corresponding lists of terms and asked to select and order the five most relevant terms. The overlap between the modified VSM and the terms selected by participates’ is .76 compared to .67 provided by the original VSM. In addition, the similarity between the order modified VSM list and the order provided by participants is .11 better than the original VSM list. These improvements indicate that not only did the programmers focus on terms from some sections of source
code more than others during summarization, but they also tended to use those keywords in their own summaries.

2.2.8 Mining Human Descriptions

Wong et al. proposes an approach to generate comments automatically by mining Q&A sites for code description such as StackOverflow (Wong et al. 2013). The first step is to build a code-description database that maps questions and answers with the associated code fragments. Answers that receive high number of votes are selected. Descriptions of Java and Android projects are refined and included in the database along with the corresponding code fragments. Some of the natural language techniques used to refine the extracted descriptions are discussed here.

**Main Sub-Tree Extraction.** Sentences that are in a question form (e.g., “How to...”) or contain personal pronouns (e.g., “you”) are not suitable to be part of comments. Stanford CoreNLP is used to generate parse tree after assign each word a part of speech tag. Then the main sub-tree is extracted: noun phrase followed by a verb phrase or the reverse. Finally, all extracted sub-tree are merged to one tree and converted to a sentence.

**Subordinate Clause Removal.** A sentence may contain more than one clause connected by a coordinating conjunction (CC) such as “and”, “but”, “or”, etc. Their tool removes the subordinate clause after the CC word “but” and “yet” as they typically signify a negative meaning. Table 11 shows an example of the two refinement steps.

To identify similar code fragments, they used clone detection between source code and code fragment that is contained in the discussion, and then automatically generate
comments. 15 graduate students evaluated 102 generated comments. The responses show some positive result although there are a significant number of disagreements. They concluded that the reasons of the disagreements due to incorrect or long and wordy descriptions. Furthermore, in many cases, code is easy-to-understand and no comment is needed to help comprehension, or the comment is too trivial.

<table>
<thead>
<tr>
<th>The original description</th>
<th>How do I read in a file with buffer reader but skip comments with java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extracting the main sub-Tree</td>
<td>read in a file with buffer reader but skip comments with java</td>
</tr>
<tr>
<td>Subordinate Clause Removal</td>
<td>read in a file with buffer reader</td>
</tr>
</tbody>
</table>

2.2.9 Static Analysis and NLP

Moreno et al. use method stereotypes and class stereotypes (Natalia Dragan et al. 2010) (L. Moreno & Marcus 2012) to generate natural-language summaries for Java classes (L. Moreno et al. 2013) (L. Moreno, Marcus, Pollock, & Vijay-Shanker 2013). The approach uses method and class stereotypes to filter out irrelevant methods to the class stereotype. Then, from the signatures’ of the selected methods, behavior phrases are generated using the approach proposed by Sridhara et al (Sridhara et al. 2010). For each class stereotype, several templates are created. The two steps of generating summaries for classes are: content section and description generation. The content selection is to determine methods to be included through a filtering process. Methods whose stereotypes are not relevant to the class stereotype are eliminated. Heuristic for each class stereotype is applied. For example, accessors are the main methods in a Data Provider class, so every
method that is not accessor is removed. Then, three type of descriptions are generated which are: general description, stereotype description, and behavior description

**General Description.** It provides a generic idea of the type of objects represented by the class. It uses the names of the super classes and interfaces of classes (if any), as qualifiers of the represented object. Considering the possible variations in class declarations and the class stereotype taxonomy, 22 different templates are defined to provide the general description.

**Stereotype Description.** The stereotype definition is used to describe a class. When it is possible, the definitions are enriched with specific information such as the represented object or the existence of certain kind of methods. 40 different templates for the second part of the summaries are created. For example, when the class to summarize is a Data Provider, the generated text uses the template shown below:

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

**Behavior Description.** For describing the behavior provided by the class, the relevant methods selected during the filtering process are summarized and included. The behavior description is divided into three blocks. The first two blocks represent the accessor and mutator methods 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. Method signatures are used to create natural language description of each method. To create a readable description, phrase generation tools (Sridhara et al. 2010) is utilized to obtain natural-language fragments from method signatures. Two examples on how summaries for mutator methods are generated shown below.
Mutators method changes the object states, therefore, the fragment that indicates the property which is being modified by the method is added. The fields that are being modified by the mutator methods is extracted from the signature of the methods as shown in Table 12. If the signature of the method does not provide the properties that are being modified, the method is removed to the third block and the class name is used as the theme in the generated phrase as shown in Table 13. The final adjustment in this block is the transformation of the action of the methods into its gerund form.

<table>
<thead>
<tr>
<th>Method signature</th>
<th>void setContext(Context c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stereotype</td>
<td>Set</td>
</tr>
<tr>
<td>Generated text</td>
<td>It allows managing:</td>
</tr>
<tr>
<td></td>
<td>- context.</td>
</tr>
</tbody>
</table>

Table 12. The summary phrase of a set method

<table>
<thead>
<tr>
<th>Class declaration</th>
<th>public class CdRipper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method signature</td>
<td>void stop()</td>
</tr>
<tr>
<td>Stereotype</td>
<td>Command (mutator method)</td>
</tr>
<tr>
<td>Field accessed</td>
<td>Tag tag</td>
</tr>
<tr>
<td>Generated text</td>
<td>It also allows:</td>
</tr>
<tr>
<td></td>
<td>- stopping cd ripper.</td>
</tr>
</tbody>
</table>

Table 13. The summary phrase of a command method

The final summary is composed of the three parts grouped together. To evaluate their approach, 22 graduate students in computer science were asked to judge the automated summaries for 25 Java classes. It was concluded that about 30% of the cases, the summary missed very important information. 11 out of 15 of the classes have no unnecessary information. It is found that most of unnecessary information corresponds to the
descriptions of the stereotypes. Some unnecessary information was caused by the poor quality of some of the identifiers. Finally, the majority agreed that the descriptions were easy to read and understand.

While we also use method stereotypes as applied in (L. Moreno et al. 2013), the main contribution of our work is to generate documentation summaries for methods, not classes. Thus, we generate a much finer granularity of documentation then that for a class. Our technique uses method stereotypes to generate a standard summary for each method. Chapter 3 demonstrates our proposed technique.

2.3 Source-code Comprehension

Program understanding/comprehension is an essential task during software maintenance for enabling successful evolution of programs or systems (Von Mayrhauser & Vans 1995). For several years, researchers have attempted to understand how programmers comprehend programs during software maintenance (Brooks 1983; Burkhardt, Détienne, & Wiedenbeck 1997; Letovsky 1986; Pennington 1987; Shneiderman & Mayer 1978; Soloway & Ehrlich 1984; Storey 2005; Von Mayrhauser & Vans 1995, 1997). It is concluded that program comprehension process uses existing knowledge to obtain new knowledge and build a mental model of the program that is under consideration (Von Mayrhauser & Vans 1995).

A mental model describes a developer's mental or conceptual representation of the source code to be understood (Storey 2005; Von Mayrhauser & Vans 1995). Code cognition models examine how programmers understand source code by describing the
cognitive processes and temporary information structures in the programmer’s head (Von Mayrhauser & Vans 1995).

One of the earliest cognition model is Shneiderman and Mayer model that explains two type of memories (Shneiderman & Mayer 1978). First, is short-term memory that records the internal semantic representation of the program via chunking mechanism. The second type of memory is the long-term memory that contains the knowledge base with semantic and syntactic knowledge. The knowledge base in the long memory helps building the short memory during comprehension process (Shneiderman & Mayer 1978).

Brooks hypothesizes that program comprehension is reconstruction knowledge about the domain of the program (Brooks 1983). In such approach programmers understand a completed program in a top-down manner. The process starts with a hypothesis about the general goal of the program. This initial hypothesis is then confirmed or refined by forming additional hypotheses.

On the other hand, Soloway and Ehrlich observe that top-down understanding is used when a developer is familiar with code (Soloway & Ehrlich 1984). They observe that expert programmers use beacons (familiar features such as sorting) and rules of programming discourse to decompose goals and plans into lower-level plans. They note that delocalized plans complicate program comprehension.

Pennington model observes that programmers first develop a control-flow abstraction of the program which captures the sequence of operations in the program (Pennington 1987). This model is referred to as the program model. Then, the situation model, bottom-
up, uses knowledge about data-flow abstractions and functional abstractions. The situation model is complete once the program goal is reached.

Letovsky proposes the knowledge-base model that depends on preprogramming expertise, problem domain, knowledge, rules of discourse plans and goals (Letovsky 1986). Finally, the latest cognition is the integrated metamodel that is found by (Von Mayrhauser & Vans 1995). The metamodel is built based on the previous models. The model consists of four major components. The first three components deal with creating mental representations at various levels of abstraction and the fourth component describes the knowledge base needed to create the mental representations.

The top-down model is usually applied as-needed (i.e., the programming language or code is familiar). It integrates domain knowledge as a starting point for formulating hypotheses. The program model may be invoked when the code and application is completely unfamiliar. This is done in examining a control-flow of a program. The situation model describes data-flow and functional abstractions in the program. It may be developed after a partial program model is formed using systematic or opportunistic strategies. The knowledge base consists of information needed to build these three cognitive models. It represents the programmer's current knowledge and is used to store new knowledge (Von Mayrhauser & Vans 1995).

A number of user studies are conducted to examine what types of mental models formed by both novice and expert programmers during comprehension (Burkhardt et al. 1997; Von Mayrhauser & Vans 1997).
Von Mayrhauser and Vans perform a user study included four developers doing maintenance task (e.g., fixing a bug) (Von Mayrhauser & Vans 1997). They execute a thinking aloud experiment; and they audio and video tape developers. They conclude that a developer switches between top-down, bottom-up, and program model in order to build the mental model. With little experience on the domain, the comprehension is done at lower level by reading the source code closely (bottom-up) to build a higher-level abstraction. Furthermore, programmers with domain expertise look for specific information such as statement execution order and definition and uses of variables.

Burkhardt et al. conduct a user study to evaluate a cognitive model of object-oriented (OO) program understanding (Burkhardt et al. 1997). In particular, they want to draw a distinction between the program model and the situation model in OO program understanding. 30 experts and 21 novices are asked to perform one of two tasks: designing a variation of the library problem or writing comment of existing the code. Then, the metal model judged based on a post-questionnaire. Burkhardt et al. conclude that the situation model is more fully developed than the program model, even in an early phase of comprehension. (Burkhardt et al. 1997) This contrasts with the results of Pennington for procedural programmers (Pennington 1987). She showed that the program model developed more fully in the early stages of comprehension, whereas the situation model emerged later. They explain the difference as the OO paradigm, with its emphasis on objects and relationships of objects, may facilitate the construction of a situation model earlier in program comprehension.
Here, we take the analysis of the comprehension process to a new level by studying if the reading behavior captured by the eye-tracking data can predict the mental model of developer during a comprehension task (i.e., summarizing a method). This is later discussed in section 7.3.

2.4 Eye Tracking in Software Engineering

Eye-tracking technology has been recently used in software engineering (Sharafi, Soh, & Guéhéneuc 2015). It has been used to study how programmers read (Bednarik & Tukiainen 2006, 2008; Busjahn et al. 2015; Crosby & Stelovsky 1990; Rodeghero & McMillan 2015), review (Sharif, Falcone, & Maletic 2012; Uwano, Nakamura, Monden, & Matsumoto 2006), and summarize (Rodeghero, Liu, McBurney, & McMillan 2015; Rodeghero et al. 2014) source code. It was also used to evaluate different UML diagram layout (Sharif 2011; Sharif & Maletic 2010a) and identifier style (Binkley et al. 2013; Sharif & Maletic 2010b). The following briefly demonstrates some findings of these studies.

2.4.1 Eye Tracking in Code Comprehension

Crosby et al. (Crosby & Stelovsky 1990) conducted an eye-tracking study of 19 subjects (10 low-experience subjects and 9 high-experience subjects) reading binary-search algorithm written in pascal. It was concluded that the process of reading source code is different from the process of reading natural text. All subjects needed numerous fixations in most areas of the algorithm than did subjects in studies using simple text (Crosby & Stelovsky 1990).
The study also examined the amount of time spent in five areas of interest which are comments, comparisons, complex statements, simple assignments, or keywords. The comment areas received the most attention from both high-experience and low-experience groups. However, on the average, subjects in the low-experience group dedicated more attention to comments (44% of the time) than the high-experience group (35% of the time). Keywords and simple statements did not attract much attention from both groups. Moreover, the low-experience group examined the comparison areas significantly longer than did the high-experience group. Finally, complex statements, that provide the most information about algorithm details, had the greatest and most significant difference in viewing time between the two experience groups. In this study, the most complex statement is “found:= left = right +2” which can be translated as “if left = right + 2 then found := true else found := false”. The high-experience group had 68% more fixations to these areas than did the low-experience group. Therefore, it was concluded that with increasing experience, subjects learn to focus on the key areas of code as suggested by research in program comprehension (Crosby & Stelovsky 1990).

Uwano et al. performed an eye-tracking study on how developers review code to find defects (Uwano et al. 2006). The study included 5 subjects and six Java programs. It was observed that subjects tend to first read through the code, and then focus on some portions. About 73% of the code lines were read in the first 30% of the review time. Furthermore, it was concluded that the longer a reviewer read through the code, the more efficiently the reviewer could find the defect. This correlation was later confirmed by Sharif et al. (Sharif et al. 2012) stating that the scan time plays an important role in defect detection time and
visual effort required to review source code. Moreover, experts tend to focus on fixations more, while novices may watch lines more broadly (Sharif et al. 2012). This result was later statistically reproduced by Busjahn et al. (Busjahn et al. 2015). They found that experts read code less linearly than novices.

Bednarik and Tukiainen conducted an eye-tracking study of 18 Novice and intermediate programmers (Bednarik & Tukiainen 2006). Programmers were asked to use a program visualization tool to aid their comprehension. Three small Java systems were used: factorial computation, recursive binary search and naive string matching. The names of the methods and variables were changed to make recognition of a program difficult. They found that more experience programmers read the code of the program first before they initiated the visualization. Then, they use the visualization to confirm their original understanding of the code. On the other hand, less experienced programmers animated the program several times to visually understand the execution then, read the code. Furthermore, it was observed that low-experience programmers repeatedly fixated on the same sections, while experienced programmers target the output of the code, such as evaluation expressions (Bednarik & Tukiainen 2006, 2008).

Kevic et al. conducted an eye tracking study on three bug fix tasks. They found that developers focus on small parts of methods that are often related to data flow. When it comes to switches between methods, they found that developers rarely follow call graph links and mostly switch to the elements in close proximity of the method within the class (Kevic et al. 2015).
Eye tracking studies have also been done to gauge the effectiveness of identifier style (Binkley et al. 2013; Sharif & Maletic 2010b). Results indicate a significant improvement in time and lower visual effort with the underscore style (Sharif & Maletic 2010b). However, expert programmers appear to not be impacted greatly by Style (Binkley et al. 2013). Finally, Sharif et al. studied the impact of class diagram layout on comprehension (Sharif 2011; Sharif & Maletic 2010a). It was concluded that organizing a class diagram based on architectural importance increases system comprehensibility (Sharif 2011).

2.4.2 Eye Tracking in Code Summarization

The previous literature suggested that the words/terms that programmers spend more time reading are more important for comprehension (Bednarik & Tukiainen 2006; Crosby & Stelovsky 1990). Taking this conclusion, Rodeghero et al. conducted an eye-tracking study to determine the statements and terms that programmers view as important when they summarize a method (Rodeghero et al. 2014). Based on the computed gaze and fixation times, it is concluded that programmers consider method signatures as the most important section of code followed by invocation terms then control flow terms. Term-based summarization was enhanced when the locations of terms is considered in VSM technique (see section 2.2.7.2).

The following explains in details how the eye-movements of programmers are investigated by Rodeghero et al. (Rodeghero et al. 2014). The collected data is analyzed at two levels of granularity which are: statement level (Rodeghero et al. 2014) and term level (Rodeghero et al. 2015). At statement level, a method signature, method invocations, and control flow are compared to other part of the method. At term level, the values of gaze
time of terms in all methods are analyzed and compared regardless to their locations in the methods. The following explains their procedure and findings in details.

2.4.2.1 At Statement Level

Rodeghero et al. found that the programmers spent more gaze time, and fixated more often on the method signatures than the method bodies. However, they did not find a statistical difference in the regression time, indicating that the programmers did not re-read the signatures more than the methods' body keywords. Their conclusions were derived by comparing the adjusted gaze time, fixation, and regression count for terms in the signatures to the keywords in the method bodies as the following.

The gaze time percentage is the amount of time that the programmer spent reading the signature keywords for a given method. For example, a programmer spent 30 seconds reading a method signature and 70 seconds to read a method body. Then, the programmer spent 30% of his or her time reading the method's signature. Then, the gaze time percentage is adjusted based on the size of the signature. For example, if a programmer spent 30% of his or her time reading a method's signature, and the signature contains 3 of the 20 keywords (15%) in the method, then the adjusted gaze time metric is $30/15 = 2$.

For the fixation and regression, the same approach was used. Finally, they test whether the difference between the adjusted (gaze time / fixation / regression) metric for method signatures and method bodies is statistically-significant using Wilcoxon test (Wolfe & Hollander).

Rodeghero et al. concluded that on average, programmers spent 31% of their time reading control flow terms, even though these terms averaged 37% of the terms in the
methods. A “control flow” term included any term inside of a control flow statement. For example, for the line if (area < maxArea), the control flow terms are “area” and “maxArea” (Haiduc et al.). Their interpretation to this observation is that programmers avoid reading code details whenever possible (de Souza, Anquetil, & Oliveira 2005). Another possible reasoning is that control flow has a well-formed structure that do not require time to comprehend.

Terms from a method invocation include the invoked method’s name and augments. For example, for the line double ca = getArea(circle, 3), the invocation keywords is “getarea” and “circle,” but not “double” or “ca.” Rodeghero et al. found no evidence that programmers read keywords from method invocations more than keywords from other parts of the methods.

2.4.2.2 At Term Level

Rodeghero et al. investigated the correlation between the time that programmers spend reading a term and the likelihood that they will use the term in a summary. The phrase “longer-viewed” is used to define terms in source code that programmers spend the most time reading (gaze time). Using a K-medoids clustering algorithm, they created two clusters based on the gaze time data: one cluster for longer-viewed words and one cluster for all others. K-medoids was used because it is less sensitive to outliers than K-means or other strategies (Kaufman & Rousseeuw 1990; Park & Jun 2009). Based on the cluster split point, terms that read longer than 3000 milliseconds are considered longer-viewed.

They found there are fewer “longer-viewed” terms contained in method summaries compared to all other terms. On average, the percentage of “longer-viewed” terms used in
method summaries was 12%, while the percentage of all source code terms used in method summaries was 16% (Rodeghero et al. 2015). Furthermore, terms programmers used in their method summaries generally have higher tf/idf scores than terms with high gaze times.

Terms that were longer-viewed and high VSM tf/idf score but not included programmers’ summaries were examined. They found statistically significant evidence that long terms (large number of characters), that were also “longer-viewed”, are used more often in the source code than they are in programmer summaries. This indicates that mainly programmers do not use long identifiers in their summaries as they appear in the code (Rodeghero et al. 2015).

2.4.2.3 Reading Pattern

Eye movement patterns are the order of words or sections that people follow when they read. These patterns are usually researched with respect to natural language text. Previous literature draw some insight on studying these patterns when reading natural language text such as: ad placement in phone books (Lohse & Peltz 1996), web site design (Group 2013), newspaper layout (Lohse & Peltz 1996), etc.

Furthermore, programmers’ reading patterns are analyzed when applied to source code (Rodeghero & McMillan 2015) (Busjahn et al. 2015). A word is considered in a programmer reading pattern if it was fixated on for 300ms or longer. First, they performed a qualitative study of the reading patterns by manually analyzing results of five programmers and four methods per programmer. The conclusions derived from the
qualitative study shaped the next study, the qualitative study. In the quantitative study, the eye movement patterns from eye tracking data were calculated and analyzed.

All five programmers tended to read code from left to right; however, they did not always read from top to bottom. Starting at the top or the bottom seems to depend of programmers’ preferences. Among the five participants, both of the preferences occurred with about the same frequency. When reading from top, one of the first areas read is the method signature, which contains the method name, the return type, and the input parameters which provides a significant amount of clues for determining the overall purpose. On the other hand, the end of the method can significantly indicate the overall purpose and sometimes provides more important information than the signature (Rodeghero & McMillan 2015). On average, programmers read from top-to-bottom about 49% of the time. Another observation is that programmers tend to skim code rather than read it in-depth. On average, programmers thoroughly read about 10% of each method.

On average, programmers read in sections about 25% of the time. Using statistical analysis, they found that programmers prefer to read code by jumping between sections rather than reading code one section at a time. Finally, it was concluded that all of the programmers followed very similar patterns in reading overall. This conclusion is drawn by comparing each programmer pattern of a method. Their quantitative metrics considered three factors: (left-to-right vs right-to-left and top-to-bottom vs bottom-to-top, skimming versus through, and disordering vs sectionally).
CHAPTER 3

METHOD DOCUMENTATION USING STREOTYPES

The following section demonstrates our approach of generating natural language
documentation summaries of OO methods. The design of methods, stereotypes, are used
to direct the content sections of the automated summaries. Then, using predefined
templates for each method stereotype, the documentation summaries of methods are
generated. The approach and the templates used are explained in here.

3.1 The Approach

Our approach takes a C++ system as input and for each method automatically
generates a documentation summary that is placed as a comment block above the method.
An example of the automatically generated documentation for method is presented in
Figure 8. The member function calcWidthParm() is in the class BinAxsLinear from the
open-source system HippoDraw. The documentation starts off with a short description,
(e.g., the first three lines in Figure 8). This emphasizes the computed value along with the
data members and parameters used in the computation. Following that is a list of calls
made from this method along with the corresponding stereotypes of the calls.

To construct the documentation, two issues must be addressed. The first is
determining what information to include in the documentation. The next is to present this
information effectively. Previous studies on source-code summarization/documentation
have investigated the former issue in depth and these results are used as a set of guidelines
for building our tool. First, the method documentation should include at least a verb (i.e., describes the action performed by the method) and an object (i.e., describes on which the action is performed) (Eddy et al. 2013; Haiduc et al. 2010; McBurney & McMillan 2014). Second, the name of the method is considered to be the most useful part for use in the documentation (Laura Moreno & Aponte 2012; Rodeghero et al. 2014). Moreover, full identifiers are more preferable than split identifiers (Eddy et al. 2013; Haiduc et al. 2010) as it assists in mapping the documentation to the source code. Calls within the method often produce side effects and the recommendation is to include them in the documentation (Laura Moreno & Aponte 2012; Rodeghero et al. 2014). Finally, local variables seem to not provide very useful information and thus are not included (Laura Moreno & Aponte 2012).

```cpp
/**
calcWidthParm is a property that
 returns a computed value new_width
 based on data member: m_range and parameter: number.
 Calls to-
 high() get
 low() get
*/
double BinAxesLinear::calcWidthParm (int number) const {
 double new_width = (m_range.high() - m_range.low())
 / (static_cast < double > (number));
 return new_width;
}
```

Figure 8. An example of a property method and its generated documentation summary. The documentation is inserted into the code as a method comment

However, capturing the method’s responsibilities requires further analysis. We found that the stereotype of a method can be used as a basis for selecting the content and format of the automatically generated documentation. In other words, the stereotype of the method is used to locate and determine the lines of code that reflect the main actions of a method.
For example, the main action of a property method is to return a computed value whereas the main action for a command method is to modify more than one data member.

With regards to effective presentation of the information, summaries/documentation must be structured in a manner that is easily read, and mapped, to the source code. Therefore, the documentation follows a predefined structure that is composed of four parts:
1) A short (one or two line) description that expresses the main responsibilities of the method; 2) External objects used by the method; 3) Data members/parameters written; and 4) List of method calls and their stereotypes.

**Figure 9. The overview of systems used to facilitate generating the documentation summaries**

**Figure 9** presents an overview of the existing systems used to facilitate fact extraction and summary generation. The main steps of our approach. First is to transfer the source code to XML format using srcML. Second, the method stereotypes are identified and added to
the source code using StereoCode. Finally, our approach takes the stereotyped source code as input to generate the automatic documentation.

Our tool, SumCode, takes an XML file of the entire project where all methods are tagged with their stereotypes as demonstrated in section 2.1. An example is shown in Figure 1 after the conversion back to source. The second step is to parse the XML file to build a method/stereotype database.

The database includes information about each class to speed up the stereotype extraction of calls during documentation generation. For each class, the base class(es), data members and method names along with their stereotype tags (block comments that start with “/** @stereotype”) are extracted. The list of data members is important in order to locate a method in the database. If a method is invoked on the data member (i.e., the data member is an object), then the type of the data member is used to identify the class owner of that method. In addition, for each method, the number of parameters and their types are also extracted to uniquely identify a method in the database and account for overloaded methods. Finally, additional information about the type of methods (static, virtual, etc.) is also extracted to better evaluate method calls.

During documentation generation, the stereotypes of calls are determined from the database, which provides additional information about the actions taken upon the call. Although the method name is an important way to indicate the role of a method, names can also be misleading (Ko et al. 2006). For example, in the HippoDraw system the name of a call to the method setRange() indicates that the method modifies one variable/data member (Range). However, in this case the method actually modifies Range and two other data
members, named $m\_num\_bins$ and $m\_width$. Including the stereotype of the call (in this case, $command$) alerts the developers of the possibility of changing other data members. Another example is a non-void method call. If the call is stereotyped as a $property$ method, programmers can conclude that the main action of the call is to return a computed value. On the other hand, stereotyping the call as a $non\_void\_command$ indicates that the call performs two actions: modifying multiple data members and returning a value. Therefore, including the stereotype of calls can reduce the need to examine the definition of these calls, which can be time-consuming, especially for calls to methods in different classes. Over a large system, this can improve the overall maintenance process.

The heuristics and templates used to generate the automated documentation summaries are explained in the following section.

### 3.2 Templates

The template for the four sections of the documentation is shown in Figure 10. If a section is not applicable, it is omitted. For example, the automatically generated documentation of the $property$ method in Figure 8 omits lines 2 and 3. The individual templates for each section of the documentation are explained below.

#### 3.2.1 Short Description

The first section of the documentation (line 1 in Figure 10) is an English description that explains the main responsibility of the method in terms of the stereotype(s). The template for the short description is composed of two phrases. The first phrase is standardized for all stereotypes while the second phrase is customized for each specific
stereotype. The first phrase of the template for the short description presents the method stereotype in the following form:

```markdown
<method> is a <stereotype>
[that collaborates with <object>]
```

The second line in the above template is optional, and is used if the secondary stereotype of the method is *collaborator* or *controller*. For example, the documentation of the method `validate()` that works with one external object `Range` gives us the template:

```markdown
validate is a void-accessor that
collaborates with Range
```

We avoid long lists of objects. Therefore, if the method is collaborating with more than one object, then the number of objects is included in the short description and the list of names is moved to the second part of the documentation (line 2 Figure 10).

The second phrase of the short description explains the main actions of the method. The particular template used depends on the stereotype of the method.

```markdown
/**
1 <Short Description>
2 Collaborates with- <obj1>, ... <objm>
3 Data members or parameters modified- <name1>, ... <namen>
4 Calls to-  <call1>        <stereotype1>
            <calln>        <stereotype1>
*/
```

Figure 10. The template of the four sections of the documentation summary

### 3.2.2 Templates for Accessors

*Accessors* are const methods that do not change the object state. The main responsibility of an *accessor* is to return: 1) a data member (get), 2) information about data members (*property and predicate*), or 3) information through parameters (*void-accessor*).
Therefore, data members or local variables used in the return expression or as reference parameters are included in the short description. To cover all the variations, we have developed 27 template phrases for the accessors’ short descriptions. The main cases are now described. A get method returns a data member only, and gives us the template:

\[
\text{that returns one data member: } \langle\text{data-member}\rangle
\]

A property returns information about data members. Thus, the return expression is the essential line that presents the main role of the method. The return expression can be a single variable, a computation, or a method call, each of which is handled differently. If the return expression is a call, then the call and its stereotype is included in the short summary as:

\[
\text{that delegates to } \langle\text{stereotype}\rangle: \langle\text{call}\rangle
\]

If the return expression is a single variable (Figure 8) or a value, the following template is used:

\[
\text{that returns a computed value } \langle\text{value}\rangle
\]

If the return expression is a computation (e.g., return \(x + y\)), the above is used and “\(<\text{value}\>\)” is omitted. Then, for the three cases explained above, data members, parameters, and calls used to compute the final value are listed using the “based on” template as given below:

\[
\text{based on data member(s): } \langle\text{data-mem}_1\rangle, \ldots, \langle\text{data-mem}_n\rangle
\]
\[
\text{and parameter(s): } \langle\text{parameter}_1\rangle, \ldots, \langle\text{parameter}_m\rangle
\]
\[
\text{and value(s) computed from } \langle\text{call}_1\rangle, \ldots, \langle\text{call}_k\rangle
\]
Finally, if the method has more than one return expression, the list of the returned values is displayed and separated with “or” as shown in Figure 11. Typically, some logic controls which return statement is executed. So the phrase “depending on” is then used to specify the properties controlling the returned values as shown below:

```cpp
returns <value>, …, or <value>
 depending on data member(s): <data-mem>, …, <data-mem>
 and parameter(s): <parameter>, …, <parameter>
 and value(s) computed from <call>, …, <call>
```

/* getLow is a property that
returns a computed from
binner_axisX->axisGetLow() or binner_axisY->axisGetLow() depending on axis
*/
double Bins2DBase::getLow ( Axes::Type axis ) const
{
    if ( axis == Axes::X ) return binner_axisX->axisGetLow();
    if ( axis == Axes::Y ) return binner_axisY->axisGetLow();
    assert ( false );
    return 0.0;
}

Figure 11. The documentation summary of a property method with multiple returns.

/* isValidLabel is a predicate that
returns a computed value: yes
based on first and a value computed from: m_labels.end()
*/
bool DataSource::isValidLabel ( const std::string & label ) const
{
    vector< string >::const_iterator first
        = find ( m_labels.begin(), m_labels.end(), label );
    bool yes = first != m_labels.end();
    return yes;
}

Figure 12. The documentation summary of a predicate method.

A predicate returns a Boolean value by an expression that includes a comparison to a data member. Therefore, the main responsibility of a predicate method is also located in
its return expression. At this point, we apply property’s templates to predicate methods. An example of a predicate method is shown in Figure 12.

A void-accessor returns information through a parameter. Therefore, the key responsibility of a void-accessor method is to modify one or more parameters. To conclude that a parameter $p$ is modified by a method $M$, two conditions are evaluated. First, the parameter $p$ is passed as a non-const reference or as a pointer. Second, the parameter $p$ is modified directly using an assignment statement or indirectly using a method call within the method $M$. A parameter that is modified by a call is presented using the following template:

```c
/* doubleToColor is a void-accessor that collaborates with Color and modifies a parameter: color via command setColor() */

void BinToColorMap::
    doubleToColor ( double value, Color & color ) const
    {
        double tmp = ( value - m_vmin ) / m_dv;
        int index = static_cast < int > ( 255.* pow( tmp, m_gamma ) );
        assert ( index < 256 );
        color.setColor ( m_reds[index], m_greens[index], m_blues[index] );
    }
```

**Figure 13. An example of a void-accessor method and its generated documentation summary.**

This analysis is done statically on the method using srcML and the database of method stereotypes for the system. To determine whether a parameter is changed by a call, the stereotype of the call is used. The stereotype of calls within a method are extracted from the method/stereotype database explained in section 3.1. When a mutator is called on the
parameter, or it is passed by reference to a *void-accessor*, we can conclude that the parameter is modified.

Consider the *void-accessor* method in Figure 13. The method is from the class `BinToColorMap` from the system HippoDraw and has two formal parameters: `value` and `color`. As the parameter `color` is by reference, it may be modified. The method `setColor()` is called on the parameter `color`. The method `setColor()` is a *mutator* method. Thus, it is determined that the parameter `color` is modified by the call to `setColor()`. Here, there is only one call so it will be included in the short summary, and the list of calls section is omitted.

If more than one parameter is modified by multiple calls, the number of modified parameters is included and the list is moved to the third section. Additionally, if more than five parameters are modified by assignment, the number is included the short description and the list is moved in the third section. The complete list of accessors’ templates is shown in Table 14.

### 3.2.3 Templates for Mutators

*Mutators* are designed to change the object state by modifying one data member (*set*) or multiple data members (*command* or *non-void-command*). Therefore, the data members changed within a method are included in the short description. In total, there are 21 templates for the short descriptions of *mutators*. The following describes the main cases. A *set* method modifies one data member only and we use the following template:

```plaintext
that modifies one data member: <data-member>
```
Table 14. The list of templates used to generate documentation summaries for *Accessor* methods

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Template</th>
</tr>
</thead>
</table>
| **get**             | [returns one data member: <data-member>]  
                       [delegates to <stereotype>: <call>]                                                                                   |
| **Property**        | // return expression has a call  
                       [delegates to <stereotype>: <call>]                                                                                   |
| and predicate       | // returns a single value  
                       [returns a computed value <value>]  
                       [based on data member(s): <data-member>, ..., <data-member>,  
                       and parameter(s): <parameter>, ..., <parameter>,  
                       and value(s) computed from <call>, ..., <call>]                                                                 |
|                     | // returns a constant value  
                       [returns <value>]                                                                                                        |
|                     | // returns an expression (e.g. x + y)  
                       [returns a computed value]  
                       [based on data member(s): <data-member>, ..., <data-member>,  
                       and parameter(s): <parameter>, ..., <parameter>,  
                       and value(s) computed from <call>, ..., <call>]                                                                 |
|                     | // multiple returns  
                       [returns <value>, ..., or <value>,]  
                       [returns a computed value <value>, ..., or <value>,]  
                       [delegates to <stereotype>: <call>, ..., or <stereotype>: <call>]                                                   |
|                     | [depending on data member(s): <data-member>, ..., <data-member>,  
                       and parameter(s): <parameter>, ..., <parameter>,  
                       and value(s) computed from <call>, ..., <call>]                                                                 |
| **Void-accessor**   | // modifies parameters via assignments  
                       [modifies parameters:  
                       <parameter>, ..., <parameter>,]                                                                                   |
|                     | // modifies More than 5 parameters  
                       [modifies <number-of-modified-parameters>]                                                                             |
|                     | // modifies one parameter via a call  
                       [modifies parameter: <parameter>  
                       via <stereotype>: <call>]                                                                                         |

A *command* method executes a complex change of the object’s state. The change involves more than one data member either directly or indirectly using another *mutator*. To identify whether a data member is modified by a call, the same analysis rules as
described for \textit{void-accessor} is used. In addition, the template of \textit{command} is similar to the template of \textit{void-accessor}. If a data member is modified by a call, the data member, the call, and the stereotype of the call are included in the summary using the following:

\begin{verbatim}
that modifies data member: <data-member>
via <stereotype>: <call>
\end{verbatim}

Data members that are modified by assignments are described using the following template:

\begin{verbatim}
that modifies data member(s): <data-mem>, ..., <data-mem>
\end{verbatim}

Figure 14 shows the automated documentation summary for the command method \texttt{resize()}. Four data members are modified by a call inside the method and one is changed via assignment statement.

\begin{verbatim}
/*
resize is a command that
 modifies 5 data member(s): m_sum, m_num, m_sumsq, and m_sumwtsq via resize()
m_values_dirty
calls to- \texttt{reset()} command
*/
void Bins1DProfile::resize ( int number )
{
    m_sum.resize( number + 2 );
    m_num.resize( number + 2 );
    m_sumsq.resize( number + 2 );
    m_sumwtsq.resize( number + 2 );
    reset();
    m_values_dirty = true;
}
\end{verbatim}

\textbf{Figure 14. A generated documentation summary for \textit{command} method}

A \textit{non-void-command} is a special form of a \textit{command} method where the return type is not void or Boolean. This method has two responsibilities: changing data members and returning a value. Therefore, in addition to the modified data members, information about
the returned value is also added to the short description. The modified data members are handled using command templates while the returned value information is expressed using property templates. If a data member is modified and returned, the following is used:

```
that returns a modified data member: <data-member>
```

Figure 15 presents the automatically generated documentation for a non-void-command collaborator method from class BinAxLinear in HippoDraw. The method modifies three data members and returns one of them, m_range. The data member m_range is included in the short description along with the call that causes the change. Modifications of the other data members are reported separately. The complete list of mutators’ templates is shown in Table 15.

```
/**
* setBinWidth is a non-void-command that collaborates with Range and
* returns a modified data member: m_range
* m_range is modified by command: setLength().
* Data members modified- m_width and m_num_bins
* Calls to- getNob() property
*/
const Range& BinAxLinear::setBinWidth (double width){
  assert ( width > 0.0 );
  m_width = width;
  m_num_bins = getNob ( width );
  m_range.setLength ( m_num_bins * width );
  return m_range;
}
```

Figure 15. A non-void-command collaborator method and its documentation.

3.2.4 Templates for Creational Methods.

A factory method is a creational method that returns an object created within the method. Typically, the created object is initialized through a constructor, copy constructor, mutator, or void-accessor. The summarization template for these is:

58
that returns an object: `<object-name>` of type `<object>`

If the object is modified by a call, this information also is included in the short description:

<object-name> is modified by `<stereotype>` : `<call>`

An example of automatic documentation for a factory method is shown in Figure 16.

```cpp
/*
createPickTuple is a factory that
   returns an object: ntuple of type: NTuple
   ntuple is modified by: setName() and setTitle()
Calls to: getTitle() get
*/
NTuple * XyPlotter::createPickTuple ()
{
    NTuple * ntuple = m_plotter -> createPickTuple ();
    string name ( "Pick table for " );
    const string & title = getTitle ();
    name += title;
    ntuple -> setName ( name );
    ntuple -> setTitle ( name );
    return ntuple;
}
```

Figure 16. A factory method and its documentation.

The automatic documentation generation is restricted to the primary action of the method and omits details of the conditional logic. This decision was based on findings from prior work (Rodeghero et al. 2014) where it was observed that when programmers write their own summary for a method, control statements (if and loops) receive lower attention compared to other parts of the method.

Finally, if the approach fails to generate a short description for a method (usually when a method is just a list of calls), the numbers of method calls from each stereotype category (accessors, mutators, collaborative, and creationals) are included in the short description.
Table 15. The list of templates used to generate documentation summaries for
*mutator* methods

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Template</th>
</tr>
</thead>
</table>
| **set**    | //modifies one data member via an assignment
[modifies one data member: <data-member>]

//modifies one data member via a call
[modifies one data member: <data-member> via <stereotype>: <call>]

| Command    | // modifies data members via assignments
[modifies data members:
<data-member>, …, <data-member>]

// modifies More than 5 data members
[modifies <number-of-modified-data-members>]

//modifies data members via calls
[modifies data member: <data-member> via <stereotype>: <call>]

| Non-void-command | // modifies data members via assignments
[modifies data members:
<data-member>, …, <data-member>]

//modifies data members via calls
[modifies data member: <data-member> via <stereotype>: <call>]

// modifies a data member and returns it
[returns a modified data member: <data-member>]

// To report the return expression, property’s templates are used

3.2.5 External Objects

The second section of the documentation (line 2 in Figure 10) includes a list of objects that the method collaborates with. The list helps to identify the external objects, which is particularly important for large methods.

| Collaborates with-<object>, …,<object>

3.2.6 Data Members and Reference Parameters

This part of the documentation is used to prevent the short description from being too long when more than one data member or parameter is modified via calls. Additionally, if
the number of parameters or data members modified by assignments is greater than five, the list is included here. This template is of the following form:

```
Data members modified- <data-member>, ..., <data-member>
<data-member> via <stereotype>: <call>
```

The template above is also used to report modification in parameters by substituting `data-member` with `parameter`.

A minor variation is when a data member is read. To avoid redundancy, if a data member is already in the short description, it is not repeated here.

```
Data member read- <data-member>, ..., <data-member>
```

### 3.2.7 List of Calls and Stereotypes

The last section of the documentation of a method is a table that includes all method calls along with their corresponding stereotypes as shown in Figure 10 line 4. Calls are ordered as they appear in the method. To reduce the complexity, the parameters of each call are removed (Ying & Robillard 2014). Since the list is structured as a table, even a long list will not reduce the readability of the documentation. If the call is a virtual method, and therefore can be overridden by another method from an inheriting class, the stereotype of the call might not be particularly useful. Also, the stereotype of the overridden method is not always readily available. Therefore, if the call is to a virtual method the keyword “virtual” is placed next to the stereotype of the call. The final documentation uses the template shown in Figure 10 to group the four sections just described. More examples of our automatically generated summaries are presented in Appendix A.
3.3 Conclusions

This chapter proposes an automatic approach to generate documentation summaries for C++ methods. Using method stereotypes, a summary template was created for each specific method stereotype. Then, static analysis is used to extract the main components of the method. Finally, each method is re документed with the generated documentation summaries. The approach is highly scalable and can be generalized to other OO programming languages. We believe that the summaries can support comprehension during maintenance.
CHAPTER 4

EVALUATION STUDY

In order for an automatically generated method documentation to be effective, it has to be accurate. It needs to express the underlying meaning of the method, while not missing any important information or including any unnecessary information. Therefore, we propose the following research questions.

1) To what extent does the automatically generated documentation accurately present what a method does without missing important information?

2) If important information is missing, what kind of information is it?

To answer these questions, we performed a survey study. Participants were asked to evaluate automatically-generated documentation, answering questions that were modeled after questions (see Table 16) in the related work by McBurney (McBurney & McMillan 2014). We feel that using these questions, instead of developing our own, mitigates some bias and also allows for some comparison of related work.

Table 16. Questions used in the study. The First four questions are answerable as “Strongly Agree”, “Agree”, “Disagree”, and “Strongly Disagree.” The Last question is open-ended

<table>
<thead>
<tr>
<th>#</th>
<th>Survey Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Independent of other factors, I feel that the summary is accurate.</td>
</tr>
<tr>
<td>2</td>
<td>The summary is missing important information, and that can hinder the understanding of the method.</td>
</tr>
<tr>
<td>3</td>
<td>The summary contains a large amount of unnecessary information.</td>
</tr>
<tr>
<td>4</td>
<td>The summary contains information that helps me understand what the method does.</td>
</tr>
<tr>
<td>5</td>
<td>In a sentence or two, please summarize the method in your own words.</td>
</tr>
</tbody>
</table>
4.1 Study Participants

The study included 77 students in computer science from three universities: Kent State University, The University of Akron, and Youngstown State University. The survey was given as an assignment for their classes and they received credit for completion. Students were assigned the survey by their instructor and given one week to complete the survey.

All participants’ responses were manually examined by two authors. Responses of 13 participants were excluded. Five were excluded due to an incomplete response and eight were excluded because the overall time spent (5-20 min) was significantly less than the overall average time (50 min). That left the responses from 64 participants: 33 graduate and 31 undergraduate (junior and senior) students. From preliminary questions on the participants’ background, 70% of the participants have two to five years of programming experience.

4.2 Experimental Design and Procedure

The study included methods from three open source C++ systems. Table 17 shows the characteristics of these systems. The systems were chosen since they were from three different domains and were all relatively well written in an object oriented manner. The goal of the study was to gain understanding on the quality of the generated documentation summaries across the categories of stereotypes. Therefore, for each stereotype, three to four methods were randomly selected (see Table 18), one method per system.

<table>
<thead>
<tr>
<th>System</th>
<th>Domain</th>
<th>Classes</th>
<th>Methods</th>
<th>KLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmartWin++ 2.0.0</td>
<td>GUI and SOAP library</td>
<td>572</td>
<td>2882</td>
<td>86</td>
</tr>
</tbody>
</table>
A total of 30 methods were selected, ten methods per system. Methods less than 6 LOC were not included. We also avoided methods greater than 30 LOC to avoid participant fatigue as we expected the survey to last approximately one hour. Each one of 64 participants evaluated 10 methods, with each method containing a different stereotype. We felt that more than 10 causes the survey to be too long and hamper our ability to collect valid data.

Table 18. The stereotype of the methods used in the study.

<table>
<thead>
<tr>
<th>Stereotype Category</th>
<th>Number of Methods</th>
<th>Stereotype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessor</td>
<td>3</td>
<td>(1) get collaborator and (1) property</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(1) property collaborator</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>(1) predicate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(2) predicate collaborator</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>(1) void-accessor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(3) void-accessor collaborator</td>
</tr>
<tr>
<td>Mutator</td>
<td>7</td>
<td>(1) set and (3) command</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(3) command collaborator</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>(2) non-void-command</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(3) non-void-command collaborator</td>
</tr>
<tr>
<td>Creational</td>
<td>3</td>
<td>(3) factory</td>
</tr>
<tr>
<td>Collaborational</td>
<td>5</td>
<td>(1) controller predicate, (1) controller property,( 3) controller command</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>Total</td>
</tr>
</tbody>
</table>

The study used a well-known survey tool (qualtrics.com) and was performed online. The survey had four sections: 1) an introduction describing the study; 2) a one-page tutorial on method stereotypes; 3) a 5-question quiz on method stereotypes; and 4) the actual survey questions. The purpose of the quiz is to ensure that the participants understood the
definition of the different stereotypes. However, we did not expect participants to remember all the stereotype definitions. Therefore, Table 1 was provided during the entire study. For each method, the automatic documentation and the method body are presented.

Prior to the start of the study, we estimated the time to evaluate ten methods to be 45-60 minutes with 5-10 minutes per method. The survey tool computed the time automatically. On average, participants completed the survey in 50 minutes (shortest 20 min- longest 3 hours). The average time spent on a method was 5 minutes. Therefore, we believe that participants spent adequate time to evaluate each method.

4.3 Data Collected

In total, we collected 640 evaluations of the summaries of 30 methods. Two authors examined each response. In 47 responses, participants stated that they did not understand the presented method, or their summaries were blank or repeated the automated summaries. These 47 responses were eliminated from the study leaving us with 593 responses.

Table 19. The distribution of the participants' responses.

<table>
<thead>
<tr>
<th>Response category</th>
<th>Accuracy</th>
<th>Miss info</th>
<th>A lot of unnecessary info</th>
<th>What the method does</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly Agree</td>
<td>12%</td>
<td>7%</td>
<td>3%</td>
<td>10%</td>
</tr>
<tr>
<td>Agree</td>
<td>68%</td>
<td>33%</td>
<td>16%</td>
<td>61%</td>
</tr>
<tr>
<td>Disagree</td>
<td>19%</td>
<td>55%</td>
<td>73%</td>
<td>25%</td>
</tr>
<tr>
<td>Strongly Disagree</td>
<td>1%</td>
<td>5%</td>
<td>8%</td>
<td>4%</td>
</tr>
</tbody>
</table>

Table 19 presents the percentage of the automatic documentation summaries that were rated in each question. We can conclude that the results are promising as the majority of ratings for each evaluation question were positive. Hence, we answer the first question as:
The automatic documentation summaries are mostly accurate, have little unnecessary information, and describe the internal of the method, but sometimes miss important information.

4.4 Results and Discussion

The number of responses for “strongly agree” and “strongly disagree” were comparatively small, so we combined “agree” with “strongly agree” and “disagree with strongly disagree” for analysis of the results. We use the terms agreement and disagreement for the discussion in this section.

More detailed information about the results is now presented. The summaries written by the participants were compared to the generated documentation summaries to understand the reasons behind their ratings. Since the generated summaries depend on the stereotype of a method, it is important to analyze the ratings of each stereotype, as presented in Table 20. This information will now be discussed in terms of the four evaluation questions. Figure 17 shows the ratings of graduate and undergraduate students. The graph indicated that the ratings of graduate and undergraduate are generally similar.
4.4.1 Per System

We analyzed the responses given by participants for each of the three systems (see Figure 18). The three systems were intentionally chosen from different domains. The results indicate that the documentation for all three systems were found to be accurate by the majority (79%-81%) and to not include unnecessary information (80%-85%).

Some methods were rated by 60%-50% of participants as “not properly expressing what the method does”. The percentage of these methods for CodeBlocks, HippoDraw, and SmartWin were 10%, 18%, and 11% respectively.
The responses for “missing important information” received the lowest ratings compared to the other evaluation questions. The percentage of methods that were determined from the survey to be missing important information for CodeBlocks, HippoDraw, and SmartWin is 30%, 36%, and 40%, respectively. As the ratings of all three systems are not significantly different, and all systems were from different domains, we believe that this indicates that the approach is generalizable to multiple application domains.

Table 20. The distribution of the participants’ responses per stereotype. Note that clb is short for collaborator.

<table>
<thead>
<tr>
<th>Method Stereotype</th>
<th>Accuracy</th>
<th>Miss Important Info</th>
<th>A lot of unnecessary Info</th>
<th>What the method does</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get Clb</td>
<td>9%</td>
<td>8%</td>
<td>6%</td>
<td>0%</td>
</tr>
<tr>
<td>Property</td>
<td>0%</td>
<td>8%</td>
<td>6%</td>
<td>0%</td>
</tr>
<tr>
<td>Property Clb</td>
<td>3%</td>
<td>8%</td>
<td>6%</td>
<td>0%</td>
</tr>
<tr>
<td>Controller Property</td>
<td>10%</td>
<td>6%</td>
<td>6%</td>
<td>0%</td>
</tr>
<tr>
<td>Predicate</td>
<td>7%</td>
<td>5%</td>
<td>6%</td>
<td>0%</td>
</tr>
<tr>
<td>Predicate Clb</td>
<td>10%</td>
<td>5%</td>
<td>6%</td>
<td>0%</td>
</tr>
<tr>
<td>Controller Predicate</td>
<td>7%</td>
<td>9%</td>
<td>6%</td>
<td>0%</td>
</tr>
<tr>
<td>Void-accosor</td>
<td>17%</td>
<td>6%</td>
<td>5%</td>
<td>0%</td>
</tr>
<tr>
<td>Void-accosor Clb</td>
<td>7%</td>
<td>9%</td>
<td>6%</td>
<td>0%</td>
</tr>
<tr>
<td>Set</td>
<td>35%</td>
<td>4%</td>
<td>6%</td>
<td>0%</td>
</tr>
<tr>
<td>Command</td>
<td>16%</td>
<td>6%</td>
<td>6%</td>
<td>0%</td>
</tr>
<tr>
<td>Command Clb</td>
<td>12%</td>
<td>6%</td>
<td>6%</td>
<td>0%</td>
</tr>
<tr>
<td>Controller Command</td>
<td>8%</td>
<td>6%</td>
<td>6%</td>
<td>0%</td>
</tr>
<tr>
<td>Non-void-command</td>
<td>18%</td>
<td>6%</td>
<td>6%</td>
<td>0%</td>
</tr>
<tr>
<td>Non-void-command Clb</td>
<td>13%</td>
<td>7%</td>
<td>6%</td>
<td>0%</td>
</tr>
<tr>
<td>Factory</td>
<td>15%</td>
<td>7%</td>
<td>6%</td>
<td>0%</td>
</tr>
</tbody>
</table>

4.4.2 Accuracy

All method documentation was judged to be accurate by the majority of participants. The stereotypes that had the highest rated summaries (i.e., 80% or more agreement from participants) are: get collaborator, property, property collaborator, controller property, command, non-void-command collaborator and factory. This includes all accessor methods, except for predicate.
Documentation summaries of two *predicate collaborator* methods were rated as inaccurate by 25%-35% of the participants. In one method, the short description included information about the data members and calls that control the returned value. This information was included in one list, even though it was a part of two if-conditions. The documentation summaries of the participants reported the two if-conditions separately in two steps. In the second *predicate collaborator* method, the returned value was modified twice (via call and via assignment) and the automatic documentation reflected only one (modification by the call) that was interpreted as inaccurate by some of the participants.

Finally, the documentation summaries of one *controller command* method and one *command collaborator* method were rated as inaccurate by 36% and 38% of the participants, respectively. We did not have enough information to explain these ratings. However, a common observation about their short descriptions was that they included the number of method calls from each stereotype category as shown in Figure 19 which could be part of the reason for the participants rating these as less accurate.

All method documentation was judged to be accurate by the majority of participants. The stereotypes that had the highest rated documentation (i.e., 80% or more agreement from participants) are: *get collaborator, property, property collaborator, controller property, command, non-void-command collaborator* and *factory*.

Documentation summaries of two *predicate collaborator* methods are rated as inaccurate by 25%-35% of the participants. In one method, the short description included information about the data members and calls that control the returned value. This information is included in one list, even though it was a part of two if-conditions. The
summaries of the participants reported the two if-conditions separately in two steps. In the second predicate collaborator method, the returned value is modified twice (via call and via assignment) and the generated documentation summary reflected only one (modification by the call) that is interpreted as inaccurate by some of the participants.

```c
/**
 * setLog is a command that collaborates with 6 objects and calls 4 mutators, a controller and a collaborator
 * ...
 */
```

**Figure 19. The short description generated for a command collaborator method.**

Finally, the documentation of one controller method and one command collaborator method is rated as inaccurate by 36% and 38% of the participants, respectively. We did not have enough information to explain these ratings. However, a common observation about their short descriptions is that they only included the number of method calls from each stereotype category (as shown in Table 14) without any other details of method's complex behavior which could be part of the reason for the participants rating these as less accurate.

### 4.4.3 Missing/Containing Important Information

The results of the second and fourth questions (missing important information and expressing what the method does) are related and discussed together.

It is noteworthy that documentation of 25 out of 30 methods is rated to properly express what the method does by the majority of participants. The documentation of stereotypes consistently rated to properly express what the method does are: property,
predicate, command, non-void-command, factory, and void-accessor (3 out of 4 methods rated properly), controller command (2 out of 3 methods rated properly), and non-void-command collaborator (2 out of 3 methods rated properly). In addition, 17 out of the 25 methods are rated as “not missing important information” by the majority of participants. The documentation of stereotypes that consistently rated to include all of the important information are: command, non-void-command, void-accessor property.

Of the generated documentation cases, 5 out of 30 methods are rated by the majority to not properly describe “what the method does” and to “miss important information”. Note, all of them are collaborational methods. The first in this group is a void-accessor collaborator method. The documentation for this method included a list of objects, data members, and a list of calls without their stereotypes. The stereotypes of the calls are not determined (method calls are not originally stereotyped) by the automated approach (a rare case). This method is from CodeBlocks and only 0.01% of the methods are not stereotyped. However, the information included in the documentation is rated as necessary by 80% of the participants.

The next two methods of these five - controller command and command collaborator - have similar behavioral characteristics. Their short descriptions included the number of method calls from each stereotype category (as shown in Figure 19). This information might be considered insufficient to reflect the method’s primary behavior, as they are somewhat complex.

The fourth method documentation that missed important information is for a command collaborator method. The documentation included the data member modified along with
the call that modifies it. However, the modification is done iteratively within a while-loop. Participants’ documentation that did not agree with the automatically generated documentation described how the loop iterates through one local variable and modified the data member.

The last method documentation that was judged to “not express what the method does” is for a non-void-command collaborator method. The automatically generated documentation correctly specifies the two method calls that are returned using an “or” conjunction. However, a pointer to an external object used to call the two methods is initialized using another call. According to participants’ summaries, information about “how the local pointer is changed before the call” is considered to be critical. The same observation applied to one controller command method that requires further investigation.

Three cases of the automated documentation were judged to express what the method does but miss some important information. The first is for a factory method where the short summaries emphasized the object created and returned. Participants considered the condition that checks the success of creating the object to be important and that it should be included in the documentation. We plan to improve the template for factory in the future to address this issue. Similarly, in case of the set method, information about “under what condition the data member is modified” is expected by the participants. We plan to improve that template also.

The third method is a property collaborator. Its documentation included the name of two calls used to compute the returned value. However, participants’ summaries emphasized the type of the returned value (i.e., a string). This information is felt to be
missing by some participants. We plan to enhance the template to reflect the type of the returned value without increasing the length of the documentation (e.g., for a string, we can use “returns a computed string” instead of “returns a computed value”).

The stereotype consistently rated as “missing important information” is predicate (2 out of 3 methods). Participants wanted information about the action performed by the method (checking the existence of an element, evaluating a value for being in a certain range, or other actions). We plan to study and improve the rules of the predicate’s documentation to include information from conditional statements.

4.4.4 Including Unnecessary Information

This question might be found biased because of using of the phrase “a large amount” in the question. However, this question received the best rating compare to the other evaluation questions. All 30 methods are judged by the majority of participants to not include unnecessary information. In fact, 87% of the documentation is rated as not having unnecessary information by >80% of the participants. Only 4 out of 30 documentation received ratings (23%-32%) that indicated they included unnecessary information: two command collaborator, one controller predicate, and one non-void-command collaborator. We do not have enough evidence to fully explain these ratings but can make some observations. In the case of the two command collaborator and one controller predicate methods, the list of calls in the documentation included multiple set and get methods, which may be unnecessary. In one non-void-command collaborator method, the
stereotypes of 3 out of 5 calls are not specified. Participants might consider including any calls without stereotypes to be unnecessary.

Of the generated documentation summaries, 25 out of 30 were half the size of the original method or smaller. Only five summaries were about the same size as the method (in LOC). This case only occurred for short methods. However, the majority of participants consider the included information to be necessary.

Based on the above discussion, we answer RQ2 as the following:

```
Control-flow statements (if/switch) are found to be important for including in the method summary especially for predicates. In addition, previous work suggested that local variables were judged by programmers to be “useless” to include in the summary [47]. However, we observed that information about local variables that are objects appear to be important to a number of participants.
```

4.5 Threats to Validity

A number of issues could affect the results of the survey study we performed and so may limit the generalizability of the results. The quality of the generated documentation might be influenced by the stereotype of the method. To control this, we have included three or four methods from each stereotype.

A learning effect might occur during the study. As soon as participants evaluated the first method, they knew what would be asked later. We did not try to mitigate or control this factor. However, the last section of the survey asked the participants to judge the
overall documentation. We compared this rating with the participants’ responses for all methods (see Table 19) and we found no significant difference. Furthermore, the participants are graduate and undergraduate (junior and senior) students. We cannot conclude whether the study with more expert programmers will reveal the same result.

4.6 Conclusions

Our evaluation suggests that using method stereotype to generate documentation summaries for C++ methods is a practical and efficient solution. However, some challenges are encountered and require further analysis. Collaborator and controller methods are more difficult to summarize compared to the other stereotypes. They usually perform complex tasks that involve multiple external objects. In these methods, the stereotype alone is not sufficient to determine what the method does.

To further assess the structure of the summarization, we are planning to perform another evaluation study. Some parts of the summarization will be randomly removed from the original summarization to study the effect of their absence. The study is presented in the next chapter.
CHAPTER 5
IMPROVING AND EVALUATING THE IMPORTANCE OF THE FOUR DOCUMENTATION SECTIONS

The results of the first evaluation indicated that the automatically generated documentation accurately expresses what the method does, and does not include a large amount of unnecessary information but sometimes misses important information. For example, control-flow statements are found to be important for the method documentation. The majority of participants stated that they were missing and the lack of them decreased the overall rating of the documentation. Therefore, we improve the automated approach by including information from if and switch statements.

5.1 Improving The Automated Approach

In particular, if the main action of a method is entirely located in an if-statement, the condition is added to the short description using the following:

```cpp
/* depending on data member(s): <data-mem1>, ..., <data-mem_n>
   and parameter(s): <parameter1>, ..., <parameter_m>
   and value(s) computed from <call1>, ..., <call_k> */
```

```cpp
void BinToColorMap::doubleToColor(double value, Color& color) const
{
    double tmp = (value - m_vmin) / m_dv;
    int index = static_cast<int>(255. * std::pow(tmp, m_gamma));
    if (index < 256)
        color.setColor(m_reds[index], m_greens[index], m_blues[index]);
}
```

Figure 20. An example of a void-accessor method and its improved summary.
At this point, we do not handle nested if statements. Figure 20 shows the improved short description of a \textit{void-accessor} method by adding the phrase \textit{depending on parameter(s): index}

5.2 Second Study

The goal of this study is to evaluate the improved approach. Furthermore, it examines the importance and necessity of the four documentation sections: short description, list of external objects, list of data members read, and list of calls and their stereotypes. To accomplish this, each one of these sections are randomly removed from a subset of selected automatically generated documentation. The study also investigates if including information from control-flow statements can improve summarization. Therefore, we seek to answer the following questions:

1) To what extent does information from conditional statements improve the automatically-generated documentation?

2) To what extent does the absence of one section of the documentation improve/affect the results?

5.2.1 Participants

The study includes 75 students in computer science from Kent State University: 51 undergraduate students and 24 graduate students. The survey was given as an extra credit assignment for their classes and they received credit for completion. Students were assigned the survey by their instructor and given one to two weeks to complete the survey. From preliminary questions on the participants’ background, 10\% of the participants have
more than 5 years of programming experience and about 50% have two to five years of programming experience. Finally, 20% of the participants have worked in industry.

### 5.2.2 Experimental Design and Procedure

The study includes 56 methods from four open source C++ systems (systems 2-5 from Table 17). Similar to the previous study, the selected methods are from different stereotype categories. The percentages of these methods per stereotype category are: 14% accessors, 25% accessor collaborators, 8% mutators, 18% mutator collaborators, 25% controller, and 10% creational methods. The size of methods ranges from 16 to 50 LOC.

We describe the characteristic of the 56 different documentation cases generated as the following: 18 include all sections, 10 without the list of objects, 10 without the short description, 9 without the list of data members read, and 9 without the list of calls and their stereotypes. Through the analysis of the data, we refer to the above five groups as documentation categories. The selection of the category to remove is random, with three conditions: 1) the documentation has at least one section to present; 2) removing the list of objects is required only for collaborator or controller methods; 3) the method name and its stereotype are present in all cases.

This study uses the same four survey sections as in the first study. Each participant evaluated 6 documentation cases where the first two cases include all four sections to familiarize participants with the documentation structure. The other four cases include all sections except one as previously explained. Overall, 450 documentation summaries of
75 participants were analyzed. Finally, participants are not aware that some sections are missing from the displayed documentation.

In addition to the questions asked in the previous survey (questions 2 to 5 from Table II), we ask participants to evaluate the structure of the documentation using the following question. “The structure of the documentation is easy to read and map to the code.” Finally, when a participant gives an answer of disagree on a question, a fill-in-the-blank text box appears that asks about what aspect that she/he disagrees with. On average, participants completed the survey in 50 min. (shortest 15 min., longest 1 hour). Overall, 450 records of 75 participants are analyzed.

5.3 Results and Discussion

In the second study, 75 participants evaluated 18 documentation full summaries. Figure 21 shows a comparison of the first and second evaluation ratings. The rating of all of the four evaluation questions improved in the following ways: 1) accuracy increases by 7%; 2) missing important information decreases by 10%; 3) expressing what the method does increases by 15%; 4) containing a large amount of unnecessary information decreases by 9%.

Therefore, the first research question is answered as:

Including information from conational statements (if and switch) improves the accuracy and content adequacy of automated documentation approach.
To understand the effect of the absence of one section, we compute the total number of responses for each of the five documentation categories per evaluation question. To study the significance of the absence of each section for each evaluation question we use the Mann-Whitney Test (Mann & Whitney 1947), a non-parametric test that does not assume normality of the data. We conclude that the result of an evaluation question is significant if and only if \( p < .05 \). We found no statistical difference of the ratings of the last evaluation question, i.e., *structure of the documentation*, in all documentation categories. In fact, for all documentation categories, 82\% or more of the ratings agree that the structure is easy to read and map to the code. This result suggests that the proposed structure of the documentation sections is appropriate.

We combined “agree” with “strongly agree” and “disagree” with “strongly disagree” for analysis of the results. This is done because “strongly agree” and “strongly disagree” do not notably differ when a section of documentation is removed. Some critical increment or decrement of these values is reported through the discussion.

Figure 21. A comparison of the first and second evaluation.
5.3.1 No short description.

A total of 75 records are collected for 10 documentation cases without short description, and compared to the ratings of 18 of the full documentation cases. Comparing to full documentation, the ratings of documentation that is missing the short description decrease visibly in accuracy (13%) and what the method does (20% where half of them are strongly disagree). However, when participants are asked what aspect is inaccurate, they state that more information is needed. Furthermore, the ratings of no short description category increase by 23% to be missing important information. These drops in ratings indicate that the short description plays an important role in the documentation.

To examine the significance of the absence of short description, Mann-Whitney Test is applied for the four evaluation questions. First, for each documentation case that omits short description, the average ratings of each evaluation question are computed. Second, for each full-documentation case, the average ratings of each evaluation question are computed. Finally, the Mann-Whitney Test is used to compare the two documentation categories in Table 21. The same steps are also repeated for the rest of the documentation categories in Table 21- Table 24.

We found statistical difference in the following evaluation questions: accuracy and missing important information. These results indicate that the short description is essential to the automatically generated documentation.
Table 21. Mann-Whitney Test for doc. without short description.

<table>
<thead>
<tr>
<th>Evaluation question</th>
<th>Category</th>
<th>N</th>
<th>Median</th>
<th>SD</th>
<th>U</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>Full</td>
<td>18</td>
<td>85.9</td>
<td>28.5</td>
<td>45.5</td>
<td>.032</td>
</tr>
<tr>
<td></td>
<td>No short</td>
<td>10</td>
<td>75</td>
<td>54.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Miss info</td>
<td>Full</td>
<td>18</td>
<td>26.1</td>
<td>75</td>
<td>44</td>
<td>.027</td>
</tr>
<tr>
<td></td>
<td>No short</td>
<td>10</td>
<td>50</td>
<td>75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A lot of unnec. info</td>
<td>Full</td>
<td>18</td>
<td>11.3</td>
<td>33.3</td>
<td>81</td>
<td>.656</td>
</tr>
<tr>
<td></td>
<td>No short</td>
<td>10</td>
<td>6.2</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>What the method does</td>
<td>Full</td>
<td>18</td>
<td>85.8</td>
<td>33.3</td>
<td>51.5</td>
<td>.064</td>
</tr>
<tr>
<td></td>
<td>No short</td>
<td>10</td>
<td>73.2</td>
<td>83.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structure</td>
<td>Full</td>
<td>18</td>
<td>82.5</td>
<td>50</td>
<td>87</td>
<td>.88</td>
</tr>
<tr>
<td></td>
<td>No short</td>
<td>10</td>
<td>84.5</td>
<td>66.6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 22. Mann-Whitney Test for doc. without list of calls.

<table>
<thead>
<tr>
<th>Evaluation question</th>
<th>Category</th>
<th>N</th>
<th>Median</th>
<th>SD</th>
<th>U</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>Full</td>
<td>18</td>
<td>85.9</td>
<td>28.5</td>
<td>59.5</td>
<td>.26</td>
</tr>
<tr>
<td></td>
<td>No list of calls</td>
<td>9</td>
<td>88.8</td>
<td>33.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Miss info</td>
<td>Full</td>
<td>18</td>
<td>26.1</td>
<td>75</td>
<td>76.5</td>
<td>.81</td>
</tr>
<tr>
<td></td>
<td>No list of calls</td>
<td>9</td>
<td>27.2</td>
<td>35.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A lot of unnec. info</td>
<td>Full</td>
<td>18</td>
<td>11.3</td>
<td>33.3</td>
<td>47</td>
<td>.06</td>
</tr>
<tr>
<td></td>
<td>No list of calls</td>
<td>9</td>
<td>.00</td>
<td>14.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>What the method does</td>
<td>Full</td>
<td>18</td>
<td>85.8</td>
<td>33.3</td>
<td>69.5</td>
<td>.54</td>
</tr>
<tr>
<td></td>
<td>No list of calls</td>
<td>9</td>
<td>85.7</td>
<td>44.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structure</td>
<td>Full</td>
<td>18</td>
<td>82.5</td>
<td>50</td>
<td>70</td>
<td>.56</td>
</tr>
<tr>
<td></td>
<td>No list of calls</td>
<td>9</td>
<td>87.5</td>
<td>33.3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.3.2 No calls and stereotypes

A total of 75 records are collected for 9 documentation cases without list of calls and their stereotypes, and are compared to the ratings of 18 of the full documentation cases. Compared to full documentation, the ratings of documentation missing the list of calls and stereotypes slightly improve in three evaluation questions. This improvement appears in case of accuracy 4%, contains unnecessary information 6%, and structure 5%. It seems that the list of calls and their stereotypes might add additional overhead for a few participants. However, for all the evaluation questions, we found no statistical significant
difference between full documentation and documentation that omits the list of calls and stereotypes (see Table 22).

Of the documentation that omits the list of calls and their stereotypes, 3/9 are evaluated by the majority to miss important information. Most of the participants complained that calls are not explained in the documentation. In fact, a few of them asked why the stereotypes of calls are not included. These observations suggest that participants value the inclusion of the list of calls and their stereotypes.

5.3.3 No data members read

A total of 75 records are collected for 9 documentation cases without list of data members, and are compared to the ratings of 11 of the 18 full documentation cases. Seven methods are excluded from this analysis as they are controller methods that by definition have no data members. The ratings of documentation that omits data members read are slightly improved in three evaluation questions. This improvement appears in case of missing important information, what the method does, and structure by 6%, 10%, and 11%, respectively. We do not have enough evidence to explain why such improvement occurs. It could be that shorter documentation (one line shorter) happens to be more preferable. However, no statistical difference is found in all of the evaluation questions as demonstrated in Table 23. Consequently, we conclude that the absence of the list of data members read is not as critical.
5.3.4 No list of objects

A total of 75 records are collected for 10 documentation cases without list of objects, and are compared to the ratings of 9 of the 18 full documentation cases of collaborative methods. Nine methods that are non-collaborational are excluded from this analysis because by definition they do not have the list of objects.

Table 23. Mann-Whitney Test for doc. without list of DM read.

<table>
<thead>
<tr>
<th>Evaluation question</th>
<th>Category</th>
<th>N</th>
<th>Median</th>
<th>SD</th>
<th>U</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>Full</td>
<td>11</td>
<td>87.5</td>
<td>28.5</td>
<td>46.5</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>No list of DM</td>
<td>9</td>
<td>88.8</td>
<td>30.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Miss info</td>
<td>Full</td>
<td>11</td>
<td>18.1</td>
<td>57.1</td>
<td>46.5</td>
<td>.81</td>
</tr>
<tr>
<td></td>
<td>No list of DM</td>
<td>9</td>
<td>20.0</td>
<td>42.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A lot of unnec. info</td>
<td>Full</td>
<td>11</td>
<td>9.0</td>
<td>33.3</td>
<td>45</td>
<td>.72</td>
</tr>
<tr>
<td></td>
<td>No list of DM</td>
<td>9</td>
<td>11.1</td>
<td>33.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>What the method does</td>
<td>Full</td>
<td>11</td>
<td>83.3</td>
<td>33.3</td>
<td>30.5</td>
<td>.41</td>
</tr>
<tr>
<td></td>
<td>No list of DM</td>
<td>9</td>
<td>90.0</td>
<td>20.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structure</td>
<td>Full</td>
<td>11</td>
<td>83.3</td>
<td>50</td>
<td>35.5</td>
<td>.27</td>
</tr>
<tr>
<td></td>
<td>No list of DM</td>
<td>9</td>
<td>90</td>
<td>28.5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 24. Mann-Whitney Test for doc. without list of objects.

<table>
<thead>
<tr>
<th>Evaluation question</th>
<th>Category</th>
<th>N</th>
<th>Median</th>
<th>SD</th>
<th>U</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>Full</td>
<td>9</td>
<td>84.5</td>
<td>25</td>
<td>31</td>
<td>.249</td>
</tr>
<tr>
<td></td>
<td>No list of objs</td>
<td>10</td>
<td>84.5</td>
<td>18.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Miss info</td>
<td>Full</td>
<td>9</td>
<td>33.3</td>
<td>65</td>
<td>36.5</td>
<td>.487</td>
</tr>
<tr>
<td></td>
<td>No list of objs</td>
<td>10</td>
<td>43.7</td>
<td>85.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A lot of unnec. info</td>
<td>Full</td>
<td>9</td>
<td>11.1</td>
<td>33.3</td>
<td>40.5</td>
<td>.709</td>
</tr>
<tr>
<td></td>
<td>No list of objs</td>
<td>10</td>
<td>7.1</td>
<td>37.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>What the method does</td>
<td>Full</td>
<td>9</td>
<td>80</td>
<td>50</td>
<td>41.5</td>
<td>.666</td>
</tr>
<tr>
<td></td>
<td>No list of objs</td>
<td>10</td>
<td>95.0</td>
<td>57.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structure</td>
<td>Full</td>
<td>9</td>
<td>80.3</td>
<td>50</td>
<td>34.5</td>
<td>.37</td>
</tr>
<tr>
<td></td>
<td>No list of objs</td>
<td>10</td>
<td>87.5</td>
<td>71.4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Except of “missing of important information”, none of the evaluation questions seem to be affected by the absence of list of objects. Compare to full documentation, the ratings of documentation that does not include list of objects increase by 5% in case of missing important information. However, no statistical difference is found in all of the evaluation
questions as demonstrated in Table 24. Consequently, we conclude that the absence of the list of objects is not as critical.

Finally, second research question is answered as the following:

```
Based on the statistical analysis, the most important section of the documentation is “short description”.
```

5.4 Threats to Validity

As the design and procedure of the second study is similar to the first study, the same threats to validity could be encountered. These issues are discussed in Chapter 4 section 4.5 (page 75).

5.5 Conclusions

Based on the analyzed ratings from 75 programmers, we conclude that the automatically-generated documentation accurately expresses what the method does. The result also demonstrates that a template-based technique, specialized for each stereotype, combined with simple static analysis is a practical approach for automatically-generated documentation. The evaluation also reveals that the most important part of the automatically-generated documentation is the short description, as their absence significantly reduce the documentation ratings. In addition, removing the list of data members read or the list of calls and their stereotypes seems to be more preferable for some of participants. Therefore, we consider that allowing programmers to choose the desired level of details of the documentation is the best approach.
CHAPTER 6

EYE TRACKING STUDY OF SUMMARIZING SOURCE CODE

Despite the amount of work done in source code summarization (Abid et al. 2015; Haiduc et al. 2010; McBurney & McMillan 2014; L. Moreno et al. 2013; Sridhara et al. 2011a, 2011b; Wong et al. 2013), information to be included in summaries needs further analysis. Rodeghero et al. conducted an eye-tracking study to determine the statements and terms that programmers view as important when they summarize a method (Rodeghero et al. 2014). They developed a new weighting scheme for Vector Space Model (VSM) by giving more weight to terms developers look at more during summarization. This new weighting scheme outperforms existing VSM scheme when compared to human term-based summaries (Rodeghero et al. 2014).

Here, we replicate and expand Rodeghero et al. eye-tracking study by overcoming some of their study’s limitations and constraints. The first limitation is that the maximum length of methods in their study is 22 (lines of code) because the stimulus used is a single screen of text that needs to appear all at once on the screen (no scrolling is allowed). In our study, the largest method is 80 lines of code and scrolling/switching between files is permitted. Second, in Rodeghero et al. study, methods are presented in isolation whereas our study presents methods within the Eclipse environment where developers have access to the entire system. We use a tool, iTrace, that is an Eclipse plugin to incorporate implicit eye tracking (Shaffer et al. 2015) and support such things as scrolling within a file and
switching between files. Third, Rodeghero et al. study does not account for methods with different stereotypes (Rodeghero & McMillan 2015) while our study includes a variety of methods based on many different stereotypes (N. Dragan et al. 2006). Method stereotypes are a concise description of a method’s intended design (N. Dragan et al. 2006; Natalia Dragan et al. 2010) that have been successfully applied to the problem of source code summarization (Abid et al. 2015; L. Moreno et al. 2013).

The overall goal of this study is to understand what programmers focus on when they summarize Java source code using a realistic integrated development environment setting. Since we compare our results to the Rodeghero et al. study, we analyze eye gaze on the three locations studied by them (Rodeghero et al. 2015; Rodeghero & McMillan 2015; Rodeghero et al. 2014) namely, method signatures, method calls, and control flow statements. In addition to the above, our work expands the analysis in several different directions based on the level of expertise of programmers, size (length) of methods, and stereotype of methods.

Our work also considers locations outside the method scope that programmers might find important such as the class name, data member declarations, and other related methods. In the previous work, this type of analysis could not be done as the methods were presented in isolation (Rodeghero et al. 2014). The following describes the design of the eye tracking study and the procedure of collecting the data.
6.1 Study Participants

The study is performed using 16 programmers: 5 experts and 11 novices. Two of the experts are industry professionals working at Turning Technologies LLC. in Youngstown and three are PhD students at Kent State University. One industry expert had between two to five years of programming experience and the other four had greater than 5 years of programming experience. We consider the PhD students to be experts as they are heavily involved in coding for open source projects. Novices are undergraduate students at Youngstown State University and graduate students at Kent State University with between one to five years of programming experience, most of which have about a year of experience.

6.2 Study Systems and Method Selection

The study includes 63 methods from five open source Java systems randomly selected from different domains (see Table 25). While the selection of methods chosen to be summarized is random, we maintain two conditions. First, we eliminate trivial methods such as setters, getters, and empty methods. Second, we select methods from different stereotype categories (45% of the selected methods are accessors or collaborational accessors, 48% of the selected methods are mutators or collaborational mutators, and 6% are creationals). For a detailed description of the stereotypes used, please refer to section 2.1. Based on the above criteria, a total of 63 methods were selected from the systems shown in Table 25.
The 63 methods are divided into three groups, each of which had 15 methods that needed to be summarized. This is done to balance the methods’ sizes and stereotypes in each group and to force overlap between the methods being summarized. Furthermore, there are six methods that appear first (in sequence) in two groups to increase the number of data points that are collected for each method (i.e., increase overlap between groups of methods summarized). These methods range from 31 to 53 LOC. Each participant is randomly assigned to one of these three groups. The first group (Group A) consisted of one expert and five novices. The second group (Group B) consisted of four experts and four novices. Finally, the last group (Group C) is performed by two novices.

We categorize a method based on how long the method is in terms of lines of code. The size of short, medium and long methods ranges between 9-22 LOC, 23-39 LOC, and 40-80 LOC respectively. A line of code is counted if and only if it is not empty and is not a comment. We used this split to maintain balanced number of methods in each size category. Furthermore, methods in Rodeghero et al.’s study may fall between the first and second category. Therefore, methods that are larger than 40 LOC are analyzed separately to study the impact of summarizing larger methods.

Table 25. An overview of the Java systems used in the study.

<table>
<thead>
<tr>
<th>System version</th>
<th>Domain</th>
<th># of methods</th>
<th>methods used</th>
<th># of classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgoUML 0.30.2</td>
<td>UML diagramming tool</td>
<td>14635</td>
<td>15</td>
<td>2673</td>
</tr>
<tr>
<td>MegaMek 0.36.0</td>
<td>Computer game</td>
<td>12490</td>
<td>15</td>
<td>2308</td>
</tr>
<tr>
<td>Siena 1.0.0</td>
<td>Database library</td>
<td>4116</td>
<td>12</td>
<td>297</td>
</tr>
<tr>
<td>sweetHome3d 4.1</td>
<td>Interior design application</td>
<td>6084</td>
<td>12</td>
<td>1757</td>
</tr>
<tr>
<td>aTunes 3.1.0</td>
<td>Audio player</td>
<td>9579</td>
<td>9</td>
<td>215</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>46904</strong></td>
<td><strong>63</strong></td>
<td><strong>7250</strong></td>
</tr>
</tbody>
</table>

6.3 Study Procedure and Instrumentation

The study was conducted in the Software Engineering Research and Empirical Studies lab at Youngstown State University. We first obtained IRB approval for the study. On the day of the study, the participants first signed an informed consent form and filled out a background questionnaire. Next, they were given a short description of how the study would be conducted. They were also given an overview of each system. A hard copy of all the methods to be summarized was available to them in case they needed to refer to it. Before they began the actual study, they were given three examples of how a summary would look like from two of the systems shown in Table 25. For this sample task, the method’s source code was shown along with a good summary for the methods. This was a necessary step as it was important for the participants to understand what they were expected to do during a summarization task. During this sample task, any questions the participant had were answered. The study took between 45 – 90 minutes for 15 method summarizations.

The participants were told that their main task is to read the assigned methods and write a summary of the method. They were also told that they could navigate the codebase if they needed. The entire study is conducted inside the Eclipse environment using the Eclipse plugin iTrace (Shaffer et al. 2015). iTrace is able to collect eye tracking data of where a developer is looking and map it on the fly to source code elements looked at even in the presence of file scrolling and file switching. Code folding was turned off. The Eclipse environment was setup with all the projects and assigned methods open in various
tabs. The participants are also able to view the method while they were writing the summary.

The participants were seated in front of a 24-inch LCD monitor. They were not required to run or build the software systems. Before the study began, we calibrated each participant using a 9-point calibration system. This is a necessary step for the eye tracker to correctly capture data for each participant. A moderator was responsible to start and stop eye tracking via iTrace for each of the 15 methods to be summarized. This was done to make sure the tracking is done consistently across all participants. The moderator also randomly assigned the participant to one of three groups. The participants wrote their summary in a text file. Gazes on the text files were also collected. At the end of each summary, we had a time-stamped eye gaze session of line-level gaze data on the source code and summary file that we used to answer our research questions.

6.4 Eye Tracking Apparatus and Measures

We use the Tobii X60 eye tracker to collect gaze data within the iTrace (Shaffer et al. 2015) environment. The eye tracker generates 60 raw gaze samples per second. The raw eye gaze is then passed through a fixation filter to generate fixations. A fixation is the stabilization of the eyes on some object of interest for a certain duration (Rayner 1998). The fixation filter in iTrace is implemented based on Olsson (Olsson 2007).

Two types of eye-movement data are used: fixations and their durations (gaze time). These are the same measures used by Rodeghero et al. They are widely used (Crosby & Stelovsky 1990; Uwano et al. 2006) and allow for comparing methods. We define gaze
time as the total number of milliseconds spent on a region of interest (ROI - e.g., a method, a keyword, etc.). Our fixation filter was set to count fixations that were more than 100 milliseconds. Throughout the analysis, we refer to the number of fixations as the number of visits on particular regions of interest.

### 6.5 Correctness

We reviewed the 256 summaries written by developers to judge their correctness. Each human summary is reviewed by two reviewers. When both of them agree about the correctness of a summary, then it is correct.

Of the 256 responses, 44 summaries (6 experts’ summaries and 38 novices’ summaries) were judged inaccurate. Some cases are due to miss-locating the assigned method or to stating that they are unable to understand the assigned method. In some responses, participants had partially or fully incorrect summaries. This indicates that they did not understand the method. These records are not included in the analysis presented here because they may affect the accuracy of the final results.

### 6.6 Gaze Behavior Inside the Method

This section analyzes within the method locations that developers read during summarization. In particular, we analyze the amount of gaze time and number of visits that programmers spend to read a method’s signature, calls, and control flow statements.

The collected eye-movements data are processed in two steps. The first step is to identify whether the term corresponds to a signature, call, or control flow term. Second is to compute the adjusted gaze time and number of visits based on the size of each location.
We seek to answer the following research questions.

RQ1) Considering all methods summarized, to what extent do experts and novices focus on a method’s signature, method’s body, calls invocation in a method, and method’s control flow?

RQ2) Does the size (length) of the method (small, medium, large) have an impact on what experts and novices look at most during the summarization task?

RQ3) Does the method stereotype (predicate collaborator/ property collaborator/ void-accessor collaborator/ command collaborator) have an impact on what experts and novices look at during summarization?

RQ1 investigates the differences and similarities of areas in methods that experts and novices focus on the most. RQ2 seeks to determine if the method size in terms of lines of code has an effect on what experts and novices examine. It might be the case that a different strategy is employed for longer methods vs. shorter methods. Finally, RQ3 inspects if the stereotype of a method changes the way it is read by novices or experts. Methods are selected from commonly used stereotype categories to study the method stereotype role on developers reading behaviors.

### 6.6.1 Term locations Identification

Information from the eye-tracking tool, iTrace (Shaffer et al. 2015), includes information about the line number, the term name and the type of the term (method, variable, conditional statements, or other types). In some cases, the exact term that a participant looks at is not identified but the line number is. We now describe the process
of how we map eye gaze onto identify method signatures. First, for all methods used in the study, the signature lines are extracted from srcML (Collard et al. 2011). Then, if the line of an eye-tracking record matches the signature line in the list, then the record considered as signature.

The participants write their summaries in a small text file present at the bottom of the Eclipse IDE. The window also has the method signature of the current assigned method. During locating the assigned method at the beginning of the summarization task, participants may spend some time reading the signature of the assigned method in the text file where they were supposed to write their summaries. Therefore, the duration and number of visits spend at the signature line in the text file is also included in the signature durations.

The control flow statements are also identified at the line level. In the set of methods used in this study, lines that include control flow stand on their own and do not have multiple statements listed on the same line. i.e., the body of an if statement would appear on another line. This simplifies the line-level analysis.

Call terms need to be handled differently as the line containing a call might include other terms that are not part of the call. We define call terms to include the method call name and parameters (Rodeghero et al. 2014). For example, in the statement, `boolean saved = saveFile(toFile),` call terms include `saveFile` and `toFile`. If a programmer looks at `saveFile` or `boolean saved`, iTrace can specify the exact term does not identify the parameter `toFile` at the time the study was conducted.
Therefore, the possibility of identifying the eye-tracking records that belong to a call depends on the way that the call is written. All calls of our methods are manually examined to identify call cases that they can or cannot be properly identified. The total number of eye-tracking records inside the method for all participants is 43,283. The number of records that are successfully identified as calls is 16,127 (37%) and the number non-call records are 25,051 (58%). The number of eye-tracking records that cannot be categorized as calls or non-calls is 2,105 (5%) of all records. We exclude this 5% from the analysis. We conclude that the absence of these records in the analysis did not affect the validity of the results (see section 6.6.8).

6.6.2 Adjusted gaze time and visits

Number of terms in signature, call and control flow vary based on the size of the method. Therefore, it is mandatory to adjust the total gaze time according to the number of terms in each method. To compute the adjusted gaze time, the gaze time percentage of the signature is first computed. For example, if a programmer spends 100ms reading the method and 30ms of them reading the signature, then the percentage gaze time for the signature is (30/100= 30%). If the signature is 15% if the method size, then the adjusted gaze time is (30/15= 2). The same approach is used the adjusted visits for signature. The approach to compute the adjusted gaze time and number of visits for signature and non-signature terms is also applied to call terms vs. non-call terms, and control flow terms vs. non-control flow terms.
In addition to adjusting the gaze time and number of visits based on the number of terms (Rodeghero et al. 2014), they are also adjusted based on the number of characters. This assists determining if the significance of results is due to the complexity of the terms (number of characters) (Rodeghero et al. 2015) (Liblit, Begel, & Sweetser 2006) or the importance of terms in that location.

Finally, Wilcoxon signed-rank test (Wolfe & Hollander) is used to compare gaze time and number of visits because our data may not be normally distributed. The significance threshold is set at .05. The following demonstrates the result of Wilcoxon test result while considering three factors: developers’ level of expertise, size of the methods, and stereotype of methods.

6.6.3 RQ1: Three Source-code Locations

A total of 26 methods are summarized by five experts and 40 methods are summarized by 13 novices. The analysis of the eye-movements of experts and novices are discussed for signature, call, and control flow terms.

6.6.3.1 Signature

We found no evidence that experts read a method signature more than the method body. On average, the experts spent 9% of their gaze time reading signatures, while the signatures averaged 10% of the methods. Similarly, the average gaze time of novices reading signatures is 10% and signatures are 11% of the methods. We computed the adjusted gaze time and number of visits (see section 6.6.2) and post four hypotheses (H1, H2, H3, and H4) as follow $H_n$ for [experts / novices], the difference between the adjusted
[gaze time / visit] metric for a method signature and the method body is not statistically-significant.

Based on computed Z and p values in Table 26, we reject H$_2$, H$_3$, and H$_4$. This strongly suggest that novices read/visit method bodies more than the method signatures. We cannot reject H$_1$ which means that there is no statistical difference in the time spent by experts reading the signature of a method. However, experts revisit the method bodies more frequently than the method signatures. When data of both experts and novices are combined, the results of Wilcoxon test for gaze time and number of visits is (Z= -3.93, p < 0.001) and (Z= -5.37, p < 0.001), respectively.

Therefore, the result indicates that programmers read method bodies more heavily than the method signatures. This behavior was previously observed by Kevic et al. (Kevic et al. 2015).

6.6.3.2 Calls

We found strong evidence that programmers read calls of a method more than other area. On average, the programmers spent 44% of their gaze time reading calls, while they averaged 38% of the methods. We define the four hypotheses (H$_5$, H$_6$, H$_7$, and H$_8$) regarding call terms as the following: $H_n$ for [experts / novices], the difference between the adjusted [gaze time / visit] metric for call terms and non-terms terms is not statistically-significant.
Table 26. statistical summary of Wilcoxon Test of experts and novices. n is the number of samples. A sample is result from one programmer and one method. (*) indicates that p is significant.

<table>
<thead>
<tr>
<th>H</th>
<th>Metric</th>
<th>Location</th>
<th>n</th>
<th>T</th>
<th>Z</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>Gaze</td>
<td>Signature</td>
<td>69</td>
<td>905</td>
<td>-1.809</td>
<td>.064</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Sig</td>
<td>69</td>
<td>1510</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H2</td>
<td>Visit</td>
<td>Signature</td>
<td>69</td>
<td>623</td>
<td>-3.49</td>
<td>&lt;.0001*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Sig</td>
<td>69</td>
<td>1792</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H3</td>
<td>Gaze</td>
<td>Signature</td>
<td>144</td>
<td>3533</td>
<td>-3.36</td>
<td>.001*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Sig</td>
<td>144</td>
<td>6907</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H4</td>
<td>Visit</td>
<td>Signature</td>
<td>144</td>
<td>3101</td>
<td>-4.22</td>
<td>&lt;.0001*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Sig</td>
<td>144</td>
<td>7339</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H5</td>
<td>Gaze</td>
<td>Call</td>
<td>69</td>
<td>1655</td>
<td>-2.67</td>
<td>&lt;.007*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-call</td>
<td>69</td>
<td>760</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H6</td>
<td>Visit</td>
<td>Call</td>
<td>69</td>
<td>1554</td>
<td>-2.07</td>
<td>.03</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-call</td>
<td>69</td>
<td>861</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H7</td>
<td>Gaze</td>
<td>Call</td>
<td>139</td>
<td>7870</td>
<td>-5.28</td>
<td>&lt;.0001*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-call</td>
<td>139</td>
<td>2570</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H8</td>
<td>Visit</td>
<td>Call</td>
<td>139</td>
<td>7939</td>
<td>-5.28</td>
<td>&lt;.0001*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-call</td>
<td>139</td>
<td>2501</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H9</td>
<td>Gaze</td>
<td>Ctrl. Flow</td>
<td>69</td>
<td>1246</td>
<td>-1.13</td>
<td>.25</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Ctrl</td>
<td>69</td>
<td>899</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H10</td>
<td>Visit</td>
<td>Ctrl. Flow</td>
<td>69</td>
<td>1487</td>
<td>-2.71</td>
<td>.007*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Ctrl</td>
<td>69</td>
<td>658</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H11</td>
<td>Gaze</td>
<td>Ctrl. Flow</td>
<td>139</td>
<td>5339</td>
<td>-1.47</td>
<td>.13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Ctrl</td>
<td>139</td>
<td>3977</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H12</td>
<td>Visit</td>
<td>Ctrl. Flow</td>
<td>139</td>
<td>5961</td>
<td>-2.83</td>
<td>.005*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Ctrl</td>
<td>139</td>
<td>3355</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We reject all the four hypotheses (H5-H8) in Table 26. Both experts and novices read call terms more heavily than non-call terms when the gaze time and number of visits are adjusted based on the number of call terms. However, it was concluded that the longer a term (based on number of character), the longer it is viewed (Rodeghero et al. 2015). Therefore, gaze time and number of visits of experts are adjusted based on the number of characters. The gaze time ($Z = -.828, p = .4$) and number of visits ($Z = -.332, p = .47$) are not significant. This indicates that experts read call terms longer due to their length (number of characters). On the other hand, for novices, even when the gaze time and number of visits are adjusted based on the number of characters, the difference between
call terms and non-call terms are remain significant for both gaze time ($Z = -4.46, p < 0.001$) and number of visits ($Z = -3.45, p < 0.001$).

Therefore, novices spend more time reading call terms than experts. It might be worth mentioning that identifiers of all five systems that are used in the study are written using camel case.

### 6.6.3.3 Control Flow

We found no evidence that experts read control flow terms longer than other area but they visit them more frequently. On average, experts and novices spent 28% of their gaze time reading control flow terms, while they averaged 27% of the methods. The following presents the definition of the four hypotheses ($H_9$, $H_{10}$, $H_{11}$, and $H_{12}$).

$H_n$ for [experts / novices], the difference between the adjusted [gaze time / visit] metric for control-flow terms and control-flow terms is not statistically-significant.

We reject hypotheses ($H_{10}$, $H_{11}$, and $H_{12}$ ) in Table 26. This strongly suggest that experts revisit control flow terms but do not read them for a long period of time in each visit. On the other hand, novices read/visit control flow more heavily than experts.

For both novices and experts, the statistical differences of number of visits remain significant when the result of control flow terms are adjusted based on the number of characters. For experts, the results of the statistical result for gaze time and number of visits is ($Z = -1.82, p = 0.068$) and ($Z = -3.38, p = .001$), respectively. From novices, the statistical result of gaze time is ($Z = -2.47, p = 0.013$) and number of visits ($Z = -3.66, p < 0.001$).
From the given result, we can conclude that calls and control flow terms are read in different ways but we do not have evidence to order them in term of importance.

6.6.4 RQ2: Analysis Based on Methods’ Sizes

This section describes the result of gaze time and number of visits with respect to the method size. The size of large methods ranges between 40 to 80 LOC while medium methods are between 23 to 39 LOC. Finally, small methods are between 9 to 22 LOC. For each size category, the eye-movements record of signature, call, and control flow statements are analyzed.

**Expert:** The hypotheses of experts are described as the following: $H_n$ for [large / medium / small] methods, the difference between the adjusted [gaze time / visit] metric for [“signature and the body” / “calls and other areas” / “control flow and other areas”] is not statistically-significant.

Table 27. Wilcoxon Test Summary of experts for large, medium, and small methods. (*) indicates that $p$ is significant

<table>
<thead>
<tr>
<th>Metric</th>
<th>Location</th>
<th>Large</th>
<th>Medium</th>
<th>Small</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaze</td>
<td>Signature</td>
<td>$H_1$</td>
<td>$H_2$</td>
<td>$H_3$</td>
</tr>
<tr>
<td></td>
<td>Non-Sig</td>
<td>18</td>
<td>22</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td></td>
<td>49</td>
<td>95</td>
<td>183</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-1.59</td>
<td>-1.02</td>
<td>-.61</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.11</td>
<td>.30</td>
<td>.53</td>
</tr>
<tr>
<td>Visit</td>
<td>Signature</td>
<td>$H_4$</td>
<td>$H_5$</td>
<td>$H_6$</td>
</tr>
<tr>
<td></td>
<td>Non-Sig</td>
<td>18</td>
<td>22</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18</td>
<td>58</td>
<td>154</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-.94</td>
<td>-2.22</td>
<td>-1.22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.003*</td>
<td>.026*</td>
<td>.22</td>
</tr>
<tr>
<td>Gaze</td>
<td>Call</td>
<td>$H_7$</td>
<td>$H_8$</td>
<td>$H_9$</td>
</tr>
<tr>
<td></td>
<td>Non-call</td>
<td>18</td>
<td>22</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td></td>
<td>71</td>
<td>191</td>
<td>344</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-1.63</td>
<td>-2.09</td>
<td>-2.51</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.528</td>
<td>.03*</td>
<td>.012*</td>
</tr>
<tr>
<td>Visit</td>
<td>Call</td>
<td>$H_{10}$</td>
<td>$H_{11}$</td>
<td>$H_{12}$</td>
</tr>
<tr>
<td></td>
<td>Non-call</td>
<td>18</td>
<td>22</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>62</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-9.08</td>
<td>-1.70</td>
<td>-2.19</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.327</td>
<td>.08</td>
<td>.028*</td>
</tr>
<tr>
<td>Gaze</td>
<td>Ctrl Flow</td>
<td>$H_{13}$</td>
<td>$H_{14}$</td>
<td>$H_{15}$</td>
</tr>
<tr>
<td></td>
<td>Non-Ctrl</td>
<td>17</td>
<td>22</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td></td>
<td>102</td>
<td>170</td>
<td>167</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-1.20</td>
<td>-1.41</td>
<td>-.444</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.227</td>
<td>.15</td>
<td>.657</td>
</tr>
<tr>
<td>Visit</td>
<td>Ctrl Flow</td>
<td>$H_{16}$</td>
<td>$H_{17}$</td>
<td>$H_{18}$</td>
</tr>
<tr>
<td></td>
<td>Non-Ctrl</td>
<td>17</td>
<td>22</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td></td>
<td>118</td>
<td>200</td>
<td>192</td>
</tr>
<tr>
<td></td>
<td></td>
<td>118</td>
<td>53</td>
<td>159</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-1.96</td>
<td>-2.38</td>
<td>-.521</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.049*</td>
<td>.017*</td>
<td>.603</td>
</tr>
</tbody>
</table>

We reject the null hypotheses $H_2$, $H_4$, $H_9$, $H_{11}$, $H_{12}$, $H_{14}$, and $H_{16}$ in Table 27. This indicates that during summarizing a large or a medium method, experts revisit the method.
signature statistically less frequently than the body. No statistical different found in case of small methods. We interpret this observation as the following. As the size of a method increases, experts visit method bodies more often than method signatures.

Control flow terms are revisited significantly in case of large and medium methods but not significant in term of small methods. The percentage of experts who rated the easiness of tasks to be easy, large methods (61%) and medium (81%) and small (80%). This implies that the harder the method, the longer time is spent reading control flow by experts.

Call terms are read significantly in case of small and medium methods but not significant in term of large methods.

**Novices:** The hypotheses of novices are described as the following: $H_n$ for [large / medium / small] methods, the difference between the adjusted [gaze time / visit] metric for [“signature and the body” / “calls and other areas” / “control flow and other areas”] is not statistically-significant.

**Table 28. Wilcoxon Test Summary of novices for large, medium, and small methods.**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Location</th>
<th>H</th>
<th>n</th>
<th>T</th>
<th>Z</th>
<th>p</th>
<th>H</th>
<th>n</th>
<th>T</th>
<th>Z</th>
<th>p</th>
<th>H</th>
<th>n</th>
<th>T</th>
<th>Z</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaze</td>
<td>Signature</td>
<td>36</td>
<td>271</td>
<td>-97</td>
<td>.33</td>
<td>.02</td>
<td>68</td>
<td>657</td>
<td>-3.15</td>
<td>.002*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Non-Sig</td>
<td>36</td>
<td>395</td>
<td></td>
<td>.01</td>
<td>.93</td>
<td>68</td>
<td>1689</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visit</td>
<td>Signature</td>
<td>36</td>
<td>240</td>
<td>-1.46</td>
<td>.14</td>
<td>.96</td>
<td>68</td>
<td>614</td>
<td>-3.41</td>
<td>.001*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Non-Sig</td>
<td>36</td>
<td>426</td>
<td></td>
<td>.16</td>
<td>.99</td>
<td>68</td>
<td>1732</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gaze</td>
<td>Call</td>
<td>36</td>
<td>437</td>
<td>-1.63</td>
<td>.10</td>
<td>.96</td>
<td>68</td>
<td>1811</td>
<td>-3.89</td>
<td>&lt;.0001*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Non-call</td>
<td>36</td>
<td>229</td>
<td></td>
<td>.46</td>
<td>.65</td>
<td>68</td>
<td>535</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visit</td>
<td>Call</td>
<td>36</td>
<td>441</td>
<td>-1.69</td>
<td>.09</td>
<td>.96</td>
<td>68</td>
<td>1818</td>
<td>-3.94</td>
<td>&lt;.0001*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Non-call</td>
<td>36</td>
<td>225</td>
<td></td>
<td>.46</td>
<td>.65</td>
<td>68</td>
<td>528</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gaze</td>
<td>Ctrl. Flow</td>
<td>32</td>
<td>309</td>
<td>-2.81</td>
<td>.06</td>
<td>.01</td>
<td>64</td>
<td>1933</td>
<td>-1.62</td>
<td>.10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Non-Ctrl</td>
<td>32</td>
<td>219</td>
<td>-2.81</td>
<td>.06</td>
<td>.01</td>
<td>64</td>
<td>887</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visit</td>
<td>Ctrl. Flow</td>
<td>32</td>
<td>536</td>
<td>-1.34</td>
<td>.12</td>
<td>.98</td>
<td>64</td>
<td>1331</td>
<td>-1.94</td>
<td>.052</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Non-Ctrl</td>
<td>32</td>
<td>192</td>
<td>-2.10</td>
<td>.035*</td>
<td></td>
<td>64</td>
<td>749</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We reject two null hypotheses $H_5, H_6, H_9, H_{10}, H_{11}, H_{12}$, and $H_{16}$ in Table 28. Novices spent a significant amount of time reading calls but less time when the method is getting larger. The number of visits of control flow is significant in case of medium methods only. The percentage of novices who rated the easiness of tasks to be easy, large methods (60%) and medium (65%) and small (50%). This may indicate that novices visit them more often than other areas because medium methods are the hardest.

6.6.5 RQ3: Analysis Based on Method Stereotypes

Methods in the study are selected from different stereotypes to ensure diversity and to study the role of a method stereotype on what the programmer looks at. A predicate method does not change the object state and returns a Boolean value about the data member(s) or other variables. The main work of a predicate is found in control flow statements by checking the existence of an element, or evaluating a value for being in a certain range. As developers tend to focus on certain locations during comprehension of source code (Bednarik & Tukiainen 2006; Crosby & Stelovsky 1990), we investigate if control flow statements receive higher attention in case of predicates more than other stereotypes. Furthermore, the previous analysis showed that call terms are read heavily than other locations. Here, we study if the results from call terms remain significant across stereotypes.

Method stereotypes that have a small sample size (<5) are not considered as the sample size may not be big enough to provide accurate and generalizable conclusions. All methods presented are stereotyped as collaborators as they work with external objects that is a return
type, parameter, or data member. Through the analysis, we refer to all methods with their first stereotype. For example, “predicate collaborators” is called “predicates”.

6.6.5.1 Predicate

As stated above, the main work of predicates is mostly found in conditional or control flow statements. Therefore, we investigate the amount of attention that control flow statements receive by developers. One method is summarized by 4 experts. The result does not show any statistical difference. However, the average time of experts reading control flow statements is 45% where 44% of the method averages control flow statements, suggesting that control flow is read in proportion to their method composition.

Table 29. Wilcoxon Test Summary of novices for predicate methods. (*) indicates that p is significant

<table>
<thead>
<tr>
<th>H</th>
<th>Metric</th>
<th>Location</th>
<th>n</th>
<th>T</th>
<th>Z</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>H₁</td>
<td>Duration</td>
<td>Call</td>
<td>13</td>
<td>70</td>
<td>-2.13</td>
<td>.03 *</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-call</td>
<td>13</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H₂</td>
<td>Visit</td>
<td>Call</td>
<td>13</td>
<td>70</td>
<td>-2.48</td>
<td>.01 *</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-call</td>
<td>13</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H₃</td>
<td>Duration</td>
<td>Ctrl. Flow</td>
<td>13</td>
<td>11</td>
<td>-2.41</td>
<td>.016*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Ctrl</td>
<td>13</td>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H₄</td>
<td>Visit</td>
<td>Ctrl. Flow</td>
<td>13</td>
<td>14</td>
<td>-2.20</td>
<td>.028*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Ctrl</td>
<td>13</td>
<td>77</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

On the other hand, on average, control flow statements are read by novices 33% of the time, although they are 39% of methods’ composition. This implies that novices read control flow statements of predicates less than other areas in the method. To further investigate this case, we define four hypotheses H₁-H₄ in Table 29 as follows: H₄. In case of novices and predicate methods, the difference between the adjusted [duration/visit] for [calls and other areas/control flow and other areas] is not statistically significant. The null
hypotheses $H_1$ and $H_2$ are rejected indicating that call terms of predicate methods are read more than other terms by novices. However, $H_3$ and $H_4$ are rejected indicating that control flow are read less than other areas in the method.

Furthermore, summarizing *predicates* is found to be easy by 67% of novices and neither easy nor hard by 20%. We manually examined these methods and we found that they have one or two if-else statements and/or loop-if (Wang et al. 2015) that are found easy to comprehend by novices. Using the explained observations, we suggest that control flow statements are read somewhat less than or same as other areas because they were somewhat straightforward. We study this assumption in later stereotypes.

### 6.6.5.2 Property

*Property* methods do not change the object state and return a computed value based on data member(s) and other variables. For property methods, we define the eight null hypotheses as follow: $H_n$: In case of [experts/novices] and *property* methods, the difference between the adjusted [duration/visit] metric for [signature and body/calls and other areas/control flow and other areas] is not statistically significant.

The four null hypotheses $H_1$-$H_4$ regarding call terms are rejected in Table 30. This indicates that call terms of property methods are read more heavily than other terms by both experts and novices. However, no statistical difference is found in case of control flow statements. Furthermore, 90% of experts rated the summarization tasks of *property* methods to be easy and 5% of experts rated them to be neither easy nor hard. While 57%
of novices rated summarizing these methods to be easy and 28% stated they were neither easy nor hard.

Although almost all experts and majority of novices found the tasks to be easy, the gaze duration and number of visits are significant for calls. Control flow terms are not read more heavily than other areas.

**Table 30. Wilcoxon Test Summary of experts and novices for property methods.** (*) indicates that p is significant

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>Metric</th>
<th>Location</th>
<th>n</th>
<th>T</th>
<th>Z</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experts</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$H_i$</td>
<td>Duration</td>
<td>Call</td>
<td>20</td>
<td>175</td>
<td>-2.61</td>
<td>.009*</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-call</td>
<td>20</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$H_2$</td>
<td>Visit</td>
<td>Call</td>
<td>20</td>
<td>166</td>
<td>-2.27</td>
<td>.02*</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-call</td>
<td>20</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Novices</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$H_3$</td>
<td>Duration</td>
<td>Call</td>
<td>40</td>
<td>750</td>
<td>-2.15</td>
<td>.03*</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-call</td>
<td>40</td>
<td>250</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$H_4$</td>
<td>Visit</td>
<td>Call</td>
<td>40</td>
<td>582</td>
<td>-2.31</td>
<td>.02*</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-call</td>
<td>40</td>
<td>238</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Experts</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$H_5$</td>
<td>Duration</td>
<td>Ctrl. Flow</td>
<td>20</td>
<td>69</td>
<td>-1.34</td>
<td>.17</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Ctrl</td>
<td>20</td>
<td>141</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$H_6$</td>
<td>Visit</td>
<td>Ctrl. Flow</td>
<td>20</td>
<td>122</td>
<td>-63</td>
<td>.52</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Ctrl</td>
<td>20</td>
<td>88</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Novices</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$H_7$</td>
<td>Duration</td>
<td>Ctrl. Flow</td>
<td>40</td>
<td>476</td>
<td>-88</td>
<td>.37</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Ctrl</td>
<td>40</td>
<td>344</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$H_8$</td>
<td>Visit</td>
<td>Ctrl. Flow</td>
<td>40</td>
<td>512</td>
<td>-1.37</td>
<td>.17</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-Ctrl</td>
<td>40</td>
<td>308</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 6.6.5.3 Void-accessor

*Void-accessor* methods do not change the object state and return information through parameters. Similar to the above cases, we post the null hypotheses as following: $H_n$: In case of [experts/novices] and *void-accessor* methods, the difference between the adjusted [duration/visit] metric for [calls and other areas/control flow and other areas] is not statistically significant.
The four null hypotheses $H_1$-$H_4$ in Table 31 cannot be rejected indicating that call terms of void-accessor methods are not read more than other terms by both experts and novices. However, the two hypotheses ($H_6$, and $H_8$) are rejected. Results indicate that control flow statements are statistical read more the other areas. In particular, experts and novices revisit control flow terms statistically more heavily than other areas, while novices read them for a significant number of times. 30% of experts rated these tasks to be easy and 10% of experts stated they were neither easy nor hard. 31% of novices found these methods to be easy to summarize while 23% found them to be neither easy nor hard. Furthermore, we manually investigated these methods and they have a complex flow of nested-if’s and loop statements.

Table 31. Wilcoxon Test Summary of experts and novices for void-accessor methods. (*) indicates that $p$ is significant

<table>
<thead>
<tr>
<th></th>
<th>$H$</th>
<th>Metric</th>
<th>Location</th>
<th>n</th>
<th>T</th>
<th>Z</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experts</td>
<td>$H_1$</td>
<td>Duration</td>
<td>Call</td>
<td>10</td>
<td>26</td>
<td>-.15</td>
<td>.87</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-call</td>
<td>10</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$H_2$</td>
<td>Visit</td>
<td>Call</td>
<td>10</td>
<td>30</td>
<td>-.25</td>
<td>.79</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-call</td>
<td>10</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Novices</td>
<td>$H_3$</td>
<td>Duration</td>
<td>Call</td>
<td>13</td>
<td>66</td>
<td>-1.43</td>
<td>.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-call</td>
<td>13</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$H_4$</td>
<td>Visit</td>
<td>Call</td>
<td>13</td>
<td>69</td>
<td>-1.64</td>
<td>.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-call</td>
<td>13</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Experts</td>
<td>$H_5$</td>
<td>Duration</td>
<td>Ctrl. Flow</td>
<td>10</td>
<td>40</td>
<td>-1.27</td>
<td>.20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-Ctrl</td>
<td>10</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$H_6$</td>
<td>Visit</td>
<td>Ctrl. Flow</td>
<td>10</td>
<td>49</td>
<td>-2.19</td>
<td>.03 *</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-Ctrl</td>
<td>10</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Novices</td>
<td>$H_7$</td>
<td>Duration</td>
<td>Ctrl. Flow</td>
<td>13</td>
<td>71</td>
<td>-1.78</td>
<td>.08</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-Ctrl</td>
<td>13</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$H_8$</td>
<td>Visit</td>
<td>Ctrl. Flow</td>
<td>13</td>
<td>81</td>
<td>-2.48</td>
<td>.013 *</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-Ctrl</td>
<td>13</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The above observations strongly suggest that the harder the method (complex control flow), the longer control flow terms are viewed. On the other hand, call terms tend not to
be affected by the difficulty of the task. Call terms were viewed longer in a set of easy methods while less viewed in harder ones.

6.6.5.4 Command

*Command* methods change the object states; via another call. To study how much call terms and flow terms are read in this set of methods, we study the following hypotheses: $H_n$. In case of [experts/novices] and *command* methods, the difference between the adjusted [duration/visit] metric for [calls and other areas/control flow and other areas] is not statistically significant.

**Table 32. Wilcoxon Test Summary of experts and novices for command methods. (*) indicates that p is significant**

<table>
<thead>
<tr>
<th></th>
<th>Metric</th>
<th>Location</th>
<th>n</th>
<th>T</th>
<th>Z</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Experts</strong></td>
<td>$H_1$</td>
<td>Duration</td>
<td>Call</td>
<td>25</td>
<td>179</td>
<td>-0.44</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-call</td>
<td>25</td>
<td>146</td>
<td>-0.52</td>
</tr>
<tr>
<td></td>
<td>$H_2$</td>
<td>Visit</td>
<td>Call</td>
<td>25</td>
<td>182</td>
<td>-0.52</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-call</td>
<td>25</td>
<td>143</td>
<td>-0.52</td>
</tr>
<tr>
<td><strong>Novices</strong></td>
<td>$H_3$</td>
<td>Duration</td>
<td>Call</td>
<td>42</td>
<td>725</td>
<td>-2.68</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-call</td>
<td>42</td>
<td>265</td>
<td>-2.68</td>
</tr>
<tr>
<td></td>
<td>$H_4$</td>
<td>Visit</td>
<td>Call</td>
<td>42</td>
<td>681</td>
<td>-2.17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-call</td>
<td>42</td>
<td>309</td>
<td>-2.17</td>
</tr>
<tr>
<td></td>
<td>$H_5$</td>
<td>Duration</td>
<td>Ctrl. Flow</td>
<td>21</td>
<td>149</td>
<td>-1.16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-Ctrl</td>
<td>21</td>
<td>82</td>
<td>-1.16</td>
</tr>
<tr>
<td></td>
<td>$H_6$</td>
<td>Visit</td>
<td>Ctrl. Flow</td>
<td>21</td>
<td>152</td>
<td>-1.26</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-Ctrl</td>
<td>21</td>
<td>79</td>
<td>-1.26</td>
</tr>
<tr>
<td><strong>Novices</strong></td>
<td>$H_7$</td>
<td>Duration</td>
<td>Ctrl. Flow</td>
<td>36</td>
<td>363</td>
<td>-0.47</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-Ctrl</td>
<td>36</td>
<td>303</td>
<td>-0.47</td>
</tr>
<tr>
<td></td>
<td>$H_8$</td>
<td>Visit</td>
<td>Ctrl. Flow</td>
<td>36</td>
<td>418</td>
<td>-1.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-Ctrl</td>
<td>36</td>
<td>248</td>
<td>-1.33</td>
</tr>
</tbody>
</table>

All eight null hypotheses of *command* cannot be rejected. This means that call terms and control flow terms are not read significantly differently as they occurred in the methods. 76% of experts rated the summarization tasks of *command* methods to be easy and 12% of experts rated them to be neither easy nor hard. 62% of novices rated
summarizing these methods to be easy and 24% stated that they were neither easy nor hard. As the majority of experts and novices found the tasks to be easy, the gaze duration and number of visits are not significant for both calls and control flow terms.

6.6.6 Discussion

The previous literature suggested that the words/terms that programmers spend more time reading are more important for comprehension (Crosby & Stelovsky 1990) (Rodeghero et al. 2014). Taking this on to consideration, we discuss how experts and novices read some areas over others with respect of the developers’ level of expertise and methods’ sizes.

With statistical evidence, the result reveals that experts and novices focus on the method bodies more than the methods’ signatures. Furthermore, focusing on signatures of methods seem to be decreased as methods’ sizes increase in the case of experts. This relationship is not found in the case of novices.

For both experts and novices, call terms are read the same proportion they occur or higher. They do not seem to be affected by the stereotype of a method and they are never read less than other areas of the method.

On the other hand, there is a statistical difference on how experts and novices read control flow terms of a method. Experts revisit control flow terms multiple times but do not fixate on them for a long period of time. While novices read/visit control terms heavily. However, the way that novices read control flow statements in a method does not seem to be related to the size of the method nor its stereotype.
Similarly, the way experts read control flow statements in a method is not related to the method’s stereotype. For example, in property methods, experts and novices focus on call terms more than other areas while control flow terms are read more in void-accessor methods. This observation seems to be related to the task difficulty. The difficulty of tasks is self-reported by the developers. We notice that experts’ adjusted gaze duration and number of visits increase as the difficulty of tasks increase while calls terms do not seem to be affected by the difficulty of tasks. Through manual inspection of the source code, we observe that when a method has a complex structure that involves many nested-ifs and loops, developers rate the task of summarizing that method to be hard. This also shows that experts read complex control flow more heavily because it is hard to comprehend. On the other hand, novices’ adjusted gaze duration and number of visits do not appear to be affected by the task difficulty for call terms and control flow terms.

6.6.7 Comparison to Rodeghero et al.’s Study

Our result does not reproduce Rodeghero et al.’s result as shown in Table 33. Rodeghero et al. found that developers read the signature more significantly than other areas while we found that signature is read the least compare to other areas. Our result reproduce the result observed by (Kevic et al. 2016). We explain this difference as the following. First, in our study, developers read methods inside context rather than in isolated (Rodeghero et al.’s settings). Second, we include methods that are much larger than Rodeghero et al.’s study. We found statistical evidence that the larger the method, the less experts fixate or revisit the signature. These results are discussed in section 6.6.4.
Table 33. A comparison of our result to Rodeghero et al.

<table>
<thead>
<tr>
<th>[Rodeghero et al.’14]</th>
<th>Our Study</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td>Call invocations</td>
</tr>
<tr>
<td>Call invocations</td>
<td>Control flow</td>
</tr>
<tr>
<td>Control flow</td>
<td>Signature</td>
</tr>
</tbody>
</table>

The result of call terms does not reproduce Rodeghero et al.’s findings although developers are performing the same comprehension tasks. Furthermore, our statistical analysis does not generate Rodeghero et al.’s result regarding control flow (Rodeghero et al. 2014). It was concluded that control flow terms are read statistically less heavily than other area of a method. As stated earlier, the complexity of the flow of conditional statements statistically increases the gaze time and number of visits on these locations. Using a limited method size (22 LOC) can be the reason behind their results. While our result has 20 methods that are larger than 22 LOC.

6.6.8 Threats to Validity

When a developer writes a summary for a method from a class, he/she may build some knowledge about this class as they can scroll up and down to read other methods. This might affect the time and the effort to understand later methods from the same class. To mitigate this, for one system, each developer should summarize methods from different classes.

To reduce the overhead for browsing so many systems, we limit the number of systems that each developer works with to three systems.
As developers mostly rely on comments to understand methods (Crosby & Stelovsky 1990), we remove comments from the entire system. This is important as the goal of the study is to examine the locations (source code statements) that developers focus on when they write their own summaries.

Syntax highlighting preferences and distractions. Developers have different IDE environment preferences that might affect their performances. This possible threat is not mitigated because the eye-tracking tool and software require using Eclipse.

As we stated earlier, 5% of the eye-tracking records cannot be determined to be calls or non-call terms. 519 records belong to experts and they are 4.5% of experts records while 1969 records belong to novices and they are 5% of their records. To test if this can affect our result, these records are added to non-calls data and we reapply the Wilcoxon test. The result of gaze time and number of visits of call terms remains significant when this applied for novices as shown in Table 34. The result is significant in case of experts. This indicates that the 5% records do not impact the validity of the result of call terms in novices’ data but not in experts’ result.

Table 34. Wilcoxon Test Summary of experts and novices After adding the non-identified records to non-call terms. (*) indicates that p is significant

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>Metric</th>
<th>Location</th>
<th>n</th>
<th>T</th>
<th>Z</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experts</td>
<td>H1</td>
<td>Gaze</td>
<td>Call</td>
<td>69</td>
<td>1509</td>
<td>-1.80</td>
<td>.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-call</td>
<td>69</td>
<td>906</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>H2</td>
<td>Visit</td>
<td>Call</td>
<td>69</td>
<td>1451</td>
<td>-1.45</td>
<td>.145</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-call</td>
<td>69</td>
<td>964</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Novices</td>
<td>H4</td>
<td>Gaze</td>
<td>Call</td>
<td>144</td>
<td>7045</td>
<td>-3.64</td>
<td>&lt;.0001*</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-call</td>
<td>144</td>
<td>3395</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>H5</td>
<td>Visit</td>
<td>Call</td>
<td>144</td>
<td>7141</td>
<td>-3.83</td>
<td>&lt;.0001*</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Non-call</td>
<td>144</td>
<td>3299</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.7 Analyzing Eye-tracking data at Line Level

The ultimate goal of studying source-code lines that developers read is to improve automatic documentation/summarization. Therefore, we examine what lines that participant read or use to formulate their final summaries. The hypothesis is that not all lines are important for summarization. However, a line with long gaze time may reflect importance or complexity (e.g., large number of identifiers). To this end, we propose the following questions are:

RQ4) Does the length of a line (number of identifiers) affect the gaze time spent on the line?

RQ5) To what extent do experts focus on lines that they use to write their summaries?

RQ6) To what degree lines used by experts relate to the lines that the literature on summarization explained in section 2.2?

RQ4 studies the relationship between the size of a line and its gaze time. We perform this analysis to decide whether the gaze time spent in each line should be adjusted based on the number of identifiers. To answer RQ5, we manually analyze experts’ summaries in order to understand what developers seek in a method summary. In particular, we determine source-code lines that used by an expert to write the summary for a method. Then, we investigate whether the gaze time of an expert can predict (or speculate) lines used by the expert to summarize a method. Finally, RQ6 scrutinizes if some of lines used
by experts are similar to the lines proposed in the previous literature. The following section demonstrates how the duration of each line is computed.

### 6.7.1 Gaze time of source-code lines

Source-code lines that developers read during summarization are analyzed. For each source-code line read by a developer, the total gaze time and number of fixations is computed. The gaze time of a line is the total amount of time spent reading any terms in that line. The number of samples (a sample is data from one developer and one method) is 69 samples from experts and 144 samples from novices.

The total number of lines that experts read is 1154 lines (the duration is greater than zero) that is found in 69 samples. The total number of lines that novices read is 2347 lines (the duration is greater than zero) that is found in 144 samples. Using a well-known clustering algorithm, k-mean (Aldenderfer & Blashfield 1985), these lines are clustered according to their gaze time. Furthermore, gaze time of each line is normalized based on the total gaze time. For example, if the gaze time of a line is 300 milliseconds and the total gaze time for the method is 1000 milliseconds, then we conclude that the participant spent \(\frac{300}{1000} = 30\%\) of his/her reading time to read that line.

Of the 69 samples, two samples are removed because they contain two abnormal outliers. In sample one, the developer spends 80% of their time reading one line. In the second removed sample, the developer spends 61% of their time reading one line. Although we keep these samples for later analysis, we decide to remove these two cases here as they may affect the performance of k-mean algorithm.
Similarly, of the 144 novices’ samples, two samples are removed because they contain three abnormal outliers. In two samples, the developers spend 61% of their time reading one line. In the third removed sample, the developer spends 60% of their time reading one line. Table 35 and

Table 36 present the k-mean clusters of experts lines when k is equal 2 and 3, respectively. We conclude that there is a few number of lines that are read for a long period while the majority of lines are read for medium or short amount of time. Consider the three clusters in Table 36. Experts spent between 6.99% to .08% of their time reading 71% of lines while 5% of lines are read for more than 18.84% of an expert’s time. This suggests that not all lines are important for summarization. However, these lines might be important or even harder to read. We discuss these cases later in section 6.7.3 and 6.7.4.

Table 37 and Table 38 present the k-mean clusters of novices lines when k is equal 2 and 3, respectively. Consider the three clusters in Table 38 novices spent between 7.7% to .04% of their time reading 72% of lines while 4% of lines are read for more than 19% of a novice’s time.

Table 35. K-mean clusters of experts’ lines where k = 2.

<table>
<thead>
<tr>
<th>Group ID</th>
<th>Cluster range</th>
<th>Type of lines</th>
<th>total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>max</td>
<td>min</td>
<td>Signature</td>
<td>flow</td>
</tr>
<tr>
<td>1</td>
<td>48.67</td>
<td>10.0</td>
<td>17 (20%)</td>
<td>14 (11%)</td>
</tr>
<tr>
<td>2</td>
<td>10.1</td>
<td>0.08</td>
<td>71 (80%)</td>
<td>113 (89%)</td>
</tr>
<tr>
<td>Total</td>
<td>88</td>
<td>127</td>
<td>757</td>
<td>176</td>
</tr>
</tbody>
</table>

Table 36. K-mean clusters of experts’ lines where k = 3.

<table>
<thead>
<tr>
<th>Group ID</th>
<th>Cluster range</th>
<th>Type of lines</th>
<th>total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>max</td>
<td>min</td>
<td>Signature</td>
<td>flow</td>
</tr>
<tr>
<td>1</td>
<td>45.79</td>
<td>18.84</td>
<td>5 (6%)</td>
<td>5 (4%)</td>
</tr>
<tr>
<td>2</td>
<td>18.48</td>
<td>7.02</td>
<td>16 (18%)</td>
<td>29 (23%)</td>
</tr>
<tr>
<td>3</td>
<td>6.99</td>
<td>0.08</td>
<td>67 (76%)</td>
<td>93 (73%)</td>
</tr>
</tbody>
</table>

115
Table 37. K-mean clusters of novices’ lines where k = 2.

<table>
<thead>
<tr>
<th>Group ID</th>
<th>Cluster range</th>
<th>Type of lines</th>
<th>total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>max</td>
<td>min</td>
<td>Signature</td>
<td>flow</td>
</tr>
<tr>
<td>1</td>
<td>55.78</td>
<td>11</td>
<td>17 (11%)</td>
<td>48 (17%)</td>
</tr>
<tr>
<td>2</td>
<td>10.8</td>
<td>0.04</td>
<td>137 (89%)</td>
<td>243 (83%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>154</td>
<td>291</td>
<td>1534</td>
<td>349</td>
</tr>
</tbody>
</table>

Table 38. K-mean clusters of novices’ lines where k = 3.

<table>
<thead>
<tr>
<th>Group ID</th>
<th>Cluster range</th>
<th>Type of lines</th>
<th>total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>max</td>
<td>min</td>
<td>Signature</td>
<td>flow</td>
</tr>
<tr>
<td>1</td>
<td>55.78</td>
<td>19</td>
<td>7 (5%)</td>
<td>8 (3%)</td>
</tr>
<tr>
<td>2</td>
<td>20.3</td>
<td>7.0</td>
<td>17 (11%)</td>
<td>71 (24%)</td>
</tr>
<tr>
<td>3</td>
<td>7.7</td>
<td>0.04</td>
<td>130 (84%)</td>
<td>212 (73%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>154</td>
<td>291</td>
<td>1534</td>
<td>349</td>
</tr>
</tbody>
</table>

We conclude that there are a small number of lines that read for a long period while the majority of lines are read for medium or short amount of time. This observation applies to all lines regardless to the line type. On another word, the cluster with the smallest rage (e.g., cluster two in Table 35) has the largest number of lines for all line types.

Finally, the only notable difference between novices and experts appear on the case of signature. Experts tend to spend a long period of time on signatures (21% in Table 35) compare to novices (11% in Table 37). Finally, the k-mean clusters of novices and experts lines are presented in Appendix B. The value of k ranges between 5 to 10.

6.7.2 Heat Map

A common graphical representation of eye-tracking data is heat map that uses color-coding to demonstrate the reading effort of an element on the screen. Here, we generate the heat map for each participant and each method; an example is shown in Figure 22.
six levels of gaze time are: longest gaze time (red lines), long gaze time (orange lines), medium gaze time (yellow lines), short gaze time (light green lines), shortest gaze time but not zero (green lines), and zero lines (un-colored lines).

![Code Snippet]

Figure 22. Heat map generated for one method and one participant.

In Figure 22, the line that read for the longest period is line number one where the participant spends 21% of gaze time; therefore, it is classified as a red line. The second most viewed lines are 17 with the gaze time percentage is 18% and 13 with the gaze time percentage is 8% These lines are orange lines. We come up with a set of observations based on the heat map of all the 69 samples are generated (see Table 39). First, on average, experts completely ignored 28% (SD = 22%) of a method. A great deal of these lines are part of catch block that might be found not important for a summarization task. Second, the average of number of lines with short (light green) and medium (yellow) gaze time with average 19% (SD = 13%) and 18% (SD = 11%) of a method, respectively. Third, the
average number of lines with long (orange) and the longest (red) gaze time lines is 10% (SD = 10%) and 10% (SD = 7%), respectively. The same steps are repeated for all the 144 novices’ samples and the average number of lines in each case is very similar to experts. We interpret this as experts and novices tend to scan through lines of a method and focus only on a few lines. Finally, the majority of longest gaze time are variable declarations and calls. A similar observation is reported by (Kevic et al. 2016).

Table 39. The average number of lines on each Heat map cotogery for experts and novices.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Zero and empty</th>
<th>Zero and not empty</th>
<th>Green</th>
<th>Light green</th>
<th>Yellow</th>
<th>Orange</th>
<th>Red</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experts</td>
<td>Average</td>
<td>33%</td>
<td>28%</td>
<td>16%</td>
<td>19%</td>
<td>18%</td>
<td>10%</td>
</tr>
<tr>
<td></td>
<td>SD</td>
<td>14%</td>
<td>22%</td>
<td>11%</td>
<td>13%</td>
<td>11%</td>
<td>10%</td>
</tr>
<tr>
<td>Novices</td>
<td>Average</td>
<td>32%</td>
<td>30%</td>
<td>15%</td>
<td>18%</td>
<td>17%</td>
<td>9%</td>
</tr>
<tr>
<td></td>
<td>SD</td>
<td>17%</td>
<td>19%</td>
<td>10%</td>
<td>10%</td>
<td>11%</td>
<td>9%</td>
</tr>
</tbody>
</table>

6.7.3 RQ4: Gaze Time VS. Line’s Length

15% of lines with zero durations are lines with no identifiers (e.g., return false; or else) as no higher level information can be obtained from them. However, a high number of identifiers may not lead directly to a long gaze time. Consider the method in Figure 22, although line 18 is the shortest line in the method, this participant spend about 3832 milliseconds which is 6% of the total gaze time. On the other hand, longer lines have shorter gaze time such as 6 and 11 that their gaze time is 933 milliseconds and 1832 milliseconds, respectively.

In this section, we seek to examine this phenomenon. We recapitulate the study as the following. First, for each data sample, according to the gaze time, lines are sorted in
decreasing order (from largest to smallest). Therefore, the line with the largest gaze time
has a rank equals to one and the line with the lowest gaze time has a rank equals to \( n \), where
\( n \) is the number of lines. Second, each line is assigned another rank based on the number
of identifiers. The line with the largest number of identifiers has a rank equals to one and
the line with the smallest number of identifiers has a rank equals to \( n \).

Finally, we compare the two ranked sets using Kendall rank correlation coefficient or
Kendall’s tau coefficient. Kendall’s tau evaluates the degree of concordance between two
sets of ranked data (Siegel 1956) (Abdi 2007) which is the suitable test in this case. If the
Kendall’s tau is 1, then the two lists are identical while -1 means that one is the reverse of
the other. However, the critical value of Kendall’s tau depends on \( n \) (in our case it is the
number of lines in each sample). For \( \alpha \leq 0.05 \) and \( Z \geq 1.96 \), \( \tau \) should be greater than or
equal .4667 if \( n \) is greater than or equal 10. There are seven samples where the number of
lines that participants looked at is less than 10. Therefore, different \( \tau \) is applied as the
following: \( \tau \) is greater than or equal .6667 if \( n = 9 \), \( \tau \) is greater than or equal .7143 if \( n = 8 \),
and \( \tau \) is equal 1 if \( n = 5 \). Finally, there is a sample where the participant looked at three
lines only, although the \( \tau \) is equal 1, we exclude this sample from the analysis because \( n \) is
not large enough (Abdi 2007).

The Tau-a test is used that is computed as:

\[
\tau = \frac{(\text{number of concordant pairs} - \text{number of discordant pairs})}{(\text{number of concordant pairs} + \text{number of discordant pairs})}
\]

The Z value is computed as:
\[
Z = \frac{3(n_c-n_d)}{\sqrt{n(n-1)(2n+5)/2}}
\]

The result of the Kendall’s tau for novices is also shown in Table 40. This analysis is based on 144 samples from 40 methods and 13 novices. Similar to experts, in novices’ data, the biggest set (70%) suggest that there is an insignificant positive correlation between the gaze time of a line and number of identifiers. Therefore, we conclude that these is a weak positive relationship between the number of identifiers in a line and its gaze time.

Table 40. A summary of the Kendall’s tau comparing the set of ranks based on the gaze time and the set of ranks based on the number of identifiers.

<table>
<thead>
<tr>
<th>Case</th>
<th>Experts</th>
<th></th>
<th></th>
<th>Novices</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of samples</td>
<td>Percentage</td>
<td>Number of samples</td>
<td>Percentage</td>
<td></td>
</tr>
<tr>
<td>Significant agreement</td>
<td>6</td>
<td>9%</td>
<td>14</td>
<td>10%</td>
<td></td>
</tr>
<tr>
<td>Insignificant agreement</td>
<td>50</td>
<td>73%</td>
<td>99</td>
<td>70%</td>
<td></td>
</tr>
<tr>
<td>Insignificant disagreement</td>
<td>0</td>
<td>0%</td>
<td>2</td>
<td>1%</td>
<td></td>
</tr>
<tr>
<td>Significant disagreement</td>
<td>12</td>
<td>18%</td>
<td>28</td>
<td>19%</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>68</td>
<td>100%</td>
<td>144</td>
<td>100%</td>
<td></td>
</tr>
</tbody>
</table>

To further examine this, we perform a similar analysis by comparing the rank of the actual gaze time and the rank of adjusted gaze time based on the number of identifiers. If a participant spent 100 milliseconds reading a line and the line has 4 identifiers, then the adjusted duration is 100/4 = 25 milliseconds. Then, the list of the original duration is compared to the adjusted duration using Kendall's tau coefficient. Table 41 presents the summary of the Kendall’s tau computed to compare the set of ranks of the actual gaze time and the set of ranks of the adjusted gaze time based on the number of identifiers.
When lines are adjusted based on number of identifiers, the order of lines are very similar to the original order in 90% of the samples. In the remaining 10% the order is similar but with weak relationship. This strongly suggest that the length of a line does not significantly impact the amount of time spent reading the line.

Similarly, for novices, when lines are adjusted based on number of identifiers, the order of lines are very similar to the original order in 88%. In the remaining 11% the order is similar but with weak relationship. This strongly suggest that the length of a line does not significantly impact the amount of time spent reading the line.

Despite the weak proportional relationship between the number of identifiers and the gaze time that is found in Table 40, this relationship does not significantly influence the analysis as shown in Table 41. Consequently, we can use the actual gaze time or the adjusted gaze time. We decided to use the actual gaze time for the rest of the analysis.

Table 41. A summary of the Kendall’s tau comparing the set of ranks of the actual gaze time and the set of ranks of the adjusted gaze time based on the number of identifiers.

<table>
<thead>
<tr>
<th>Kendall’s tau</th>
<th>Experts</th>
<th>Novices</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of</td>
<td></td>
</tr>
<tr>
<td></td>
<td>samples</td>
<td>Percentage</td>
</tr>
<tr>
<td>Significant agreement</td>
<td>61</td>
<td>90%</td>
</tr>
<tr>
<td>Insignificant agreement</td>
<td>7</td>
<td>10%</td>
</tr>
<tr>
<td>Insignificant disagreement</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Significant disagreement</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Total</td>
<td>68</td>
<td>100%</td>
</tr>
</tbody>
</table>

6.7.4 RQ5: Gaze Time and Line Usage

We found a strong evidence that the gaze time of a line can predict the importance of a line for the method summary. Our analysis reveals that developers tend to use source-
code lines that they read the most when they write their own summaries. Figure 23 demonstrates the steps we take to draw this conclusion.

Figure 23. The three steps performed to answer RQ6.

Step one: defining lines used.

We determine source-code lines that used by a participant to write the summary for a method. This is done by manually examining the human summaries. Consider the method presented in Figure 22. The summary written by the expert A is “Deploys units for the game, communicating this over a network connection”. The two words “Deploys” and “units” appear together in signature line 1. The main verb “deploy also appears at line 17. Therefore, the list of line used has line 1 and line 17. The second part of the summary “communicating this over a network connection” suggest the usage of line 17 although none of the words directly used from the method call in line 17. Then, if gaze time of line 1 and 17 are not zero, they are added to the lines used list.

The summary of the same method and written by the expert B is “Transmit game unit object data positioning over the network and flush the connection” which suggests the use of Line 5, 17 and 18. Line 5 is added because this participant specifies that type and the name of the data being sent “object data”. Finally, both experts use the word “game”
although it does not appear in the method. They might gain this information by reading the description of the system. Finally, if a phrase used by participant is found in multiple lines (e.g. a call that appears three times in the method), the three lines are included in the used lists as long as the duration is greater than zero. Of the 1154 non-zero lines, 181 lines are used by 5 experts. On average, experts use three lines to write their summaries for a method. The same approach is used to define 634 lines used by novices.

Furthermore, we found that novices’ summaries are longer and contain more details than experts’ summaries. The average size of experts’ summaries is 18 words while the size of novices is 30 words. Furthermore, novices seem to write very detailed summaries by describing the majority of the method’s responsibilities. On the other hand, experts write summaries based on a selected set of lines which is what majority of he proposed summarization approaches follow (Abid et al. 2015; McBurney & McMillan 2014; Rodeghero et al. 2014; Sridhara et al. 2010).

**Step two: sorting eye-tracking lines read by an expert.**

The gaze time for each line is computed as explained in 6.7. Therefore, each line appears once in each sample from one method and one expert. Then, the lines are sorted in decreasing order.

**Step three: comparing lines used to the top lines (lines with the longest gaze time).**

Finally, step three is to examine if lines used (line 1 and 17) are lines that the expert read the most. As shown in Figure 23 (step three), line 1 is the most read line and line 17 is the third most line. Therefore, expert A focus the most on lines that she uses to write her summary for the method.
The three steps are repeated for all the 69 experts’ samples for 181 lines used. Table 42 shows the number of used lines that happen to experience the highest effort. By looking at the top 30% of lines read by participants, we can actually predict 70% of the lines used by participants. The top of 50% of lines of a method (heights gaze time) can predict 87% of lines that are essential for the method summary. 100% accuracy is obtained when 86% of lines of a method is examined.

Table 42. A summary of number of lines used by experts that have the top efforts of 181 lines.

<table>
<thead>
<tr>
<th>Top 30%</th>
<th>Top 35%</th>
<th>Top 40%</th>
<th>Top 45%</th>
<th>Top 50%</th>
<th>Top 60%</th>
<th>Top 70%</th>
<th>Top 80%</th>
<th>Top 86%</th>
</tr>
</thead>
<tbody>
<tr>
<td>126</td>
<td>134</td>
<td>147</td>
<td>153</td>
<td>157</td>
<td>167</td>
<td>174</td>
<td>179</td>
<td>181</td>
</tr>
<tr>
<td>70%</td>
<td>74%</td>
<td>81%</td>
<td>84%</td>
<td>87%</td>
<td>92%</td>
<td>96%</td>
<td>99%</td>
<td>100%</td>
</tr>
</tbody>
</table>

We repeat the same approach for all the 144 novices’ samples for 634 lines used. Table 43 shows the number of used lines that happen to experience the high-reading effort. By looking at the top 30% of lines read by participants, we can actually predict 65% of the lines used by participants. The top of 50% of lines of a method (heights gaze time) can predict 84% of lines that are essential for the method summary. 100% accuracy is obtained when 89% of lines of a method is examined.

Table 43. A summary of number of lines used by novices that have the top efforts of 634 used lines.

<table>
<thead>
<tr>
<th>Top 30%</th>
<th>Top 35%</th>
<th>Top 40%</th>
<th>Top 45%</th>
<th>Top 50%</th>
<th>Top 60%</th>
<th>Top 70%</th>
<th>Top 80%</th>
<th>Top 89%</th>
</tr>
</thead>
<tbody>
<tr>
<td>414</td>
<td>444</td>
<td>477</td>
<td>502</td>
<td>535</td>
<td>581</td>
<td>615</td>
<td>630</td>
<td>634</td>
</tr>
<tr>
<td>65%</td>
<td>70%</td>
<td>75%</td>
<td>79%</td>
<td>84%</td>
<td>91%</td>
<td>97%</td>
<td>99%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Lines used by experts is slightly better predicted than lines used by novices. We conclude that the gaze time of a line can be used to identify how important the line for
summarization. Here, we define important lines as lines that developers used in their summaries.

6.7.4.1 The Similarity Between Lines Used

A line from a method can appear more than once in the set of 181 used lines if more than one expert has used it. We found that 46% of lines the “lines used” are actually used by at least two experts. To examine if participants use similar lines, we analyze methods that is summarized by at least two experts. 15 methods are summarized by three to five experts who use 85 unique lines to write their summaries. A shown in Table 44, 54% of lines are used by one expert only and 46% of lines used by at least two experts. We interpret this as experts tend to use similar lines but the details level varies.

Table 44. The Overlap between lines used by all experts to write their summaries.

<table>
<thead>
<tr>
<th>Case</th>
<th>Number</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used by one expert</td>
<td>46</td>
<td>54%</td>
</tr>
<tr>
<td>Used by two experts</td>
<td>21</td>
<td>25%</td>
</tr>
<tr>
<td>Used by three to five experts</td>
<td>18</td>
<td>21%</td>
</tr>
<tr>
<td>Total</td>
<td>85</td>
<td>100%</td>
</tr>
</tbody>
</table>

Similarly, a line from a method can appear more than once in the set of 634 used lines if more than one novice has used it. We found that 49% of lines the “lines used” are actually used by one novice as shown in Table 45 and 51% of lines are used by at least two novices.

Table 45. The Overlap between lines used by all novices to write their summaries.

<table>
<thead>
<tr>
<th>Case</th>
<th>Number</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used by one novice</td>
<td>168</td>
<td>49%</td>
</tr>
<tr>
<td>Used by two novices</td>
<td>89</td>
<td>26%</td>
</tr>
<tr>
<td>Used by three novices</td>
<td>61</td>
<td>18%</td>
</tr>
<tr>
<td>Used by four to five novices</td>
<td>24</td>
<td>7%</td>
</tr>
<tr>
<td>Total</td>
<td>342</td>
<td>100%</td>
</tr>
</tbody>
</table>
6.7.5 RQ6: Type of Lines Used

Lines used by experts to write their summaries are manually analyzed. We choose to focus on experts’ lines on this analysis because they write higher-quality summaries compare to novices. Therefore, the lines that experts use to write their summaries are carefully chosen. On the other hand, novices tend to write a short description about every three or four of lines of a method that compose their final very-detailed summaries.

108/113 of used lines by experts belongs to one of the summary units proposed by the previous literature in source code summarization (Sridhara et al. 2010) (Abid et al. 2015; McBurney & McMillan 2014; Rodeghero et al. 2014; Sridhara et al. 2011b). These tools use a set of lines from the method based on linguistic and/or static analysis. Table 46 demonstrates summarization lines categories introduced by the literature and how many of lines used fall in to these categories.

The first category is the method name. The method name is commonly used to summarize a method (Abid et al. 2015; McBurney & McMillan 2014; Rodeghero et al. 2014) which describes the goal of a method. The second category is the body. The body has four main categories of summary lines that are presented in the left most column in Table 46. A summary should have at least one of the main summary units. From the code example shown in Figure 22, last void call (in this case the last two void calls (line 17 and 18 are the main summary units). One of secondary summary units can be used to support one of the main summary units. For example, line 5 (the declaration of the local variable
data) is used to add more information to line 17. Similarly, the additional information category is used to add more information to secondary summary units.

**Table 46. Lines used by experts that fall into one of the summary units proposed by the literature automatic summarization/documentation.**

<table>
<thead>
<tr>
<th>Category</th>
<th>Type</th>
<th>Sub-type</th>
<th>Number of occurrences</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Signature</strong></td>
<td>(1) Method Name</td>
<td></td>
<td>13</td>
<td>[McBurney’14] [Rodeghero’14] [Abid’15]</td>
</tr>
<tr>
<td><strong>Main Summary Units</strong></td>
<td>(2) Last void call</td>
<td></td>
<td>13</td>
<td>[Sridhara’10]</td>
</tr>
<tr>
<td></td>
<td>(3) Same action</td>
<td></td>
<td>11</td>
<td>[Sridhara’10]</td>
</tr>
<tr>
<td></td>
<td>(4) Modifying a data member by</td>
<td>call</td>
<td>7</td>
<td>[Abid’15]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Assignment</td>
<td>2</td>
<td>[Abid’15]</td>
</tr>
<tr>
<td></td>
<td>(5) Modifying a parameter by</td>
<td>call</td>
<td>4</td>
<td>[Sridhara’11] [Abid’15]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>assignment</td>
<td>0</td>
<td>[Sridhara’11] [Abid’15]</td>
</tr>
<tr>
<td></td>
<td>(6) Return</td>
<td></td>
<td>1</td>
<td>[Sridhara’10] [Abid’15]</td>
</tr>
<tr>
<td><strong>Secondary summary units</strong></td>
<td>(7) Control flow that controls one of the main summary units</td>
<td>If/switch</td>
<td>19</td>
<td>[Sridhara’10] [Abid’15]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Loops</td>
<td>3</td>
<td>[Sridhara’10] [Wang’15]</td>
</tr>
<tr>
<td></td>
<td>(8) Modifying the Returned value by</td>
<td>call</td>
<td>6</td>
<td>[Sridhara’10] [Abid’15]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>assignment</td>
<td>12</td>
<td>[Sridhara’10] [Abid’15]</td>
</tr>
<tr>
<td></td>
<td>(9) Variable declaration and/or initialization that used in one of the main summary unit</td>
<td>Call</td>
<td>11</td>
<td>[Sridhara’10]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Assignment</td>
<td>2</td>
<td>[Sridhara’10]</td>
</tr>
<tr>
<td><strong>Additional Information</strong></td>
<td>(10) Variable declaration and/or initialization that used in one of the secondary summary units</td>
<td>Used in if-stmt</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><strong>Others</strong></td>
<td>(11) Void-call</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(12) If-statement</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td></td>
<td>113</td>
<td></td>
</tr>
</tbody>
</table>

We conclude that Table 46 validates the literature at a fine-grained level. The lines used by experts are actually the same lines that the literature has proposed.

### 6.7.6 Discussion

As stated earlier, the length of a line does not critically impact the time spent by an expert or a novice on the line. Experts and novices spent longer gaze time on lines that they use to write their summaries. By examining 30% of lines with top reading efforts, we can
find 70% of the lines used by experts and 65% of the lines used by novices to write their summaries.

Furthermore, the majority of lines used by experts fall in one of the summarization unit categories that is proposed by the literature. This finding strongly suggests that combining both structural (Abid et al. 2015) (McBurney & McMillan 2014) (Rodeghero et al. 2014) and linguistic (Sridhara et al. 2010) (Sridhara et al. 2011b) clues can improve the content of summarization to be as good as summaries written by developers. The following explain some guidance to improve summarization that drawn after our analysis and observations.

13/25 of the signatures are used by at least an expert to write their summaries. Usually, the expert starts the summary of the verb from the signature to explain the main action of a method. Then, she adds more details from a set of selected calls and other lines from the body. Therefore, we suggest using the following set of guidance to generate automatic summaries.

1) Using the method signature to represent the main action of a method.

2) Selecting one or more main summary units.

To do so, we need to tag all methods in a system with their stereotypes. The stereotype information can determine what kind of main summary unit should be used. For example, summary unit 4 (Modifying a data member) is used if the method is command while the summary unit 5 (modifying a parameter) is used on the method is void-accessor. The last void call is one of the most occurrences can and should always be added to the summarization.
3) Choosing one or more of the secondary summary units that are related to the main action as demonstrated in Table 46.

Finally, the result of this study can benefit other kind of summarization such as auto folding. As we stated earlier, during summarization or maintenance (Kevic et al. 2016), not all lines are important. For example, developers tend to ignore catch blocks as they are mostly not important for summarization. Furthermore, experts tend to focus on about 14% of a method to write a summary. These lines can be used to support auto folding of source code by folding source code lines that commonly ignored by experts.

6.7.7 Threads to Validity

Lines used are determined by the author and they are subject to human error. To mitigate this thread, the author identifies all lines that qualify to be used. Then, a line is eliminated if and only if the duration is zero.

6.8 Conclusions

This chapter presents an eye-tracking study of 18 experts and novices reading and summarizing Java methods. The study included methods from different sizes study their influences to programmers reading behavior. Three locations of a method are analyzed which are the method signature, calls, and control flow statements.

The result reveals that the signature of a method is statistically less visited by experts as the size of methods increases. Call terms does not seem to be affected by the difficulty of tasks while the harder the task, the longer control flow statements are viewed. On the
other hand, novices need a longer time and a larger number of visits when reading call or control flow terms.

Furthermore, analyzing lines that experts and novices read during summarization can speculate the important lines for automatic summarization. However, lines used by experts is better predicted than lines used by novices. Therefore, we conclude that, the gaze time of a line can properly predict important lines for summarization.
CHAPTER 7

CODE COMPREHENSION DURING SUMMARIZING TASK

This chapter discusses the activities and behaviors that a participant performs to support the process of source code comprehension. Here, we use the same data collected from the eye-tracking study described in Chapter 6. We analyze participants’ behaviors in three directions. First, we study when, how often and where developers switch to locations outside the scope of the assigned method. When a programmer starts summarizing a method and then switches to some location outside of the scope of the assigned method, this may be an indication that they need more information to understand the method or to confirm an assumption they have made. Accordingly, they spend some time looking for clues that might be found somewhere else, either within the same file or outside the file that has the assigned method to be summarized. These possible scenarios are examined for experts and novices.

Second, we adopt an approach used by (Rodeghero & McMillan 2015) to analyze eye-movement patterns, the order of words or sections that people follow when they read. One example of reading pattern is to read top-to-bottom or bottom-to-top. Rodeghereoo found that unlike natural English text (that is typically read top-to-bottom), developers tend to switch between top-to-bottom and bottom-to-top during reading source code. Here, we examine the same three type of reading patterns namely: top-to-bottom vs bottom-to-top, skimming vs thorough, and disorderly vs sectionally.
Finally, we take the analysis of participants’ reading behaviors to a new level. In particular, we propose a mechanism to predict the mental model (Von Mayrhauser & Vans 1995) that developers apply during the comprehension process. We distinguish between two models: top-down and bottom-up. Top-down is applied when a developer has a hypothesis about the code under consideration and want to confirm it by examining set of locations. On the other hand, bottom-up is used when a developer has no understanding of the code, nor any hypothesis about the code, and must closely examine (i.e., read) the code.

Considering the above three directions, we aim to answer the following research questions.

RQ1) To what degree do experts and novices examine locations outside the assigned method’s scope?

RQ2) What source code elements (if any) does a programmer look at outside the scope of the assigned method?

RQ3) To what degree [experts / novices] do prefer to [read from top to bottom vs bottom to top], [skim source code vs read thoroughly], and [read sectionally or disorderly]?

RQ4) Can eye-tracking data be used to predict participants’ conceptual model?

7.1 Gaze Behavior Outside the Method vs. Inside the Method

As explained in section 6.3, using iTrace (Shaffer et al. 2015), the participants are able to scroll up and down within the file and outside of the file at any time during the experiment. This feature available in the tool is exploited to investigate RQ1 and RQ2.
Moreover, we investigate the level of expertise of the participants influence their examination behavior with respect to inspecting locations out of the method’s scope. To answer RQ1 and RQ2, the within method and between methods examination behaviors of the participants during all method summarization tasks are analyzed.

### 7.1.1 RQ1: Time spend Outside the method scope

All participants looked outside the scope of the assigned methods of at least once during the entire experiment. On average, they spend 94.23% of their time examining locations within the assigned method and 5.77% of the time is spent examining other locations outside the scope of the method. The average time spent by experts to examine locations outside of the method is generally lower than the average time spent by novices. No average, novices spent about 6% of the time examining out-of-scope-locations; whereas, experts spent about 4% examining out-of-scope-locations.

We further inspected the responses collected from the participants after the experiment and compared their responses to the average time spent outside of the method. For all methods for which participants spent more than the average time outside of the scope of the method (i.e., methods that required more examination outside of the assigned method’s scope) at least one of the participants described the method (or one of its components such as a data member, a method call, etc.) as vague, used the word “unsure” in their summary, or answered the question “It was easy to summarize this method” with disagree.
7.1.2 RQ2: Locations examined outside the scope of the assigned method

To answer the second question, we differentiate between eight out-of-scene locations that a participant may have examined while summarizing an assigned method which are:

1) the class declaration (to which the assigned method belongs).

2) a data member defined within the same class (or file).

3) a definition of a call inside the assigned method (within the same file).

4) a usage of the assigned method (i.e., a call of the assigned method inside another method).

5) a set of lines read while searching for information.

6) a set of lines read while returning to the assigned method.

7) a set of lines read during a general scan of the source file.

8) close approximation to the assigned method, i.e., statements that belong to methods that precede or succeed the assigned method.

Table 47 and Table 48 presents the average time spent on each of the above locations by experts and novices, respectively. For both experts and novices, the largest duration spent outside the method scope is to read close approximation. Specifically, we observed that when a participant is reading the first few lines of the assigned method, he or she also look at the methods or lines that precede the method (with respect to the location in the file). Similarly, when a participant is inspecting the last few statements of an assigned method, he or she may also look at those lines that appeared after the method in the file.
This observation confirms the result of (Kevic et al. 2015). However, it does not provide any insight that the participants might have got help from looking at those nearby locations.

### Table 47. The average time spent by each expert reading different locations outside the assigned method.

<table>
<thead>
<tr>
<th>Expert ID</th>
<th>Number of tasks when outside the method is read</th>
<th>Outside Locations and Behaviors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Definition of calls</td>
<td>Data member</td>
</tr>
<tr>
<td>E1</td>
<td>(10/14) 72%</td>
<td>0</td>
</tr>
<tr>
<td>E2</td>
<td>(9/15) 60%</td>
<td>0</td>
</tr>
<tr>
<td>E3</td>
<td>(13/15) 86%</td>
<td>1%</td>
</tr>
<tr>
<td>E4</td>
<td>(6/13) 47%</td>
<td>0</td>
</tr>
<tr>
<td>E5</td>
<td>(6/12) 50%</td>
<td>0</td>
</tr>
<tr>
<td>Average</td>
<td>63%</td>
<td>1%</td>
</tr>
</tbody>
</table>

### Table 48. The average time spent by each novice reading different locations outside the assigned method.

<table>
<thead>
<tr>
<th>Novice ID</th>
<th>Number of tasks when outside the method is read</th>
<th>Outside Locations and Behaviors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Definition of calls</td>
<td>Data member</td>
</tr>
<tr>
<td>N1</td>
<td>(3/5) 60%</td>
<td>0</td>
</tr>
<tr>
<td>N2</td>
<td>(6/8) 75%</td>
<td>0</td>
</tr>
<tr>
<td>N3</td>
<td>(6/8) 75%</td>
<td>0</td>
</tr>
<tr>
<td>N4</td>
<td>(12/14) 86%</td>
<td>0</td>
</tr>
<tr>
<td>N5</td>
<td>(6/12) 50%</td>
<td>0</td>
</tr>
<tr>
<td>N6</td>
<td>(3/6) 50%</td>
<td>0</td>
</tr>
<tr>
<td>N7</td>
<td>(12/15) 80%</td>
<td>0</td>
</tr>
<tr>
<td>N8</td>
<td>(11/15) 74%</td>
<td>0</td>
</tr>
<tr>
<td>N9</td>
<td>(9/15) 60%</td>
<td>0</td>
</tr>
<tr>
<td>N10</td>
<td>(4/5) 80%</td>
<td>0</td>
</tr>
<tr>
<td>N11</td>
<td>(13/14) 93%</td>
<td>5%</td>
</tr>
<tr>
<td>N12</td>
<td>(4/14) 29%</td>
<td>0</td>
</tr>
<tr>
<td>N13</td>
<td>(9/13) 70%</td>
<td>0</td>
</tr>
<tr>
<td>Average</td>
<td>68%</td>
<td>1%</td>
</tr>
</tbody>
</table>

Data members and class declaration are the second common two locations that experts and novices examine outside the assigned method. 7/13 novices never examine data members. Locating data members and class declaration is mostly done by scrolling up or down rather than using the search option. This is might be the case as the class declaration is always found at the beginning of the file and data members are found either and the
beginning or the end of the source file. Also, scrolling up or down to look for information is used usually when the file that contains the assigned method is short. Using the search option is noticed in some cases of large files.

Additionally, one expert (E3) and one novice (E11) switch from a statement with a method call (within the assigned method) to an out-of-scope location follows the sequence of the methods captured in (or represented by) the call graph. From Table 47 and Table 48, for both experts and novices, reading outside the method scope is more related to a developer’s preference. There are novices (N2, N6, an N12) who do not read outside the scope of the assigned method expect for close approximation. The developers (E2, E4, E5, N3, N8, N9, and N10) do minimal reading outside the assigned the method. On other hand, there are experts (E1 and E3) and novices (N4, N5, N8 and N11) who read outside the method more often.

Furthermore, a random scan of other methods inside the file is a behavior that is carried more frequently by novices. On the other hand, experts tend to focus on the assigned methods and read outside the method if they want to look for some information such as the class or data member declarations.
7.2  Reading Pattern

Here, we present the three reading pattern that previously examined by (Rodeghero & McMillan 2015). We use the same approach used to compute the three metrics. Then, we compare our results to one Rodeghero’s results.

7.2.1  Top-to-bottom vs Bottom-to-top

We observed that experts and novices tend to start at the beginning of a method (not necessarily the signature) on the 75% of the cases. However, developers do not examine each line in the code. Instead, a developer may read code on the following order line 2, line 5, back to line 4, then to line 6. The overall pattern of the previous example is top-to-bottom. To examine if a developer is reading top-to-bottom or bottom-to-top, we compute the two metric for each data sample. For each sample, we keep track of every time a developer changes line by moving up or down. Then, to determine the significance, we compared the percentage that the developers read from top-to-bottom or bottom-to-top using Wilcoxon test (Wolfe & Hollander) and post the two hypotheses as the following:

\[ H_0: \text{for [experts / novices], the difference between the computed metric for reading top-to-bottom and bottom-to-top is not statistically significant.} \]

We reject a hypothesis if \(|Z| > 1.96 \) and \( p <= .05 \). From Table 49, we reject two hypotheses (H1 and H2). This indicates that both novices and experts tend to read top-to-bottom more often than bottom-to-top.
7.2.2 Skimming vs Through

As we demonstrated in section 6.7.2, participants tend to skim the code and focus on only on 7% of lines. Therefore, we expect that the skimming reading pattern is used more frequently than reading thoroughly by participants.

Rodeghero uses 1000ms to be the cut point to define the reading thoroughly pattern. Furthermore, we observe that developers spend 1000ms to 10,000ms when they focus on a word or a line. Therefore, we will use 1000ms to be our cut point to define reading thoroughly. On average, experts and novices read thoroughly 10% and 11% of the time, respectively. To measure the significance of the result, we compare the average of cases when a developer read thoroughly (spending mover 1000ms) vs skimming the code using Wilcoxon test. Then, we post the two hypotheses:

\[ H_0: \text{for [experts / novices], the difference between the computed metric for reading thoroughly and skimming the code is not statistically significant.} \]

From Table 49, we reject hypothesis H3 and H4. This strongly suggests that both novices and experts tend to skim the code and focus on a few lines only.

7.2.3 Disorderly vs Sectionally

Reading disorderly means that a developer jumps between terms/lines while reading the source code. On the other hand, reading sectionally means that the developer inspects through sections surrounding terms/lines. Similar to (Rodeghero & McMillan 2015), we define a section as a set of three continues lines. Then, we compute the percentage in which a developer changes the section to the percentage in which the developer states in the same
section. Finally, the percentages are compared using Wilcoxon test to answer the following hypotheses:

\[ H_\alpha: \text{for [experts / novices], difference between the computed metric for reading disorderly and sectionally the code is not statistically significant.} \]

From Table 49, we reject hypothesis H6 but not H5 which indicates that experts tend to use combination of disorderly and sectionally techniques when reading code while novices read the code more sectionally.

| Table 49. Statistical summary of Wilcoxon Test of experts and novices that compares the three reading patterns |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
|                 | Metric          | Reading pattern | n   | T    | Z    | p       |
| Experts         | **H**1 Gaze     | Top-to-bottom   | 69  | 2335 | -6.74 | <.0001* |
|                 |                 | Bottom-to-top   | 69  | 80   |       |         |
| Novices         | **H**2 Gaze     | Top-to-bottom   | 144 | 10270| -10.1 | <.0001* |
|                 |                 | Bottom-to-top   | 144 | 170  |       |         |
| Experts         | **H**3 Gaze     | Skim            | 69  | 2346 | -7.22 | <.0001* |
|                 |                 | Through         | 69  | 0    |       |         |
| Novices         | **H**4 Gaze     | Skim            | 144 | 1    | -10.4 | <.0001* |
|                 |                 | Through         | 144 | 10439|       |         |
| Experts         | **H**5 Gaze     | Sectionally     | 69  | 1242 | -1.60 | .10     |
|                 |                 | Disorderly      | 69  | 773  |       |         |
| Novices         | **H**6 Gaze     | Sectionally     | 144 | 7667 | -5.89 | <.0001* |
|                 |                 | Disorderly      | 144 | 2062 |       |         |

7.2.4 Compare Experts’ Patterns to Rodeghero’s Patterns.

Rodeghero and McMillan conclude that developers tend to read top-to-bottom 49% of the time while in our case developers read top-to-bottom about 59% of the time (Rodeghero & McMillan 2015). Furthermore, they conclude that developers to jump between sections 75% of the time while our result suggests that developers use both jumping and reading within a section.
One way to explain the above differences is that, in Rodeghero’s study, developers see a fixed screen where the entire code is displayed (no scrolling is allowed). With this limited size, the memory lost is minimal and developers can return to the previous location easily. In our study, methods are large in size and developers need to scroll up and down during the summarization task. When they scroll up and down some of the code become invisible. Although our study environment makes the task a bit harder than the one in (Rodeghero & McMillan 2015), our study environment simulates the real world working environment. Finally, similar to (Rodeghero & McMillan 2015), we conclude that developers tend to skim the code more frequently and read thoroughly 10% of the time.

7.3 The Comprehension Process of Developers

According to the integrated metamodel (Von Mayrhauser & Vans 1995), during source-code comprehension, developers apply one of the two comprehension models which are: Top-down and bottom-up. Top-down is used when a developer has a hypothesis and wants to confirm it by bouncing in several locations. On the other hand, bottom-up is used when a developer has less clues and need to read closely to form a hypothesis. During the comprehension process, a developer mentally chunks or groups each set of statements into higher level abstractions. These abstractions (chunks) are aggregated further until a high-level understanding of the program is obtained (Von Mayrhauser & Vans 1995) (Storey 2005).

These models are mostly theoretical and only a few user studies that uses traditional techniques (e.g., think a loud) to understand what metal model to people perform during
maintenance or debugging tasks (Von Mayrhauser & Vans 1997). Here, we utilize the eye-tracking data to predict which model is used during the summarization task.

### 7.3.1 Top-down vs. Bottom-up

To determine the type of mental model that a developer performs, we first divide each method into chunks/sections. Identifying source code sections (or code snippet) is explored by Chatterjee et al. (Chatterjee et al. 2017). They concluded that the code snippets embedded in different document types ranges between 10 to 13 LOC, which suggest that they are code examples for single and specific actions (Chatterjee et al. 2017).

Here, we identify code snippets or chunks within source-code files. A chunk is set of continuous statements that contains various levels of text-structure abstractions. Through manual analysis, we group each set of lines that form one high level action to form a chunk. The chunk ranges between 2 LOC to 10 LOC. Figure 24 presents an example of a method that is divided into four chunks. Second, we determine the metal model by comparing every two contiguous eye-tracking records.

For example, if a developer starts reading at chunk1 and the following line read also belongs to the same chunk, then we conclude that the developer is reading closely by perform bottom-up model. When the participant switches chunk, we conclude that the participant is bouncing by performing top-down.

Furthermore, there are three main locations on the screen that a participant can look at which are: the text file to write her summary, inside the assigned method and outside the method. In our analysis, we detect four reading behaviors namely; top-down inside the
method, bottom-up inside the method, top-down outside the method and bottom-up outside the method.

```java
public void run() {
    int port = 0;
    try {
        port = Integer.parseInt(hostPort.getText());
        ServerSocket serverSocket = new ServerSocket(port);
        Socket s = serverSocket.accept();
        serverSocket.close();
    } catch (IOException ex) {
        System.out.println("Accepted peer connection.");
        conn = ConnectionFactory.getInstance().createServerConnection(s, 0);
        conn.addConnectionListener(connectionListener);
        board = new Board();
        panConnect.setEnabled(false);
        panXmit.setEnabled(true);
    }
}
```

Figure 24. The four chunks of one of the method used in the study.

Table 50. The average time of reading bottom-up or top-down by experts and novices.

<table>
<thead>
<tr>
<th>Level of expertise</th>
<th>Average of Actual duration</th>
<th>Inside the method</th>
<th>Outside the method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experts</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Actual duration</td>
<td>43913ms</td>
<td>37351ms</td>
</tr>
<tr>
<td></td>
<td>Percentage of duration</td>
<td>49%</td>
<td>48%</td>
</tr>
<tr>
<td>Novices</td>
<td>Actual duration</td>
<td>66780ms</td>
<td>59763ms</td>
</tr>
<tr>
<td></td>
<td>Percentage of duration</td>
<td>48%</td>
<td>45%</td>
</tr>
</tbody>
</table>

As presented in Table 50, for experts and novices, bottom-up is more applied than top-down. This is expected as both groups are not familiar with the systems used in the study.

To examine the significance of this difference, we use Wilcoxon test to compare the parentage of time spent performing each model. We write the two hypotheses as the following:
Hₙ: for [experts / novices], the difference between the computed metric for reading top-down and bottom-up is not statistically significant.

From Table 51, We can not reject hypothesis H₁ and H₂ which indicates that the difference between top-down and bottom-up is not statically significant.

**Table 51. Statistical summary of Wilcoxon Test of experts and novices that compares Top-down and Bottom-up**

<table>
<thead>
<tr>
<th>H</th>
<th>Metric</th>
<th>Mental model</th>
<th>n</th>
<th>T</th>
<th>Z</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experts</td>
<td>H₁</td>
<td>Gaze</td>
<td>Top-down</td>
<td>30</td>
<td>937</td>
<td>-1.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bottom-up</td>
<td>36</td>
<td>1274</td>
<td></td>
</tr>
<tr>
<td>Novices</td>
<td>H₂</td>
<td>Gaze</td>
<td>Top-down</td>
<td>75</td>
<td>5164</td>
<td>-.47</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bottom-up</td>
<td>65</td>
<td>4706</td>
<td></td>
</tr>
</tbody>
</table>

7.3.2  Timeline of Participants’ Mental Models

Considering the three locations that a participant has access to and the type of mental model applied, we generate the timeline of each participant. Figure 25 shows the timeline of one expert. The x-axis reflects the changes overtime and y-axis reflects the location that the expert read and and the mental model applied. Notice that there is a straight line in front of a behavior first start at bottom-up inside the method. This indicates that he/she stained at on chunk for quit some time. The time line is also fragmented according to the level of jumping between sections and their locations. The first four steps in this diagram, the expert switches behaviors or locations. The order of locations of the first four steps are: text, top-down inside, top-down outside, and top-town inside. This switching behavior is an indicator that developer is getting ready to start and we call this the preparation phase. Once
this behavior is followed by four or higher of stable behaviors, we conclude that the participant started the new phase which is the reading phase.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{timeline.png}
\caption{The timeline of an expert summarizing the method in Figure 22}
\end{figure}

The middle section is the reading phase; the participant switches between bottom-up and top-down inside the method. The final stage is the writing phase when the participant starts switching between the text file (to write the summary) and the source code base. The text file is the file where participants write their summaries in. The first three lines of the file contain the task number, the name of the assigned method, and the path of the file containing the assigned method. This followed by a section that asks the participant to write the summary. It is possible that the participant look at the method name or the path during the performing the task. Therefore, to detect the writing phase, we check the line number of the text that a participant read. Writing phase starts if the participant starts looking at line number four or higher at the text file.

Figure 26 presents the time line of a novice. Notice that, the writing phase is not shown in this diagram although this participant did write the summary. It is possible sometimes
that the participant moves her eyes too far and the eye-tracker can not capture the eye-
movement.

![Figure 26. The timeline of one novice summarizing the method in Figure 22](image)

Most of the participants followed the sequence displayed in Figure 25 if the writing
phase is detected. After locating the method, a participant starts by bouncing between the
text file, inside and outside the method before actually reading the method. This preparation
phase is detected on 54% of novices’ samples and 53% of experts’ samples. All experts
and novices perform the reading phase. Finally, the writing phase is detected on 15% of
novices samples and 17% of experts’ samples. During writing phase participants switch
between the text file and the source code to refine their summary. Additionally, there are
two participants who start writing their summaries in early stages in several tasks. Then,
they read the source code for some time to refine the summary. Figure 27 presents the time
line of a expert who starts writing the summary at an early stage time. Finally, Figure 28
demonstrates the timeline of a novice; notice that either the preparation phase or the writing phase are not detected.

![Figure 27. The timeline of an expert during one summarization task](image1)

![Figure 28. The timeline of a novice during one summarization task](image2)

Table 53 and Table 52 presents the average time spent in each phase by experts and novices, respectively. On average, expert spend about 19% of their time preparing for the task. They usually collect the information they need about the class and data member during this phase. Novices spend about 16% of their time during the preparing stage. Although the percentage of the over time is not different between experts and novices, the
durations spend by novices is higher. On average, novices spend about 16,500ms in the preparation phase while expert spend about 12,500ms. Similarly, novices spend a longer time in reading and writing phases. This is not a surprising result and it means that novices need a longer time to understand and describe source code.

**Table 52. The Average actual duration and percentage of duration spent in each phase by experts**

<table>
<thead>
<tr>
<th>Phase Name</th>
<th>Number of non-zero cases</th>
<th>Average Number of steps</th>
<th>Average Percentage of steps</th>
<th>Average Duration in ms</th>
<th>Average Percentage of duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preparation</td>
<td>37/69</td>
<td>17</td>
<td>14%</td>
<td>12500ms</td>
<td>19%</td>
</tr>
<tr>
<td>Reading</td>
<td>69/69</td>
<td>86</td>
<td>86%</td>
<td>75500ms</td>
<td>85%</td>
</tr>
<tr>
<td>Writing</td>
<td>12/69</td>
<td>40</td>
<td>23%</td>
<td>38000ms</td>
<td>22%</td>
</tr>
</tbody>
</table>

**Table 53. The Average actual duration and percentage of duration spent in each phase by novices**

<table>
<thead>
<tr>
<th>Phase Name</th>
<th>Number of non-zero cases</th>
<th>Average Number of steps</th>
<th>Average Percentage of steps</th>
<th>Average Duration in ms</th>
<th>Average Percentage of duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preparation</td>
<td>79/144</td>
<td>18</td>
<td>18%</td>
<td>16500ms</td>
<td>16%</td>
</tr>
<tr>
<td>Reading</td>
<td>144/144</td>
<td>114</td>
<td>82%</td>
<td>122140ms</td>
<td>83%</td>
</tr>
<tr>
<td>Writing</td>
<td>29/144</td>
<td>83</td>
<td>39%</td>
<td>77700ms</td>
<td>38%</td>
</tr>
</tbody>
</table>

**7.4 Conclusion**

This chapter discusses certain activities that a developer performs during summarizing source code. Participants from both groups (experts and novices) visit locations outside the assigned method such as data members and the class declaration. However, reading outside is highly dependable on the participant preference.
On average, experts and novices read the assigned method more closely (using bottom-up mental model) than bouncing (using top-down). However, on average, novices spend a longer gaze time preforming bottom-up (66780ms) compare to experts (43913ms). Finally, we conclude that the majority of experts and novices summarize each method throughout three phases (preparation phase, reading phase, and writing phase).
CHAPTER 8

CONCLUSIONS

The dissertation addresses two research issues namely: source-code summarization and program comprehension. The first issue deals with generate automatic summarization of Object-oriented methods. Using method stereotypes, a summary template was created for each specific method stereotype. Then, static analysis is used to extract the main components of the method. Finally, each method is re-documented with the generated documentation summaries. The structured information can assist the developer to more quickly locate and understand the main purpose of the method. Furthermore, two evaluation studies to assess our automatic approach are presented.

The goal of the first study is to evaluate the effectiveness of using method stereotypes to solve the summarization problem. The first study is performed by 64 programmers who in total evaluated the automatically generated documentation of 30 methods. The results indicated that the automatically generated documentation accurately expresses what the method does.

To study the necessity of each section of the automatically generated documentation, the second evaluation is conducted using 56 methods. Based on the analyzed ratings from 75 programmers, we conclude that the most important part of the automatically generated documentation is the short description followed by the list of calls and their stereotypes. Their absences significantly reduce the documentation ratings. The results from the two evaluation studies demonstrate that method stereotypes can directly support the process of
automatically generating useful method documentation in an efficient manner. Furthermore, the approach is highly scalable and can be generalized to other OO programming languages. We believe that the automatically generated documentation can improve program comprehension during maintenance.

An eye-tracking study evolving 16 developers is performed. The goal of the study is to examine ways to improve automatic documentation and study the comprehension process of experts and novices. The study included methods from different sizes study their influences to programmers reading behavior. In the first step of the analysis, three locations of a method are analyzed which are the method signature, calls, and control flow statements. The result reveals that the signature of a method is statistically less visited by experts as the size of methods increases. Call terms does not seem to be affected by the difficulty of tasks while the harder the task, the longer control flow statements are viewed. On the other hand, novices need a longer time and a larger number of visits when reading call or control flow terms. Furthermore, we found that by looking at the 30% of lines with top reading efforts, we can find 70% of source-code lines that experts use to write their summaries.

The collected data is also used to analyze two types of mental models that developers perform during source-code comprehension namely: top-down and bottom-up. Top-down is used to confirm a predefined hypothesis by bouncing in several locations while bottom-up is to read closely to form a new one. We found that experts and novices switches between both types but spend longer time applying bottom-up. This is expected as all developers in the study do not have a domain knowledge about the code used in the study.
Finally, on average, novices spend a longer gaze time preforming bottom-up (66780ms) compare to experts (43913ms).

8.1 Contributions

The main contributions of the presented research are to automatically generate documentation summaries for C++ methods and improve the content of the documentation via an eye-tracking study.

The first main research contribution is to use method stereotype to automatically generate documentation. It uses method stereotypes to guide the content section of the documentation. According to two evaluation studies, we conclude that a template-based technique, specialized for each stereotype, combined with simple static analysis is a practical approach for automatically generated documentation.

The second contribution is an eye-tracking study of developers summarizing source-code methods. Using eye-tracking technology, we are able to collect a very-rich type data that can be examined to study several phenomena or hypotheses. We first focus of three locations which are a method signature, set of calls, and set of control flow statements. The reading effort on these locations are analyzed with respect to developers’ level of expertise, size of methods, and stereotype of methods. Furthermore, by analyzing lines that experts used to write summaries, we come up with set of recommendations to improve automatic documentation. The final contribution is an empirical analysis of activities during summarization and relate them to comprehension process.
8.2 Future work

As part of moving forward, we plan to extend our work in several directions. The summarization approach can be improved using by combining static analysis (method stereotypes) and part-of-speech tagging of identifiers (AlSuhaibani, Newman, Collard, & Maletic 2015) to refine the summarization templates. Furthermore, the approach can be enhanced by allowing a developer to choose the content of summarization (e.g., limiting the summarization to include the short description only). Additionally, although our tool is specifically for C++, the approach is easily extended to other object-oriented programming languages (ex., Java, C#).

The results from the eye-tracking experiments open the door for new research directions. First, one can fold the set of lines that ignored by developers to reduce reading effort during comprehension, then study if such approach can improve comprehension. Second, during a maintenance task, the automatic documentation summaries and/or the stereotype of a method can be displayed, then using eye-tracking data, one can examine if developers tend to read and/or use the provided documentation. Third, as there is no limit for the length of the displayed source file, the study presented here can be performed at class level by asking developers to write summaries for source code classes.

Finally, the mental models applied by a developer depends on her or level of expertise of the domain of the presented source-code. One can conduct an eye-tracking study of developers reading two sets of methods. The first set includes methods from a domain that
the developer is familiar with and the second includes methods that from a new domain for the developer. Then, the mental models from both sets are analyzed and compared.
APPENDIX A

EXAMPLES OF METHODS REDOCUMENTED BY THE SUMCODE

This appendix shows some examples of methods that used in the first and second evaluation study and their automatically generated documentation.

A.1 Accessor Methods

```cpp
// Openoffice
/*
  GetFontHeight is controller property that
  returns a computed value nHeight
  depending on parameter(s): nSize
*/

ULONG SvxCSS1Parser::GetFontHeight( USHORT nSize ) const
{
    USHORT nHeight;

    switch( nSize )
    {
        case 0: nHeight = 8*20; break;
        case 1: nHeight = 10*20; break;
        case 2: nHeight = 11*20; break;
        case 3: nHeight = 12*20; break;
        case 4: nHeight = 17*20; break;
        case 5: nHeight = 20*20; break;
        case 6:
            default: nHeight = 32*20; break;
    }

    return nHeight;
}
```
/*
Combine is predicate that
that modifies four parameters: top, bottom, left, and right and
returns a computed value combined
depending on parameter(s): block

Data member read - wxEmptyBlockDouble, m_y1, m_y2, m_x1, and m_x2

calls to -     Intersect()             voidaccessor collaborator
             IsEmpty()()             predicate

*/
bool wxBlockDouble::Combine( const wxBlockDouble &block,
                           wxBlockDouble &top, wxBlockDouble &bottom,
                           wxBlockDouble &left, wxBlockDouble &right) const
{
    top = bottom = left = right = wxEmptyBlockDouble;
wxBlockDouble iBlock;
    Intersect(*this, block, &iBlock);
    if (iBlock.IsEmpty()) return false;
    if (iBlock == *this) return true;

    bool combined = false;
    if ( block.m_y1 < m_y1 )
    { 
       top = wxBlockDouble( block.m_x1, block.m_y1, block.m_x2, m_y1 );
       combined = true;
    }
    if ( block.m_y2 > m_y2 )
    { 
       bottom = wxBlockDouble( block.m_x1, m_y2, block.m_x2, block.m_y2 );
       combined = true;
    }
    if ( block.m_x1 < m_x1 )
    { 
       left = wxBlockDouble( block.m_x1, iBlock.m_y1, m_x1, iBlock.m_y2 );
       combined = true;
    }
    if ( block.m_x2 > m_x2 )
    { 
       right = wxBlockDouble( m_x2, iBlock.m_y1, block.m_x2, iBlock.m_y2 );
       combined = true;
    }
    return combined;
}
/*
GetCharStr is controller void accessor that
modifies one parameter: rStr
collaborates with – sal_Int32, sal_Int16, sal_uInt32, and sal_Unicode
*/

void DefaultNumberingProvider::GetCharStr(
    sal_Int32 nValue, sal_Int16 nType, OUString& rStr ) const
{
    if(!nValue)
        return;

    const sal_uInt32 coDiff = 'Z' - 'A' +1;
    char cAdd = (NumberingType::CHARS_UPPER_LETTER == nType ? 'A' : 'a') - 1;
    sal_uInt32 nCalc;
    do {
        nCalc = nValue % coDiff;
        if( !nCalc )
            nCalc = coDiff;
        sal_Unicode nTmp = cAdd + nCalc;
        OUString sTmp(&nTmp, 1);
        rStr = sTmp += rStr;
        nValue -= nCalc;
        if( nValue )
            nValue /= coDiff;
    } while( nValue );
}
fillRefPixelValues is a void accessor that modifies one parameter: values via clear(), reserve(), and push_back()

Data member modified - m_status

calls to - getImageDimensions() property
    fits_make_keyn()
    doubleValueForKey() property

*/
void FitsFile::
fillRefPixelValues ( std::vector < double > & values ) const
{
    values.clear ();
    int naxis = getImageDimensions ();
    values.reserve ( naxis );

    char key [ FLEN_KEYWORD];
    char * keyroot = const_cast < char * > ( "CRVAL" );
    for ( int i = 0; i < naxis; i++ ) {
        m_status = 0;
        fits_make_keyn ( keyroot, i+1, key, & m_status );
        bool yes = hasKey ( key );

        if ( yes ) {
            double value = doubleValueForKey ( key );
            values.push_back ( value );
        } else {
            values.push_back ( 0. );
        }
    }
}
//       codeblock
/*
unindentLine is property that
  modifies one parameter: line via erase()
  returns a computed value: charsToErase
  based on data member(s) indentLength and parameter(s): unindent

Data member read - useTabs and indentLength
*/

int ASEnhancer::unindentLine(string &line, const int unindent) const
{
  size_t whitespace = line.find_first_not_of(" \	");

  if (whitespace == string::npos)
    whitespace = line.length();

  if (whitespace == 0)
    return 0;
  
  size_t charsToErase;

  if (useTabs)
  {
    charsToErase = unindent;
    if (charsToErase <= whitespace)
      line.erase(0, charsToErase);
    else
      charsToErase = 0;
  }
  else
  {
    charsToErase = unindent * indentLength;
    if (charsToErase <= whitespace)
      line.erase(0, charsToErase);
    else
      charsToErase = 0;
  }

  return charsToErase;
}
void CL_IPAddress_GetAddr::get_addrinfo(int type, sockaddr &addr, int &len, int domain) const
{
    addrinfo *tmp = info;
    while(tmp != NULL)
    {
        if(type == CL_Socket::tcp && (tmp->ai_socktype == SOCK_STREAM) && (tmp->ai_family == domain))
        {
            len = tmp->ai_addrlen;
            memcpy(&addr, tmp->ai_addr, tmp->ai_addrlen);
            return;
        }
        else if(type == CL_Socket::udp && (tmp->ai_socktype == SOCK_DGRAM) && (tmp->ai_family == domain))
        {
            len = tmp->ai_addrlen;
            memcpy(&addr,tmp->ai_addr,len);
            return;
        }
        tmp = tmp->ai_next;
    }
    throw CL_Error("CL_Socket: Could not find appropriate socket");
}
A.2 Mutator Methods

```cpp
// codeblock
/*
breakLine is a command that
modifies nine data members

Data members modified - isLineReady, isInLineBreak, spacePadNum,
formattedLineCommentNum, prependEmptyLine, readyFormattedLine,
isAppendPostBlockEmptyLineRequested,
isPrependPostBlockEmptyLineRequested, and formattedLine
*/
void ASFormatter::breakLine()
{
    isLineReady = true;
    isInLineBreak = false;
    spacePadNum = 0;
    formattedLineCommentNum = string::npos;
    prependEmptyLine = isPrependPostBlockEmptyLineRequested;

    readyFormattedLine = formattedLineRequest;
    if (isAppendPostBlockEmptyLineRequested)
    {
        isAppendPostBlockEmptyLineRequested = false;
        isPrependPostBlockEmptyLineRequested = true;
    }
    else
    {
        isPrependPostBlockEmptyLineRequested = false;
    }
    formattedLine = "";
}
```
operator<< is non-void-command that
    returns *this and
    modifies five data members: pBufPos, nBufActualPos, nBufActualLen,
    nBufFree, and bIsDirty
calls to - Write() command
*/
SvStream& SvStream::operator<< ( signed char v)
{
    //SDO
    int tmp = eIOMode;
    if(tmp == STREAM_IO_WRITE && sizeof(signed char) <= nBufFree )
    {
        *pBufPos = v;
        pBufPos++;
        nBufActualPos++;
        if( nBufActualPos > nBufActualLen )
            nBufActualLen = nBufActualPos;
        nBufFree--;  
        bIsDirty = TRUE;
    }  
    else
        Write( (char*)&v, sizeof(signed char) );
    return *this;
}
bool TiXmlNode::RemoveChild( TiXmlNode* removeThis )
{
    if ( removeThis->parent != this )
    {
        assert( 0 );
        return false;
    }
    if ( removeThis->next )
        removeThis->next->prev = removeThis->prev;
    else
        lastChild = removeThis->prev;
    if ( removeThis->prev )
        removeThis->prev->next = removeThis->next;
    else
        firstChild = removeThis->next;
    delete removeThis;
    return true;
}

int CL_OutputSource_MemoryGeneric::write( const void *data, int size )
{
    if (m_pos + size > m_size)
    {
        int new_size = m_size + size + m_blocksize;
        unsigned char *old = m_data;
        m_data = new unsigned char[new_size];
        memcpy(m_data, old, m_pos);
        delete[] old;
        m_size = new_size;
    }
    memcpy(m_data + m_pos, data, size);
    m_pos += size;
    return size;
}
A.3 Collaborational Methods

// Openoffice
/*
StartElement is a command that modifies one data member: sAuthor or sDateTime depending on a value computed from IsXMLToken()
collaborates with - XAttributeList and sal_Int16
calls to - getLength() get
GetImport()
GetNamespaceMap()
getKeyByAttrName property collaborator
getNameByIndex() property collaborator
getValueByIndex() property collaborator
IsXMLToken() no stereotype
*/

void XMLChangeInfoContext::StartElement(
    const Reference<XAttributeList> & xAttrList)
{
    sal_Int16 nLength = xAttrList->getLength();
    for(sal_Int16 nAttr = 0; nAttr < nLength; nAttr++)
    {
        OUString sLocalName;
        sal_uInt16 nPrefix = GetImport().GetNamespaceMap().
            getKeyByAttrName( xAttrList->getNameByIndex(nAttr),
                &sLocalName );
        OUString sValue = xAttrList->getValueByIndex(nAttr);
        if (XML_NAMESPACE_OFFICE == nPrefix)
        {
            if ( IsXMLToken( sLocalName, XML_CHG_AUTHOR ) )
            {
                sAuthor = sValue;
            }
            else if ( IsXMLToken( sLocalName, XML_CHG_DATE_TIME ) )
            {
                sDateTime = sValue;
            }
        }
    }
}
A.4 Factory Methods

```cpp
// Hippodraw
/*
getFunctionRep is a factory that
returns the object: frep of type: FunctionRep
collaborates with - PlotterBase, DataRep, and FunctionRep
Data member modified - m_func_reps via fillFunctionReps()
calls to - fillFunctionReps()    void accessor controller
dept -> getTarget ()      get collaborator
dept -> isComposite () predicate collaborator
*/
FunctionRep *
FunctionController::
getFunctionRep ( const PlotterBase * plotter, const DataRep * datarep ) const
{
    FunctionRep * frep = 0;
    fillFunctionReps ( m_func_reps, plotter, datarep );
    for ( unsigned int i = 0; i < m_func_reps.size(); i++ ) {
        FunctionRep * rep = m_func_reps[i];
        const DataRep * drep = rep -> getTarget ();
        if ( drep != datarep ) continue;
        frep = rep;
        if ( frep -> isComposite () ) break;
    }
    return frep;
}
```
APPENDIX B

CLUSTERING THE DURATION OF LINES THAT DEVELOPERS READ DURING SUMMARIZATION

This appendix shows clusters of lines read by experts and novices using k-mean clustering. The size of the clusters starts from five clusters and up to 10 clusters.

### B.1 Experts’ Clusters

<table>
<thead>
<tr>
<th>Group ID</th>
<th>Cluster range</th>
<th>Signature total</th>
<th>Total</th>
<th>% Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>max</td>
<td>min</td>
<td>flow</td>
<td>call</td>
</tr>
<tr>
<td>1</td>
<td>45.79</td>
<td>31.64</td>
<td>2 (2%)</td>
<td>1 (1%)</td>
</tr>
<tr>
<td>2</td>
<td>29.64</td>
<td>18.84</td>
<td>1 (1%)</td>
<td>1 (1%)</td>
</tr>
<tr>
<td>3</td>
<td>18.48</td>
<td>10.92</td>
<td>14 (17%)</td>
<td>8 (6%)</td>
</tr>
<tr>
<td>4</td>
<td>45.83</td>
<td>25.83</td>
<td>17 (20%)</td>
<td>42 (34%)</td>
</tr>
<tr>
<td>5</td>
<td>4.82</td>
<td>0.08</td>
<td>50 (60%)</td>
<td>72 (58%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td>84</td>
<td>124</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group ID</th>
<th>Cluster range</th>
<th>Signature total</th>
<th>Total</th>
<th>% Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>max</td>
<td>min</td>
<td>flow</td>
<td>call</td>
</tr>
<tr>
<td>1</td>
<td>45.79</td>
<td>40.71</td>
<td>2 (2%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>2</td>
<td>36.96</td>
<td>29.36</td>
<td>0 (0%)</td>
<td>1 (1%)</td>
</tr>
<tr>
<td>3</td>
<td>27.25</td>
<td>17.44</td>
<td>3 (4%)</td>
<td>3 (2%)</td>
</tr>
<tr>
<td>4</td>
<td>17.14</td>
<td>9.9</td>
<td>13 (16%)</td>
<td>10 (8%)</td>
</tr>
<tr>
<td>5</td>
<td>9.86</td>
<td>4.34</td>
<td>18 (21%)</td>
<td>46 (37%)</td>
</tr>
<tr>
<td>6</td>
<td>4.3</td>
<td>0.08</td>
<td>48 (57%)</td>
<td>64 (52%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td>84</td>
<td>124</td>
</tr>
<tr>
<td>Group ID</td>
<td>Cluster range</td>
<td>Type of lines</td>
<td>total</td>
<td>%</td>
</tr>
<tr>
<td>---------</td>
<td>---------------</td>
<td>---------------</td>
<td>-------</td>
<td>-----</td>
</tr>
<tr>
<td></td>
<td>max</td>
<td>min</td>
<td>Signature</td>
<td>flow</td>
</tr>
<tr>
<td>1</td>
<td>45.79</td>
<td>45.79</td>
<td>1 (1%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>2</td>
<td>41.53</td>
<td>40.71</td>
<td>1 (1%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>3</td>
<td>36.98</td>
<td>29.36</td>
<td>0 (0%)</td>
<td>1 (1%)</td>
</tr>
<tr>
<td>4</td>
<td>27.25</td>
<td>19.65</td>
<td>1 (1%)</td>
<td>1 (1%)</td>
</tr>
<tr>
<td>5</td>
<td>19.06</td>
<td>13.04</td>
<td>6 (7%)</td>
<td>6 (5%)</td>
</tr>
<tr>
<td>6</td>
<td>13.02</td>
<td>8.02</td>
<td>13 (16%)</td>
<td>13 (11%)</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>3.61</td>
<td>22 (26%)</td>
<td>46 (37%)</td>
</tr>
<tr>
<td>8</td>
<td>3.59</td>
<td>0.08</td>
<td>40 (48%)</td>
<td>57 (46%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>84</td>
<td>124</td>
<td>773</td>
<td>179</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group ID</th>
<th>Cluster range</th>
<th>Type of lines</th>
<th>total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>max</td>
<td>min</td>
<td>Signature</td>
<td>flow</td>
</tr>
<tr>
<td>1</td>
<td>45.79</td>
<td>45.79</td>
<td>1 (1%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>2</td>
<td>41.53</td>
<td>40.71</td>
<td>1 (1%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>3</td>
<td>36.98</td>
<td>31.64</td>
<td>0 (0%)</td>
<td>1 (1%)</td>
</tr>
<tr>
<td>4</td>
<td>23.22</td>
<td>17.44</td>
<td>3 (4%)</td>
<td>3 (2%)</td>
</tr>
<tr>
<td>5</td>
<td>17.14</td>
<td>11.63</td>
<td>7 (8%)</td>
<td>4 (3%)</td>
</tr>
<tr>
<td>6</td>
<td>11.57</td>
<td>7.05</td>
<td>10 (12%)</td>
<td>20 (16%)</td>
</tr>
<tr>
<td>7</td>
<td>7.02</td>
<td>3.22</td>
<td>27 (32%)</td>
<td>41 (33%)</td>
</tr>
<tr>
<td>8</td>
<td>3.18</td>
<td>0.08</td>
<td>35 (42%)</td>
<td>55 (45%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>84</td>
<td>124</td>
<td>773</td>
<td>179</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group ID</th>
<th>Cluster range</th>
<th>Type of lines</th>
<th>total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>max</td>
<td>min</td>
<td>Signature</td>
<td>flow</td>
</tr>
<tr>
<td>1</td>
<td>45.79</td>
<td>45.79</td>
<td>1 (1%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>2</td>
<td>41.53</td>
<td>40.71</td>
<td>1 (1%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>3</td>
<td>36.98</td>
<td>19.65</td>
<td>1 (1%)</td>
<td>1 (1%)</td>
</tr>
<tr>
<td>4</td>
<td>24.56</td>
<td>14.68</td>
<td>4 (5%)</td>
<td>5 (4%)</td>
</tr>
<tr>
<td>5</td>
<td>19.06</td>
<td>10.73</td>
<td>10 (12%)</td>
<td>5 (4%)</td>
</tr>
<tr>
<td>6</td>
<td>10.6</td>
<td>6.67</td>
<td>5 (6%)</td>
<td>20 (16%)</td>
</tr>
<tr>
<td>7</td>
<td>6.64</td>
<td>3.11</td>
<td>29 (35%)</td>
<td>37 (30%)</td>
</tr>
<tr>
<td>8</td>
<td>3.09</td>
<td>0.08</td>
<td>33 (40%)</td>
<td>56 (44%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>84</td>
<td>124</td>
<td>773</td>
<td>179</td>
</tr>
</tbody>
</table>
### B.2 Novices’ Clusters

<table>
<thead>
<tr>
<th>Group ID</th>
<th>Cluster range max</th>
<th>Cluster range min</th>
<th>Signature flow</th>
<th>Signature call</th>
<th>Signature others</th>
<th>total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>47.38</td>
<td>33.65</td>
<td>2 (1%)</td>
<td>2 (1%)</td>
<td>10 (1%)</td>
<td>17</td>
<td>1%</td>
</tr>
<tr>
<td>2</td>
<td>31.52</td>
<td>19.83</td>
<td>5 (3%)</td>
<td>8 (3%)</td>
<td>62 (4%)</td>
<td>85</td>
<td>4%</td>
</tr>
<tr>
<td>3</td>
<td>19.67</td>
<td>11.47</td>
<td>8 (5%)</td>
<td>23 (8%)</td>
<td>165 (11%)</td>
<td>223</td>
<td>10%</td>
</tr>
<tr>
<td>4</td>
<td>11.42</td>
<td>4.96</td>
<td>25 (16%)</td>
<td>71 (25%)</td>
<td>434 (29%)</td>
<td>606</td>
<td>26%</td>
</tr>
<tr>
<td>5</td>
<td>4.95</td>
<td>0.04</td>
<td>116 (75%)</td>
<td>178 (63%)</td>
<td>847 (56%)</td>
<td>1378</td>
<td>60%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>158</strong></td>
<td><strong>282</strong></td>
<td><strong>1518</strong></td>
<td><strong>351</strong></td>
<td><strong>2309</strong></td>
<td><strong>100%</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group ID</th>
<th>Cluster range max</th>
<th>Cluster range min</th>
<th>Signature flow</th>
<th>Signature call</th>
<th>Signature others</th>
<th>total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>47.38</td>
<td>37.63</td>
<td>2 (1%)</td>
<td>1 (0.3%)</td>
<td>7 (0.5%)</td>
<td>12</td>
<td>1%</td>
</tr>
<tr>
<td>2</td>
<td>37.20</td>
<td>27.07</td>
<td>1 (0.6%)</td>
<td>1 (0.3%)</td>
<td>18 (1%)</td>
<td>24</td>
<td>1%</td>
</tr>
<tr>
<td>3</td>
<td>26.64</td>
<td>18.22</td>
<td>5 (3%)</td>
<td>10 (4%)</td>
<td>62 (4%)</td>
<td>86</td>
<td>4%</td>
</tr>
<tr>
<td>4</td>
<td>18.13</td>
<td>10.66</td>
<td>9 (6%)</td>
<td>25 (9%)</td>
<td>185 (12%)</td>
<td>247</td>
<td>11%</td>
</tr>
<tr>
<td>5</td>
<td>10.61</td>
<td>4.61</td>
<td>26 (17%)</td>
<td>71 (25%)</td>
<td>446 (29%)</td>
<td>619</td>
<td>27%</td>
</tr>
<tr>
<td>6</td>
<td>4.62</td>
<td>0.04</td>
<td>115 (73%)</td>
<td>174 (62%)</td>
<td>800 (53%)</td>
<td>1321</td>
<td>57%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>158</strong></td>
<td><strong>282</strong></td>
<td><strong>1518</strong></td>
<td><strong>351</strong></td>
<td><strong>2309</strong></td>
<td><strong>100%</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group ID</th>
<th>Cluster range max</th>
<th>Cluster range min</th>
<th>Signature flow</th>
<th>Signature call</th>
<th>Signature others</th>
<th>total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>47.38</td>
<td>40.71</td>
<td>2 (.6%)</td>
<td>0 (0%)</td>
<td>4 (2%)</td>
<td>7</td>
<td>0%</td>
</tr>
<tr>
<td>2</td>
<td>40.68</td>
<td>33.65</td>
<td>1 (0.6%)</td>
<td>2 (1%)</td>
<td>6 (3%)</td>
<td>10</td>
<td>0%</td>
</tr>
<tr>
<td>3</td>
<td>31.52</td>
<td>22.12</td>
<td>3 (2%)</td>
<td>3 (1%)</td>
<td>44 (3%)</td>
<td>59</td>
<td>3%</td>
</tr>
<tr>
<td>4</td>
<td>22.02</td>
<td>15</td>
<td>4 (3%)</td>
<td>17 (6%)</td>
<td>97 (6%)</td>
<td>128</td>
<td>6%</td>
</tr>
<tr>
<td>5</td>
<td>14.88</td>
<td>9.04</td>
<td>13 (8%)</td>
<td>35 (12%)</td>
<td>190 (13%)</td>
<td>272</td>
<td>12%</td>
</tr>
<tr>
<td>6</td>
<td>9.02</td>
<td>4.07</td>
<td>26 (17%)</td>
<td>64 (22%)</td>
<td>452 (30%)</td>
<td>618</td>
<td>27%</td>
</tr>
<tr>
<td>7</td>
<td>4.08</td>
<td>0.04</td>
<td>111 (70%)</td>
<td>161 (57%)</td>
<td>725 (48%)</td>
<td>1215</td>
<td>53%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>158</strong></td>
<td><strong>282</strong></td>
<td><strong>1518</strong></td>
<td><strong>351</strong></td>
<td><strong>2309</strong></td>
<td><strong>100%</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group ID</th>
<th>Cluster range max</th>
<th>Cluster range min</th>
<th>Signature flow</th>
<th>Signature Call</th>
<th>Signature others</th>
<th>total</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>47.38</td>
<td>41.71</td>
<td>1 (.6%)</td>
<td>0 (0%)</td>
<td>4 (2%)</td>
<td>7</td>
<td>0%</td>
</tr>
<tr>
<td>2</td>
<td>40.6</td>
<td>33.66</td>
<td>1 (0.6%)</td>
<td>2 (1%)</td>
<td>6 (3%)</td>
<td>10</td>
<td>0%</td>
</tr>
<tr>
<td>3</td>
<td>31.52</td>
<td>25.10</td>
<td>1 (0.6%)</td>
<td>1 (0.3%)</td>
<td>25 (2%)</td>
<td>30</td>
<td>1%</td>
</tr>
<tr>
<td>4</td>
<td>24.71</td>
<td>18.94</td>
<td>5 (3%)</td>
<td>9 (3%)</td>
<td>43 (3%)</td>
<td>66</td>
<td>3%</td>
</tr>
<tr>
<td>5</td>
<td>18.64</td>
<td>13</td>
<td>5 (3%)</td>
<td>13 (5%)</td>
<td>120 (8%)</td>
<td>153</td>
<td>7%</td>
</tr>
<tr>
<td>6</td>
<td>12.8</td>
<td>7.70</td>
<td>12 (6%)</td>
<td>46 (16%)</td>
<td>230 (15%)</td>
<td>327</td>
<td>14%</td>
</tr>
<tr>
<td>7</td>
<td>7.69</td>
<td>3.53</td>
<td>29 (18%)</td>
<td>59 (21%)</td>
<td>436 (29%)</td>
<td>600</td>
<td>26%</td>
</tr>
<tr>
<td>8</td>
<td>3.52</td>
<td>0.04</td>
<td>104 (65%)</td>
<td>152 (54%)</td>
<td>654 (43%)</td>
<td>1116</td>
<td>48%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>158</strong></td>
<td><strong>282</strong></td>
<td><strong>1518</strong></td>
<td><strong>351</strong></td>
<td><strong>2309</strong></td>
<td><strong>100%</strong></td>
<td></td>
</tr>
</tbody>
</table>
| Group ID | Cluster range | Type of lines | total | %
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>max</td>
<td>min</td>
<td>Signature</td>
<td>flow</td>
</tr>
<tr>
<td>1</td>
<td>47.38</td>
<td>43.92</td>
<td>1 (.6%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>2</td>
<td>41.71</td>
<td>36.86</td>
<td>1 (.6%)</td>
<td>1 (.3%)</td>
</tr>
<tr>
<td>3</td>
<td>34.88</td>
<td>29.87</td>
<td>1 (.6%)</td>
<td>1 (.3%)</td>
</tr>
<tr>
<td>4</td>
<td>29.06</td>
<td>23.42</td>
<td>1 (.6%)</td>
<td>1 (.3%)</td>
</tr>
<tr>
<td>5</td>
<td>23.27</td>
<td>17.55</td>
<td>4 (2%)</td>
<td>12 (4%)</td>
</tr>
<tr>
<td>6</td>
<td>17.50</td>
<td>11.97</td>
<td>6 (4%)</td>
<td>15 (5%)</td>
</tr>
<tr>
<td>7</td>
<td>11.88</td>
<td>7.05</td>
<td>12 (8%)</td>
<td>51 (18%)</td>
</tr>
<tr>
<td>8</td>
<td>7.03</td>
<td>3.21</td>
<td>31 (20%)</td>
<td>54 (19%)</td>
</tr>
<tr>
<td>9</td>
<td>3.2</td>
<td>0.04</td>
<td>101 (64%)</td>
<td>147 (52%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>158</td>
<td>282</td>
<td>1518</td>
<td>351</td>
</tr>
</tbody>
</table>

| Group ID | Cluster range | Type of lines | total | %
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>max</td>
<td>min</td>
<td>Signature</td>
<td>flow</td>
</tr>
<tr>
<td>1</td>
<td>47.38</td>
<td>43.92</td>
<td>1 (.6%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>2</td>
<td>41.71</td>
<td>36.86</td>
<td>1 (.6%)</td>
<td>1 (.3%)</td>
</tr>
<tr>
<td>3</td>
<td>34.88</td>
<td>29.87</td>
<td>1 (.6%)</td>
<td>1 (.3%)</td>
</tr>
<tr>
<td>4</td>
<td>29.06</td>
<td>23.42</td>
<td>1 (.6%)</td>
<td>1 (.3%)</td>
</tr>
<tr>
<td>5</td>
<td>23.27</td>
<td>17.55</td>
<td>4 (2%)</td>
<td>12 (4%)</td>
</tr>
<tr>
<td>6</td>
<td>17.50</td>
<td>11.97</td>
<td>6 (4%)</td>
<td>15 (5%)</td>
</tr>
<tr>
<td>7</td>
<td>11.88</td>
<td>7.05</td>
<td>12 (8%)</td>
<td>51 (18%)</td>
</tr>
<tr>
<td>8</td>
<td>7.03</td>
<td>3.21</td>
<td>31 (20%)</td>
<td>54 (19%)</td>
</tr>
<tr>
<td>9</td>
<td>3.2</td>
<td>0.04</td>
<td>101 (64%)</td>
<td>147 (52%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>158</td>
<td>282</td>
<td>1518</td>
<td>351</td>
</tr>
</tbody>
</table>
REFERENCES


Representations. Paper presented at the 2008 Symposium on Eye Tracking Research & Applications (ETRA), Savannah, Georgia.


the 31st International Conference on Software Engineering (ICSE), Vancouver, British Columbia, Canada.


classes. Paper presented at the 21st International Conference on Program Comprehension (ICPC), San Francisco, CA, USA.


International Conference on Software Maintenance and Evolution (ICSME) Bermen, Germany.


