

CODE FRAGMENT SUMMARIZATION

by

Annie T.T. Ying

School of Computer Science
McGill University, Montreal, Quebec

2016-03-23

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
DOCTOR OF PHILOSOPHY

Copyright © 2016 by Annie T.T. Ying

Abstract

Code fragments are an important resource for understanding the Application Programming Interface (API) of software libraries. Many usage scenarios for code fragments require them to be distilled to their essence: for example, when serving as cues to longer documents, for reminding developers of a previously known idiom, or for displaying search results. This dissertation reports on research on shortening, or summarizing, code fragments and makes three main contributions: a set of lessons learned from a case study on a supervised machine learning approach to the generation of code fragment summaries; an empirically grounded catalog of source code summarization practices; and the design, implementation and evaluation of a novel optimization-based summarization technique for code fragments.

The case study on the generation of code fragment summaries was based on a supervised machine learning approach that classifies whether a line in a code fragment should be in a summary. We present the lessons learned that were key to the two subsequent parts of the research: the best performing feature set being a combination of syntactic and query-related features, and three limitations on our supervised machine learning approach and the line-based problem formulation. The limitations were in using line as the granularity, obtaining training data with high quality, and using only features that are local to a line without considering dependencies among different parts of the code.

Motivated by the limitations of line-based summaries, we studied how humans shorten code fragments to understand the nature of the output of the summarization process. Based on 156 hand-generated summaries obtained

from 16 participants, we analyzed decisions on which content to select and how to present this content in a summary. We elicited a catalog of common summarization practices behind these decisions across the summaries, as well as the rationale behind the practices, using a mix of qualitative and quantitative methods. We found that none of the participants exclusively extracted code verbatim for the summaries. Participants employed many practices to modify the content, by trimming a line, truncating code, aggregating a large amount of code, and refactoring code. Not only were the participants concerned with the main goal of the task to shorten code, but also with whether the summary looked compilable, readable and understandable.

With the insights from the machine learning case study and the catalog of summarization practices, we devised a technique to generate summaries constrained in both height and width: given as input a code fragment and a query (a set of keywords), our technique produces a shorter version of the fragment that fits in a two-dimensional space (L lines by W columns) and that captures as much as possible of the essential elements of the original code related to the *query*, while remaining *readable*. To generate these summaries, we developed a code summarization tool called Konaila. Konaila maximizes the value of the content selected and constrains the chosen content to be formatted within the L by W space. In a human evaluation on summaries generated from code fragments on Stack Overflow, a popular question answering forum, we found evidence that Konaila’s summaries are effective: the majority of Konaila’s summaries were judged to capture as much as possible the original elements of the code related to the Stack Overflow question while remaining readable. One important insight for future summarization technology is that optimization is an essential step in the generation of effective summaries.

The three contributions of this dissertation are a step towards generating effective code fragment summaries that can benefit usage scenarios involving the vast amount of publicly available code on the web.

Résumé

Les fragments de code (petite portion réutilisable de code source) sont d'importantes ressources afin de comprendre les interfaces de programmation (API) de bibliothèques logicielles. Dans plusieurs scénarios d'utilisation de fragments de code, ils doivent être réduits à l'essentiel. Par exemple, lorsqu'ils servent d'indices pour de plus longs programmes, afin de rappeler aux développeurs des idiomes qu'ils connaissent déjà ou encore pour afficher des résultats de recherche. Cette dissertation fait un compte-rendu de la recherche sur la réduction, ou la synthèse, des fragments de code, et apporte trois principales contributions : des leçons apprises lors de l'étude de cas sur une approche d'apprentissage automatique supervisée afin de générer des résumés de fragments de code ; un catalogue de pratiques de résumé de code source justifié empiriquement ; et une conception, une implémentation et une évaluation de techniques novatrices de résumé, basées sur l'optimisation, pour les fragments de code.

L'étude de cas sur la génération de résumé de fragments de codes est basée sur une approche d'apprentissage automatique supervisée qui détermine si chaque ligne dans un fragment de code devrait faire parti de la résumé. Nous présentons les leçons que nous avons apprises et qui sont essentielles à la réussite des deux étapes suivantes de la recherche : la collection de caractéristiques la plus performante, qui est une combinaison de caractéristiques syntactiques et particulières à des requêtes, et trois restrictions de la méthode d'apprentissage automatique supervisée et de la formulation du problème par chaque ligne. Ces restrictions provenaient de l'emploi de lignes comme le niveaux de granularité, des problèmes pour l'obtenir des données de bonne qualité and des problèmes

de travailler seulement sûr une ligne sans considérer les dépendances entre différentes parties du code.

Motivés par les limites de la résumé par ligne, nous avons étudié comment les humains réduisent des fragments de codes afin de comprendre la nature du résultat du procédé de résumé. D'après 156 résumés faites à la main par 16 participants, nous avons analysé les décisions de sélection et de présentation du contenu de la résumé. Nous avons obtenu un catalogue des pratiques de résumé communes pour ces décisions parmi ces résumés, ainsi qu'une explication de ces pratiques, en employant des méthodes qualitatives et quantitatives. Nous avons découvert qu'aucun des participants a construit une résumé en utilisant textuellement des fragments de code en façon exclusif. Les participants ont utilisé diverses techniques afin de modifier le contenu, raccourcir une ligne, tronquer le code, amasser de vaste quantités de code et réingénierie logicielle. Les participants n'étaient seulement occupés par l'objectif principal de réduction de code, mais voulaient également obtenir un code lisible, compréhensible et pouvant être compilé.

En prenant en considération ce que nous avons appris des études de cas sur l'apprentissage automatique et le catalogue de pratiques de résumé, nous avons conçu une technique pour créer des résumés restreintes par leur hauteur et leur largeur : à partir d'un fragment de code et d'une requête (un ensemble de mots clés), notre technique produit une version du fragment plus courte qui occupe un espace bidimensionnel de L lignes par W colonnes et qui capture autant d'éléments du code original que possible, relatifs à la requête, tout en demeurant lisible. Afin de générer ces résumés, nous avons développé un outil de résumé de code : Konaila. Konaila utilise un algorithme d'optimisation qui maximise la valeur du contenu sélectionné et le restreint à l'espace bidimensionnel en L par W . Dans une évaluation avec des juges humaines sur les résumés créées à partir de fragments de code sur "Stack Overflow", un site internet populaire où les utilisateurs répondent à des questions, nous avons découvert des indications démontrant que les résumés de Konaila sont efficaces : il fut jugé que la majorité des résumés de Konaila capture autant d'éléments originaux du

code relativement aux questions de "Stack Overflow" que l'espace le permet, tout en demeurant lisibles. Un important résultat pour les prochaines technologies de résumé serait que l'optimisation est une étape essentielle à la création de résumés efficaces.

Les trois principales contributions de cette dissertation forment une étape vers la création efficace de résumé des fragments de code qui peuvent bénéficier aux scénarios d'utilisation comportant de vastes quantités de code web à accès public.

Acknowledgements

It takes a village to do a PhD. I'd like to thank the following people in my village:

First of all, this research would not be possible without my supervisor, Martin Robillard. Martin has enabled the production of the best quality research. He has the nose for the "right" direction and it's impossible to find anyone who would put in the effort and interest in working through the millionth revision of a paper. I would also like to thank Martin for the incredible amount of patience with me: things don't necessarily come fast with me, and thanks for putting up with my hectic time-line, even at the very end. Finally, I appreciate the freedom he has given me to write the thesis I wanted.

Part of this freedom was also enabled by the generous funding from McGill and NSERC. McGill awarded me a 3-year full-time funding via the Tomlinson Fellowship and NSERC a 2-year full-time funding via the CGS-D scholarship. I'm also grateful to IBM, in particular my former department head Brent Hailpern, for granting me a leave of absence to pursue this PhD.

I'd like to thank my PhD defense examiners and committee members: Andrian Marcus, Laurie Hendren, Karyn Moffatt, and Joelle Pineau. Since each of the thesis contributions was closely related to a different research area (machine learning, HCI, program analysis), I'm grateful to have the expertise and comprehensive comments from this inter-disciplinary committee.

On a daily-basis during my PhD, current and former residents of McConnell 225 have provided me with tremendous support: Francisco Ferreira (the most patient listener to my complains and helped me sort out the theoretical or

conceptual aspects of my work); Christoph Treude (the best proof-reader); Bart Dagenais and Ekwa Duala-Ekoko (the best PhD role models); Shawn Otis (who did the best job translating my abstract from English to French on a short notice); in addition, Andrew Cave, Gayane Petrosyan, Yam Chhetri, Peter Rigby, Tristan Ratchford, David Kawrykow, Steven Thepshourinthone, and Gias Uddin all made the lab a humane and fun place to come in every day.

I'd like to thank two people who have shaped the researcher I am before I even started my PhD, and who has given me continuous encouragement. Gail Murphy nurtured me from the beginning (when I was still an undergraduate) and taught me critical thinking and technical writing by example. Mark Chu-Carroll, who was formerly at IBM Research and has very different research interests and personality than myself, has complemented my research experience tremendously. In addition, I'd like to thank Rob DeLine and Wesley Weimer for comments that significantly improved my research.

Xi Chen, Ying Zhang, Francisco Ferreira, Jelena Vlasenko, Nancy Songtaweasin - I will simply say thank you and give you a big hug, friends. I wouldn't have survived the PhD without your support. It was also incredible to see the larger group of friends in my thesis defense, some of whom have traveled from out of town, and others had taken off work to come.

I am extremely grateful to my parents, Ting Lee Ying and Lai Lin Ying. They may not necessarily know what research is about, but they always believe in me, and they are always there for me. They have sacrificed tremendously to immigrate to Canada, enabling my education in this amazing and nurturing country that Hong Kong could not provide me. I'm forever in-doubted to Cissy and MO for their bottomless support during some of the most difficult times in my life. Also, thank you, the Lim's for everything, for being there for me no matter what. I also would like to thank my siblings, my sister Maggie and my brother Henry – I feel like I neglected many duties as a big sister; I just want to give you a big hug.

Last but most importantly, I'd like to thank my husband Pablo. Pablo believes in me more than I believe in myself, or anyone believes in me. He

motivated me since the beginning to pursue my PhD and flourish. I'm extremely lucky to have someone like Pablo, who not only is a loving husband but also is extremely good with brainstorming, sys. admin., debugging, and proof-reading – basically went above and beyond to help me in every imaginable aspect of my PhD. Pablo, thank you for all the love, all the help, and thank you for bearing with me on this roller-coaster. Having you by my side and the PhD is a dream come true. I am dedicating this thesis to you.

Dedication

To my husband, Pablo.

Contents

Abstract	i
Résumé	iii
Acknowledgements	vii
Dedication	xi
Contents	xiii
List of Tables	xvii
List of Figures	xix
1 Introduction	1
1.1 Definition of a Code Fragment	5
1.2 Motivating the Need for Code Fragment Summaries	6
1.2.1 Serving as Cues for Longer Documents	7
1.2.2 Search Scenarios	7
1.3 Design Space and Problem Formulation	12
1.4 Line-Based Summary Generation Case Study	13
1.5 Catalog of Summarization Practices	15
1.6 Generating Summaries Constrained in Both Height and Width	15
1.7 Organization of the Dissertation	16

2	Related Work	19
2.1	Summarization of Software Artifacts	19
2.2	Studies on Code Examples	22
2.3	Code Examples in Search Engines	24
2.4	Source Code Search Engines	25
3	Code Summarization Dimensions and Problem Formulation	27
3.1	Design Space	28
3.1.1	Declarative Completeness	28
3.1.2	Query Context Dependence	30
3.1.3	Output Granularity	31
3.1.4	Summary Composition	32
3.1.5	Space Constraints	33
3.2	Problem Formulation	34
4	Generating Line-Based Extractive Summaries Using Machine Learning	37
4.1	Corpora	41
4.1.1	Main Corpus: Eclipse Original-Summary Pairs	41
4.1.2	Secondary Corpus for Cross-API Experiments: Android Original-Summary Pairs	44
4.2	Features	46
4.2.1	AST Features	47
4.2.2	Query-related Features	54
4.2.3	Call Frequency Features	55
4.3	Feature Investigation Results	57
4.3.1	Cross-Validation Settings	57
4.3.2	Metric: R-Precision	58
4.3.3	Results	58
4.4	Effectiveness Experiment	64
4.5	Generated Vs. Annotators' Summaries	67

4.6	Overall Quality	68
4.6.1	Qualitative Analysis	68
4.6.2	Pyramid Precision Results	69
4.7	Chapter Summary	71
5	Understanding Code Fragment Summarization	75
5.1	Study Set-Up	75
5.1.1	Summarization Task	76
5.1.2	Code Fragments	79
5.1.3	Context Generation	80
5.1.4	Participants	81
5.2	Conceptual Framework	81
5.2.1	Links to the Evidence	83
5.2.2	Threats to Validity	84
5.3	Selection Practices	85
5.3.1	Practices Related to Language Constructs	87
5.3.2	Practices Based on Query Terms	89
5.3.3	Practices Considering the Human Reader	90
5.4	Presentation Practices	91
5.4.1	Trimming a Line When Needed	93
5.4.2	Compressing a Large Amount of Code	94
5.4.3	Truncating Code	96
5.4.4	Formatting Code for Readability	98
5.4.5	Improving Code	99
5.5	Discussion	101
5.5.1	Selection Beyond Code and Query Terms	101
5.5.2	Most Summaries are Abstractive	102
5.5.3	Abstractive Summary Generation	102
5.5.4	Silhouette of a Summary is Important	104
5.5.5	Programming Environment Design	105
5.6	Chapter Summary	106

6	Optimization-Based Code Fragment Summarization	107
6.1	Overview of Konaila	110
6.2	Parsing and Unit of Summarization	112
6.2.1	Parsing Code Fragments	112
6.2.2	Definition of Selection Units	113
6.3	Scoring Salient Selection Units	114
6.3.1	Query Relevant Call Filter	116
6.3.2	Call-Back Filter	120
6.3.3	Control Flow Filter	123
6.3.4	Variable-Definition-Use Filter	125
6.4	Optimization and Dependency Handling	127
6.4.1	Determining the Length	129
6.4.2	Knapsack Solution and Dependency Handling	130
6.5	Evaluation	133
6.5.1	Data: Code Fragment Selection	135
6.5.2	Study Set-Up	136
6.5.3	Participants	136
6.5.4	Results	138
6.5.5	Agreement	141
6.6	Chapter Summary	141
7	Conclusion	143
7.1	Future Work	145
7.2	Closing Remark	149
A	Changes to the Java Grammar to Parse Code Fragments	165
B	Definition of Selection Units	167
B.1	Parent Reference	168
B.2	Constructs with a Body	168
B.3	Sub-division Algorithm	170
C	Predetermined Scores When There are No Salient Candidates	171
D	Changes to the Eclipse Formatter Profile	173

List of Tables

4.1	Running time for feature generation	65
5.1	Participants' Development Experience	81
5.2	Evidence of the presentation practices	92

List of Figures

1.1	A mock-up of a pop-up window containing a code fragment summary. Konaila automatically generates this summary, taking as input code fragment in Figure 1.2	8
1.2	A web page containing a code fragment of interest. Konaila can automatically generate a summary for this code fragment (see Figure 1.1)	9
1.3	Summary for the top search result for the query “android drag and drop events” on the Google search engine	9
1.4	Summary for the top search result for the “android drag and drop events” on Stack Overflow	10
1.5	Summary for a search result for the “android drag and drop events” on the Black Duck Open Hub code search engine	10
1.6	Summary for a search result for the “android drag and drop events” on the Codota search engine	11
3.1	(a) An incomplete Java program demonstrating the difficulty to extract information about method calls (b) The part of the program missing in the incomplete program demonstrated in (a). We adapted this example from work by Dagenais and Hendren [19, p.1-2].	30
4.1	A summary generated from a code fragment from the Eclipse Official FAQ [23]. The code fragment shown here is reproduced with permission under the Eclipse Public License.	38

4.2	Top Google result on the query “eclipse open editor outside eclipse”	38
4.3	Annotation tool demonstrating a summarization task on a code fragment from the Eclipse Official FAQ [23]. The code fragment is reproduced with permission under the Eclipse Public License.	42
4.4	Basic statistics of the two corpora of original-summary pairs	43
4.5	Summarization factor	44
4.6	Leave-one-feature-out experiment on the AST features	51
4.7	Percentage of variance for each successive MCA dimension added to the projected space.	53
4.8	Comparing the AST feature sets. The statistically significant results for within-API were as follows: ASTReduced+Query (R-precision=0.714) > ASTCompact+Query (0.660), ASTReduced+Query+CallFreq (0.729) > ASTCompact+Query+CallFreq (0.652), ASTReduced+Query+CallFreq (0.729) > ASTAll+Query+CallFreq (0.694). For cross-API, the only statistical significant result was ASTCompact+Query (0.498) > ASTReduced+Query	59
4.9	Six comparisons on the contribution of the Query feature set to prediction performance measured in average R-precision. From the leftmost bars to the right, the six pairs of average R-precision <i>without Query</i> and <i>with Query</i> are (0.575, 0.714), (0.372, 0.458), (0.538, 0.570), (0.073, 0.344), (0.572, 0.694), (0.384, 0.458). The pairs in bold had statistically significant differences.	62
4.10	Six comparisons on the contribution of the CallFreq feature set to the prediction performance measured in R-precision. From left to right, the six pairs of R-precision <i>without CallFreq</i> and <i>with CallFreq</i> are (0.575, 0.572), (0.372, 0.384), (0.540, 0.570), (0.341, 0.344), (0.714, 0.694), (0.458, 0.458). None of the pairs were statistically significantly different.	63
4.11	ROC curves	66

4.12	Agreement result	67
4.13	Comparing the <i>AST+Query</i> and <i>Query</i> summaries using two metrics	70
5.1	Study Set-Up and Conceptual Framework	76
5.2	Annotation tool featuring a summary by P6 (the “Shortened code example” box) and the original 25-line long code fragment. The code fragment is reproduced from Android API Guides [2] with permission under the Apache Software License, Version 2.0.	78
5.3	Summaries on the same fragment, with variations on the presentation highlighted	82
5.4	How often a construct was in a summary	86
5.5	All three summaries on the same example contained a comment listing overriding methods	95
5.6	Sample of summarized control structures	95
5.7	A summary without formatting, by P10	97
5.8	P4 refactored code from line 1 to line 2	100
6.1	Even summaries on the same fragment have a lot of differences due to the inherent subjectivity of the summarization task. . .	109
6.2	A code fragment reproduced from the Eclipse Official FAQ [23] and with permission under the Eclipse Public License	110
6.3	A summary automatically generated by Konaila from the code fragment in Figure 6.2	110
6.4	An example of a long statement taken from a code fragment used in our study on summarization practices (Chapter 5). The code fragment is from the Android API Guides [2], demonstrating the task of “displaying a fixed-duration progress indicator.” The code fragment is reproduced with permission under the Apache Software License, Version 2.0.	115

6.5	A code fragment reproduced from Android API Guides [2] (used in the summarization study in Chapter 5) with permission under the Apache Software License, Version 2.0. The code fragment has more than half of the lines on mathematical computations.	117
6.6	Is a method signature a call-back?	121
6.7	The function <i>salience_{defUse}</i>	128
6.8	An illustration to the change made to the Knapsack algorithm for a code fragment	131
6.9	Constructing the input (compound items) to encode dependencies for the knapsack routine	132
6.10	Snapshot of the web interface used in the evaluation study. The interface presents the three summaries in a random order for each task (a task corresponds to evaluating the three summaries for one fragment). For this task, Summary 1 is the Baseline ; Summary 2 is the Greedy summary; and Summary 3 is the Optimized summary (Konaila). The code fragment is reproduced from Stack Overflow with permission under the Creative Commons License, Version 2.0.	133
6.11	Distribution of the Likert ratings	137
6.12	Distribution of the Likert ratings between the 3x50 and 5x50 summaries for the Konaila summaries	137
6.13	For this task, Summary 1 was the Greedy summary; Summary 2 was the Baseline ; and Summary 3 was the Optimized summary (Konaila).	139
A.1	Three production rules that allow ellipses	166
B.1	A code fragment from the Android Official Guide demonstrating “Terminating communication with an accessory”	169
B.2	Selection units (each unit residing on a separate line) from Figure B.1	169

B.3	A statement extracted from a code fragment demonstrating “How do I open an editor programmatically”	170
B.4	Selection units extracted from Figure B.3	170

Chapter 1

Introduction

Code examples are an important type of artifact in software engineering. The Application Programming Interfaces (API) documentation for many libraries contains human crafted code examples, which are typically small fragments of code demonstrating the usage of the API. Code examples in formal documentations are one of the most effective component of good documentation [58]. Code examples have become an expected component of formal Application Programming Interfaces (API) documentation [78].

Programmers search for code examples frequently and extensively: nearly a third of the respondents in a survey of programmers searched for code examples every day, and programmers working on implementation tasks in the field conduct web search sessions almost exclusively for finding code examples [83]. Code examples are often the explicit targets of web searches [48, 71, 83, 84]. Overall, the web is an important resource for a programmer: as much as 20% of a programmer's time could be spent on the web [7].

Code examples are also an important element on popular question answering forums such as Stack Overflow.¹ Stack Overflow attracts 32 million visitors monthly, including 25 million returning visitors; these returning visitors on average use Stack Overflow 6 times every month.² Sixty-five percent of the

¹<http://stackoverflow.com>

²<http://stackoverflow.com/research/developer-survey-2015>

answers accepted by the original questioner contain code examples [89], while unanswered questions often lack code [3].

Although most software engineering researchers will agree that code examples are useful and desirable in many software engineering contexts, the question of code example *effectiveness* is much more elusive. What makes a code example effective? While the effectiveness or usability of a code example can generally be related to its intended usage, evidence is mounting that concise code examples are particularly desirable, especially for pedagogical purposes:

“It’s tough to know the context of the example and yet it has to be very small, and only highlight exactly what the concept in the API is that you’re looking for”—a Team Lead at Microsoft [78].

Highly rated answers on Stack Overflow tend to include code examples that are concise [65]. These fragments of code are typically less than four lines and are “shorter than similar code inside other answers to the same question” with “reduced complexity” [65]. In contrast, longer code examples can be difficult to understand [78] or even be misleading [21]. Longer code examples also occupy valuable screen real estate for summarizing documents, e.g., in web search results [88]. Like summaries for text documents and thumbnails for images, there are many situations where compact versions of a code example could be useful. These include: to display the result of code searches [88], to show code outline views in integrated development environments [20], and in the table of contents of API documentation.

Given the importance of code examples, the amount of publicly available source code, and the desirable properties of concise code examples, there is great potential for technology that can automatically shorten a code example, or more generally any code fragment. No such technology exists, and current knowledge and technology on natural-language summarization do not necessarily apply to source code because of fundamental differences in structure of the input data. For example, the assumption that topic and concluding sentences in

paragraphs are likely summary sentence candidates [43] does not directly apply to source code.

This dissertation reports on our research on summarizing code examples, more formally, *code fragments*, which we define as partial programs that illustrate the use of programming idioms, constructs, concepts, or API. We report on the challenges involved in devising an automatic technique for generating code fragment summaries. While most of the efforts in code summarization were targeted to code-to-text summarization, including generating textual keywords [28, 79] and textual summaries [62, 64, 73, 85] from source code in a project, we targeted our research on *code fragment-to-code fragment summarization*. Research has shown that textual summary snippets that contain exact phrases from the web page are more effective in providing evidence for a searcher to assess the relevancy of the page [97]. This finding motivated us to investigate the generation of code fragment summaries as opposed text summaries: having the output also as a code fragment enables a larger overlap with the original fragment compared to having the output as natural language text.

This dissertation contributes to the software engineering body of knowledge in three ways: lessons learned from a case study on a supervised machine learning approach to generate code fragment summaries; an empirically grounded catalog of summarization practices that serve as requirements for the problem formulation and techniques; and the design, implementation and evaluation of a novel summarization technique. We focused on summarizing code fragments written in Java.

Case study on a machine learning approach for code fragment summarization: We first investigated the generation of code fragment summaries using a supervised machine learning approach that classifies whether a line in a code fragment should be in a summary. Using features known to work well in text summarization, we investigated three types of features that exploit:

syntactic constructs in a code fragment, whether a line is related to the given query, and API calls. We investigated four research questions:

- **Feature Investigation:** Which feature combination performs well in the summary line classification problem?
- **Baseline Comparison:** Are the summaries based on a selective combination of the features better than three baselines, including a classifier that only depends on query features that did not require code parsing?
- **Annotator Comparison:** How do the summaries we generate compare to human-generated summaries?
- **Overall Summary Quality:** What is the quality of the overall summary?

Empirically grounded catalog of summarization practices: We studied how humans shorten code examples in order to understand what natural summaries look like. We asked the participants to provide free-form summaries because we found that line-based summaries often do not form sensible summaries. Using a mix of qualitative and quantitative methods, we extracted code summarization practices and their justification from human participants to inform the development in code fragment summarization. We had two research questions in this study:

- **Selection:** Which parts of the code from an original code fragment should be selected for a summary, and why?
- **Presentation:** How should the code be presented in a summary, and why?

Generation of summaries constrained by height and width: With the insights from the machine learning case study and the catalog of summarization practices, we devised a technique to generate summaries constrained in both height and width: given as input a code fragment and a query (a set of keywords), our technique produces a shorter version of the fragment that fits

in a two-dimensional space (L lines by W columns) and that captures as much as possible of the essential elements of the original code related to the *query*, while remaining *readable*. To generate these summaries, we developed an optimization-based algorithm and the corresponding tool called Konaila. For the evaluation for the summaries, we report on an experiment based on two research questions:

- **Ratings:** How much do the Konaila summaries capture the original elements of the code related to the query while remaining readable?
- **Baseline comparison:** How good are the Konaila summaries compared to a baseline using a competitive algorithm that included code units that maximally fill the given space?

For the rest of this chapter, we first provide a definition of a code fragment (Section 1.1) and the motivation for code fragment summaries (Section 1.2). In Section 1.3, we outline five dimensions important for the design of code summarization techniques, as well as the two problem formulations we used in our research. In Sections 1.4 to 1.6, we provide an overview on the research that corresponds to the three contributions of the dissertation: the lessons learned from the machine learning study on line-based summaries; the catalog of summarization practices; and the design, implementation, and evaluation of Konaila. Section 1.7 outlines the structure of the entire dissertation.

1.1 Definition of a Code Fragment

The definition of code examples we have used thus far is what Robillard [77] called small fragments of code with at least one of the following intended usage:

- “intended to demonstrate how to access the basic API functionality,” or
- as part of a series of code fragments that “form a more or less complete application” in a tutorial.

We refer to these two categories of code examples using a more general terminology, code fragments. We distinguish these two categories of code examples from code examples that form complete applications, including “both the demonstration samples sometimes distributed with an API and open source projects that developers can download from various source code repositories” [77].

In addition to documentation, human-crafted code fragments also are prevalent in community question answering sites such as Stack Overflow. A study on code fragments from Stack Overflow posts [89] found that 58% of the accepted answers on questions tagged as “android” contain at least one code fragment. Filtering out code fragments with less than three lines of code (LOC),³ the average length of the remaining fragments was 16.4 LOC and the median was 9 LOC. In terms of the content of the fragments, 17% were complete Java files, 16% were full Java method declarations, and 66% were of sets of Java statements.

Automatically extracted or retrieved code fragments also appear in research code search engines [5, 11, 103] and commercial search engines such as Black Duck Open Hub Code Search⁴ and Codota.⁵

1.2 Motivating the Need for Code Fragment Summaries

Short code fragments are not only easier to understand, but also appropriate for many usage contexts where for code fragments have to be distilled to their essence.

³The authors determined that code fragments less than three lines typically lack context that are essential for the fragment usage [89].

⁴<https://code.openhub.net/>

⁵<http://www.codota.com/>

1.2.1 Serving as Cues for Longer Documents

Figure 1.1 illustrates a mock-up showing how a code fragment summary is useful as a cue for a long web page (Figure 1.2). The mock-up demonstrates a code fragment summary in a pop-up window as a mouse is hovered over the link of the web page in Figure 1.2. Konaila (Chapter 6) generated this summary automatically, taking as input the code fragment extracted from the web page in Figure 1.2.

1.2.2 Search Scenarios

To search for code fragments, a programmer must interact with summary representations of code fragments in the search results. We illustrate some of such search scenarios:

Summary Snippet Returned by a General Search Engine: General purpose search engines typically represent a web page in the results list using a textual summary of the web page, along with the title of the web page and the link. Figure 1.3 shows a textual summary snippet of a web page returned by Google for the search query “android drag and drop events.” The summary is extracted as if the code were text. Even when a web page containing a relevant example is returned by a search engine, the summary foils a programmer’s attempt to evaluate whether a search hit is worth pursuing. In this situation, the programmer either overlooks relevant results or has to open and scan many of the pages linked from the result page [88]. Yet, in the context of general search engines, textual snippets form a significant part of evaluating whether a particular returned link is worth visiting. Cutrell and Guan found that of the time a searcher spent on a search result page, forty percent of the time was on the textual snippets [18].

Summary Snippet Returned by a Community Question Answering Forum: Stack Overflow uses similar textual summaries in the results page, along with

1.2 Motivating the Need for Code Fragment Summaries

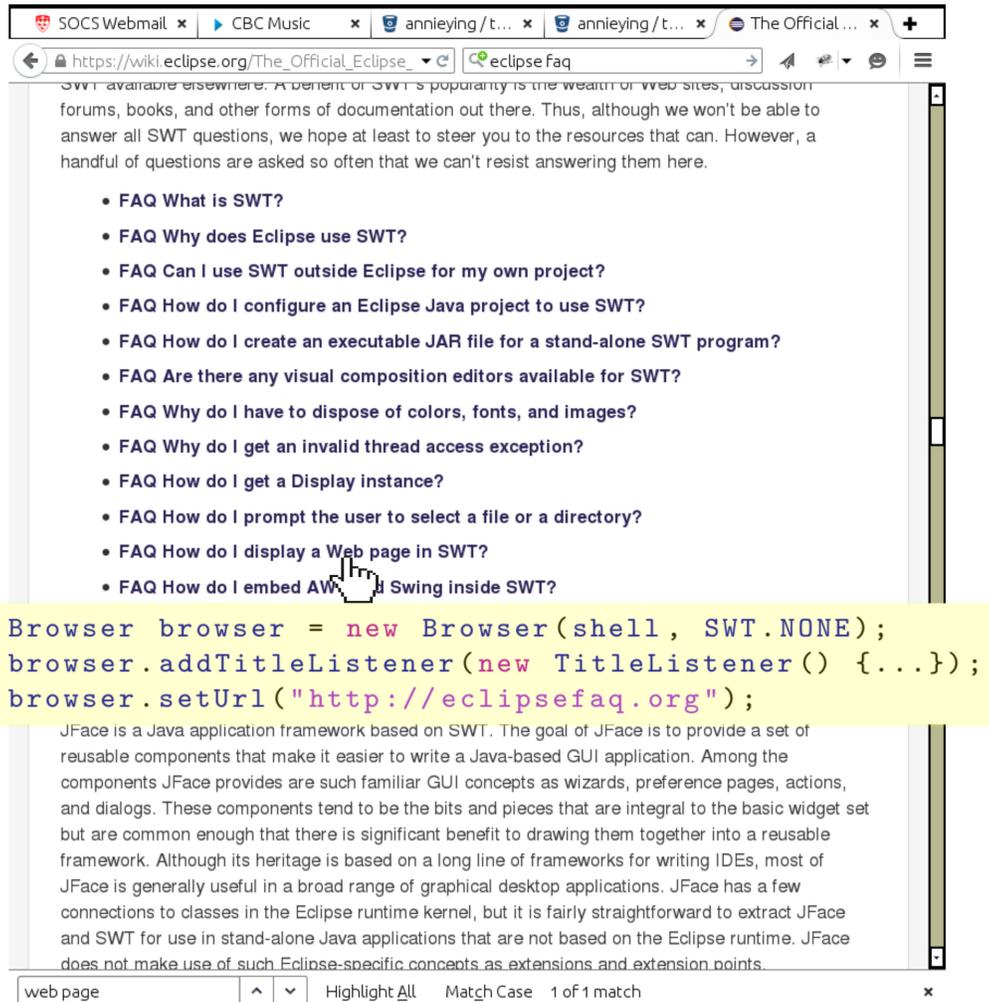


Figure 1.1: A mock-up of a pop-up window containing a code fragment summary. Konaila automatically generates this summary, taking as input code fragment in Figure 1.2

1.2 Motivating the Need for Code Fragment Summaries

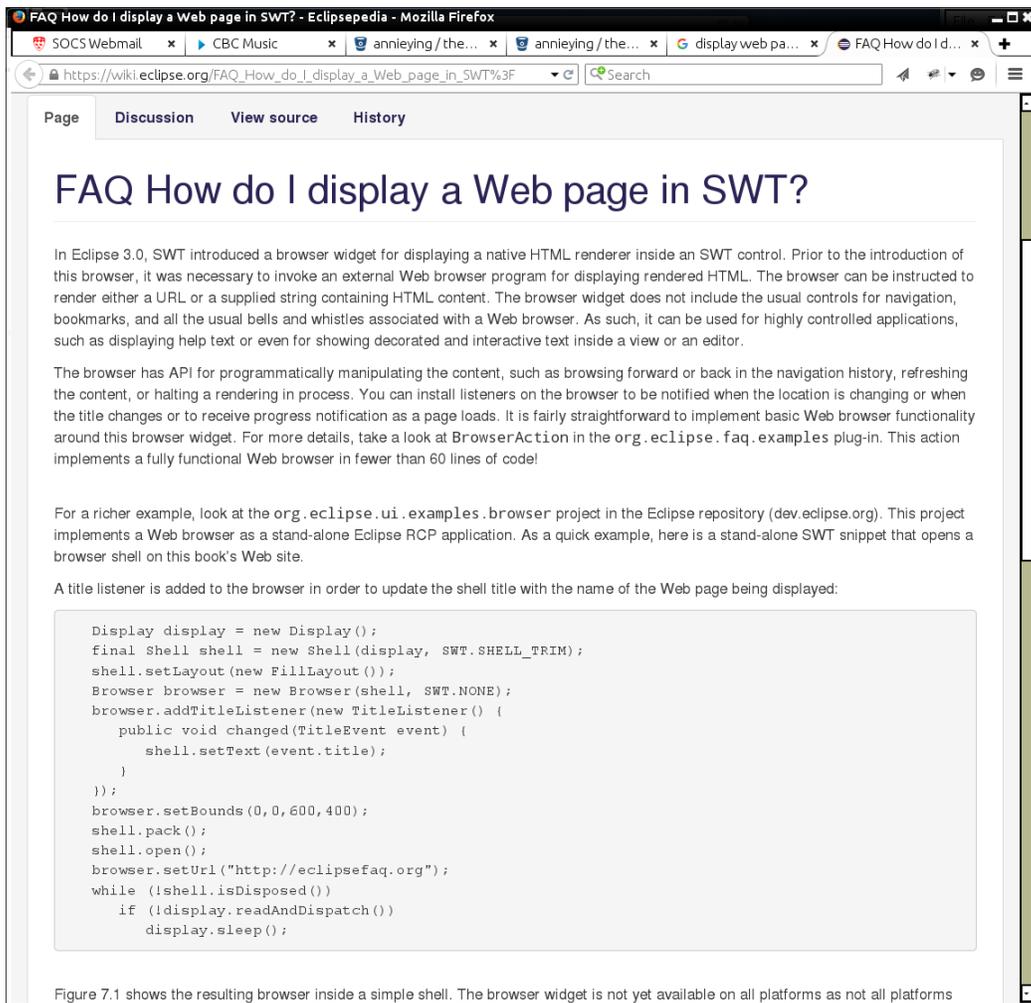


Figure 1.2: A web page containing a code fragment of interest. Konaila can automatically generate a summary for this code fragment (see Figure 1.1)

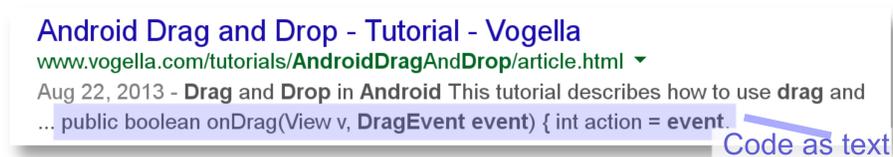


Figure 1.3: Summary for the top search result for the query “android drag and drop events” on the Google search engine

1.2 Motivating the Need for Code Fragment Summaries

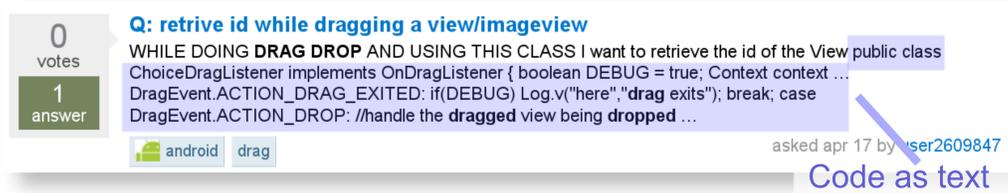


Figure 1.4: Summary for the top search result for the “android drag and drop events” on Stack Overflow



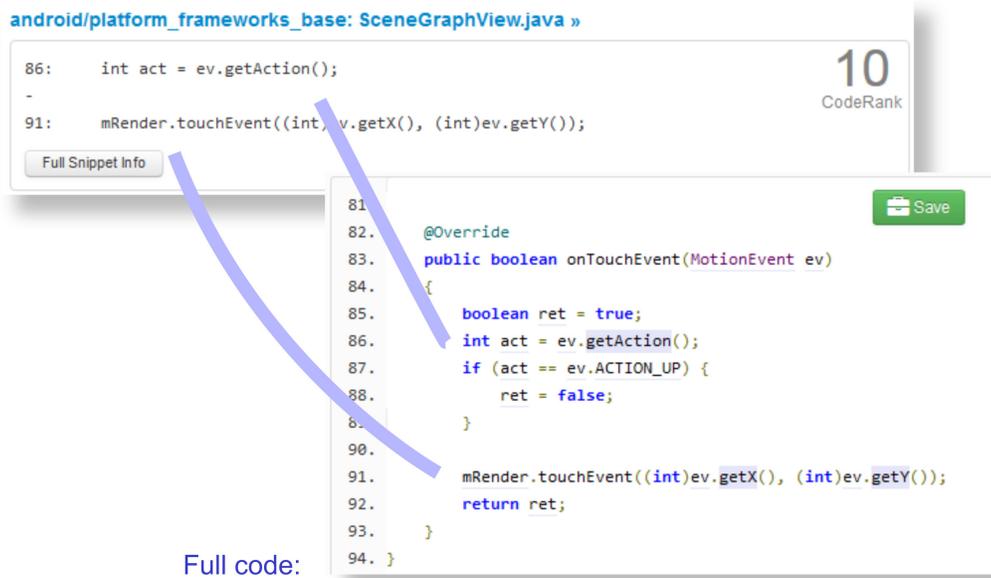
Figure 1.5: Summary for a search result for the “android drag and drop events” on the Black Duck Open Hub code search engine

the question, number of votes and tags. Figure 1.4 shows such a summary for the same query “android drag and drop events.”

Summary Returned by Code Search Engines: Researchers [5, 11, 103] and companies have built code specific search engines and recommendation systems for automatically extracting or retrieving relevant code fragment. These systems mainly focus on the accuracy and usefulness of code examples extracted or retrieved. Figure 1.5 illustrates a summary snippet on the same query returned by a commercial engine called Black Duck Open Hub. The summary is constructed from code surrounding the matched query keywords, essentially

1.2 Motivating the Need for Code Fragment Summaries

Codota's code search engine's summary:



The screenshot shows a search result for "android/platform_frameworks_base: SceneGraphView.java". The summary snippet displays lines 86 and 91: `int act = ev.getAction();` and `mRender.touchEvent((int)ev.getX(), (int)ev.getY());`. A "Full Snippet Info" button is visible. The full code is shown below, with a "Save" button. The full code includes lines 81-94, showing the `@Override` annotation, the `onTouchEvent` method signature, and the implementation logic between lines 85 and 92. A blue arrow points from the summary snippet to the corresponding lines in the full code.

```
android/platform_frameworks_base: SceneGraphView.java »  
86:   int act = ev.getAction();  
-  
91:   mRender.touchEvent((int)ev.getX(), (int)ev.getY());  
Full Snippet Info  
10  
CodeRank  
81  
82.  @Override  
83.  public boolean onTouchEvent(MotionEvent ev)  
84.  {  
85.      boolean ret = true;  
86.      int act = ev.getAction();  
87.      if (act == ev.ACTION_UP) {  
88.          ret = false;  
89.      }  
90.  
91.      mRender.touchEvent((int)ev.getX(), (int)ev.getY());  
92.      return ret;  
93.  }  
94. }  
Save  
Full code:
```

Figure 1.6: Summary for a search result for the “android drag and drop events” on the Codota search engine

displaying a small contiguous window of code surrounding the keywords. Because code is not linear, such a summary snippet is not very informative, as Sim et al. reported [83].

Figure 1.6 illustrates a summary snippet from Codota, an open source engine made available by researchers.⁶ Improving on the window approach, Codota’s summary includes source code statements (lines 86 and 91). However, the summary can still miss the logic of the code (such as in line 87) and the method call back `onTouchEvent` (line 83). The call back is especially important because a programmer has to know to override this particular method from the Android API in order for the application to react to a touch event and execute the code between lines 85 and 92.

⁶<http://www.codota.com/>

1.3 Design Space and Problem Formulation

Based on a review of work in text summarization and lessons learned from the dissertation research, we distilled this knowledge into five dimensions in the design space of code summarization in Chapter 3. The following is a brief description of the dimensions.

Declarative Completeness (Input Dimension): This dissertation is specifically about summarizing code fragments, as opposed to requiring the whole program or compilation unit as part of the input. While technology capable of summarizing a code fragment can support many application scenarios (Chapter 1.2), analyzing code fragments presents additional technical challenges.

Query Context Dependence (Input Dimension): Is the problem query-based summarization or not? In text summarization [72], researchers draw the distinction between generic summaries (without a query) and query-based or task-specific summaries. This dissertation specifically focuses on query-based summaries.

Granularity (Output Dimension): The output granularity in the extraction is an important part of the problem formulation and also in the implementation of the solution technology. Is a summary a set of tokens, lines, statements, or some other granularity? We used the line granularity (Section 3.1.3) in the case study and a granularity close to the statement level (Section 6.2.2) in Konaila.

Summary Composition (Output Dimension): Is the summary generated *extractive* (summaries generated by copying and pasting selected parts of the original code verbatim) or *abstractive* (summaries with modifications to the original code)?

Space Constraints (Output Dimension): In a textual summary, summary sentences within a paragraph are given a one-dimensional total order and printed in a two-dimensional textual display with line wrap. In a code summary, however, selected code cannot be simply printed with line wrap because in code, formatting affects readability.

In this dissertation, we investigated two query-based code fragment summarization problems in this design space. Based on a definition of a text summary [72], we generated **line-based extractive summaries** as an initial case study. In this formulation, we define a code fragment summary as a code fragment shorter than the original one, where any line in the summary is more informative (in the context of a specific query) than any other line not in the summary.

A lesson learned from the line-based extractive summarization was that line-based summaries did not always result in sensible code fragment summaries. Together with the summarization practices we learned empirically, we formulated a second summarization problem with **constraints in both height and width of the summaries**: the generation of summaries constrained in both height and width. Given as input a code fragment and a query (a set of keywords), produce a shorter version of the fragment that fits in a two-dimensional space (L lines by W columns) and that captures as much as possible of the essential elements of the original code related to the *query*, while remaining *readable*.

Chapter 3 expands on the five dimensions in the design space and the two problem formulations. Part of the discussion on these dimensions and formulations appear in two conference publications [100, 102] and a published book chapter [101].

1.4 Line-Based Summary Generation Case Study

We conducted a case study on the generation of code fragment summaries using a supervised machine learning approach that classifies whether a line in a code fragment should be in a summary. The problem formulation here was the line-based extractive summarization we presented in Section 1.3. We examined whether features known to work well in text summarization could be applied to the problem of code fragment summarization. The code fragment

summarization features included syntactic constructs in a code fragment, the amount of overlap between tokens in the source code and query words, and features describing API calls. For training and evaluating our classifier, we constructed two corpora, each consisting of a set of code fragments and the corresponding summary oracle. The summary oracle for the main corpus was constructed by four human annotators (Section 4.1.1), while the oracle for the secondary corpus was obtained through formatting conventions, i.e., the lines in a code fragment that is bold in a documentation (Section 4.1.2).

We showed that a combination of syntactic constructs in a code fragment and the overlap with a query resulted in better summaries, compared to other combinations of the feature sets, with statistical significance. This combination of features could approximate summaries in the oracle with a precision of 0.71 when we allowed summaries to be of the same length as the oracle.⁷ To interpret the level of precision reported, we computed the level of agreement among the summaries generated by the four annotators, and determined whether using our automatically generated summaries in place of one of the annotator’s summaries would degrade the level of agreement. We did not find a degradation in the agreement, indicating that given this data, the classifier is an effective summarizer for this problem. In addition, we found that API calls related features did not improve the classification performance.

We also report on three limitations which we subsequently addressed in later parts of the research: a limitation on line granularity, the difficulty in obtaining training data with high quality, and limitations on features that were local to a line without considering dependencies among different parts of the code.

Chapter 4 presents the details of the machine learning summarization approach and the evaluation. Part of this work appears in a conference publication [100].

⁷The measure is called R-precision (Section 4.3.2).

1.5 Catalog of Summarization Practices

We conducted a study to learn code summarization practices and their justification from human participants, with the goal of informing future development in source code summarization technology. Motivated by the limitations of line-based summaries, we asked participants to provide free-form summaries. The study considered 156 summaries generated by 16 programmers on 52 code fragments. Generating these summaries required determining *which* content to select and *how* to present this content. Using a mix of qualitative and quantitative methods, we analyzed practices behind these decisions across the hand-generated representations, as well as the rationale behind the practices.

We found that none of the participants exclusively extracted code verbatim for the summaries. Participants employed many practices to modify the content, by trimming a line, truncating code, aggregating a large amount of code, and refactoring code. Not only were the participants concerned with the main goal of the task to shorten code, but also with whether the summary looked compilable, readable and understandable. This study provides new ideas for how an automatic algorithm can summarize code to make the output code concise, similar to the work on what makes a good summary in the field of text summarization [36].

Chapter 5 describes the summarization study, which appears in a conference publication [102].

1.6 Generating Summaries Constrained in Both Height and Width

We devised a technique to generate summaries constrained in both height and width using an optimization-based algorithm. We implemented our technique in a tool called Konaila. Konaila uses optimization that maximizes the value of the content selected and constrains the chosen content to be formatted within a

space of L lines W characters wide. Konaila determines the value of a code unit using several factors: the similarity of the words in a code unit to the query, the extent to which the syntax of the code unit is indicative of importance, and the extent to which a code unit is involved in dependencies among a variable declaration and its usage. For example, a code unit reading or writing to a variable x is more likely to make sense in a summary if x 's variable declaration is also included in the summary.

In a human evaluation of 364 sets of summaries generated from code fragments found on Stack Overflow, we found evidence that Konaila's summaries are effective. First, the majority of Konaila's summaries (52.1%) captured as much as possible the original elements of the code related to the Stack Overflow question while remaining readable. Second, Konaila's summaries were better, with statistical significance, than a competitive baseline based on an algorithm that included code units that maximally fill the given space. One important insight for future summarization technology is that using optimization is an essential part in the effectiveness of Konaila's summaries. The conclusion is that an approach such as Konaila, based on the use of meaningful code units, a two-dimensional formulation, simple-to-compute features (based on code constructs, the overlap with the given query, and simple program analysis to determine the variation definition-use relationships within a method can produce summaries that captured the essential elements of the original code.

Chapter 6 describes the design, implementation and evaluation of Konaila.

1.7 Organization of the Dissertation

We begin the remainder of the dissertation with a discussion of related work (Chapter 2). We then present in Chapter 3 an overview of five dimensions in the design space of code summarization techniques, along with the two problem formulations we used in our research.

In Chapter 4, we describe a case study on a supervised machine learning technique to generate line-based code fragment summaries, taking advantage of three different types of features: syntactic constructs in a code fragment, the amount of overlap between tokens in the source code and query words, and features describing API calls. We present the lessons learned that were key to the two subsequent parts of the research: the best performing feature set being a combination of syntactic and query-related features, and three limitations on our supervised machine learning approach and the line-based problem formulation. The limitations were in using line as the granularity, obtaining training data with high quality, and using only features that are local to a line without considering dependencies among different parts of the code. The lessons learned from this case study constitute the first contribution of this dissertation.

We present in Chapter 5 a study on how humans shorten code fragments, in order to understand the nature of the output of the summarization process. Based on an analysis of practices we observed and their justification, we constructed a catalog of summarization practices that described decisions on which content to select and how to present this content in a summary. This catalog serves as requirements for the problem formulation and techniques to generate code fragments summaries, and constitutes the second contribution of this dissertation.

Using insights from the case study on line-based code fragment summarization and the catalog of summarization practices, we devised an optimization-based technique for generating summaries constrained in both height and width. We describe Konaila, the tool embodying this technique, and the evaluation in Chapter 6. We report on the evidence that Konaila generates effective summaries. Konaila's summaries were better, with statistical significance, than a competitive baseline that included code units that maximally fill the given space. The design, implementation, and evaluation of Konaila form the third contribution of this dissertation.

Finally, we discuss the contributions and future work in Chapter 7.

Chapter 2

Related Work

In this Chapter, we focus the related work discussion in the area of software engineering. We defer the discussion on text summarization that helped form the basis of the formulation of the code fragment summarization problem to Chapter 3.

We first discuss in Section 2.1 summarization technology in software engineering that is mostly related to the machine learning case study (Chapter 4) and the empirical study on summarization practices (Chapter 6). We then present work related to the question of what is a good code example in Section 2.2. This question is the main topic of interest in the empirical study on summarization practices (Chapter 5). Finally, we discuss summaries used in various search interfaces in Section 2.3 and source code search engines in Section 2.4.

2.1 Summarization of Software Artifacts

We look at summarization technology that takes as input source code in a project:

Code to text: Many efforts have focused on the generation of different forms of textual summaries for project code. These textual summaries are in three forms:

(1) a succinct set of keywords [28, 79]; (2) natural language phrases or sentences for specific code elements, such as exceptions [12], method parameters [87], a set of statements [86, 98], a method [85], and a class [62]; (3) textual summary interleaving with code elements for the purpose of summarizing a commit [73] or producing a release note on a large number of changes [64].

Code to code-like documentation: Buse and Weimer [13] proposed a technique to generate code-like documentation on a code change (a diff), using symbolic execution [40]. The summaries were of the form: “`When calling A(), If X, do Y, Instead of Z.`” The intention was to summarize the conditions for the control flow of a program (at run-time) to reach the changed statements.

In terms of techniques used, one main category of solutions [28, 79] constructed summaries by treating code as a block of natural language text, taking advantage of ideas from text summarization. One of the earliest techniques [28] used vector space model, a representation of text as a set of documents (vectors) containing terms. In the context of source code, Haiduc et al. modelled the code in a project as a vector space, a method as a vector, and code identifiers as terms. Their summarizer uses cosine similarity with tf-idf term weighting [80] to retrieve important terms as the summary. Our call frequency features in the machine learning case study (Section 4.2.3) were inspired by this vector space model.

Another key category of solutions involved applying known patterns that found success in summarization tasks [62, 73, 85, 86]. This knowledge-base approach was also taken by Konaila. Konaila uses patterns (we called them salience filters, Section 6.3) derived from our machine learning case study, our empirical study on summarization practices, and the heuristics used in the work by Sridhara et al [85]. Sridhara et al.’s work on generating textual summaries for methods was based on heuristics to select code units within a method. For example, code of the form “`return;`” should not be in a summary. Comparing to Sridhara et al.’s work, there are two novel ideas in Konaila: the use of formatting to deal with a constrained space, and optimization for selecting

high-valued units.

The main difference between our work and these efforts is in the problem formulation: our work is a form of **code fragment to code fragment** summarization. Both the line-based summarizer and Konaila shorten code fragments instead of producing text. In addition, our summarizers handle code fragments, as opposed to source code in a complete program in these previous summarization efforts. Code fragments are more difficult to parse and to be applicable to standard program analysis techniques. The capability to handle code fragments enables Konaila to summarize code fragments on the web, including Stack Overflow posts and on-line documentation.

The capability to parse code fragments is not unique to our work, however. Moonen’s work on parsing with Island Grammars [61] can handle code fragments with irregularities that include syntax errors and embedded code in another programming language. The idea of Island Grammars is to specify precise production rules describing the parts of the code of interest (the islands) while employing liberal production rules to match the remaining code (the water). However, because we needed to fully parse the code fragment, we did not find it sufficient to use a partial parser such as Moonen’s parser. We took a conceptually different approach: instead of ignoring parts of the code, we fully parsed the code fragment by specifying exactly what constitutes a code fragment in our grammar (Section 6.2). In the implementation of our parser, in order to allow slight irregularities in a code fragment that are not specified by our grammar, we allowed n number of lines to be non-parsable.¹ We empirically determined that $n = 3$ worked well in terms of accepting Java code fragments while not accepting spurious fragments.

Finally, several researchers have investigated **text to text** summarization, the summarization of software textual artifacts such as bug reports [50, 54, 74]. The earliest work was the application of single-document extractive

¹Another option would be to use the number of non-parsable tokens. However, parsers may not recover from a non-parsable token immediately; thus, this number is not a good measure in this context.

summarization on bug reports by Rastkar et al [74]. They applied a logistic regression classifier trained on generic conversation data (emails and meeting data) and compared against the classifier specifically trained on bug reports. They found that the classifier trained on bug reports out-performed the generic classifier. Here the major difference with code fragment summarization is in the input to the summarization procedure (i.e., text as opposed to code). This difference required our work to use features and design that are specific to code.

2.2 Studies on Code Examples

The following studies provided us with valuable knowledge on what is important in a code example. These studies are most related to the study described in Chapter 5.

Nasehi et al. investigated the characteristics of code examples in highly rated answers on Stack Overflow [65]. They found that these examples tend to be “concise”: the examples are typically less than four lines and “shorter than similar code inside other answers to the same question,” with “reduced complexity” and “unnecessary details” left out. Our focus on summarization was motivated by these highly regarded concise examples. However, in our study (Chapter 5), the design differed in that we studied summarization in a lab setting with *access to the summary author*. This design was complementary to Nasehi et al.’s study which had more generalizability but lacked the rationale behind decisions taken in generating concise code by the authors themselves. For example, was a particular part of the code intended to be an essential part of the code example, or was it just a detail? The lab setting also allowed us to have *multiple* authors summarizing the same code example so that we could examine the variability among different code summary authors.

Rodeghero et al.’s eye-tracking study [79] investigated which part of the code is important for a code summary, The study involved tracking participants’

eye movements and gaze fixation during a code-to-text summarization task. The assumption here was that eye gaze on a part of the code was indicative of the importance for inclusion in a summary. The authors found that a list of keywords generated by the term weighing scheme tf-idf [80] (as in the work by Haiduc et al [28] described in Section 2.1) were important for the selection task. The authors also investigated whether three types of program constructs (signatures, control-flow constructs, and invocations) attracted more eye gaze than other parts of the code. They found that programmers spent significantly more gaze time and fixated more often on signatures, but not on control-flow constructs and invocations. In the context of the summary-line classification task, results from our feature comparison study (Figure 4.6, groups 1, 2, and 4) corroborated these findings: leaving out these constructs from the classification model decreased the classification performance, except for the *if* expression. However, the results from our empirical study in Chapter 5 were not entirely consistent with these findings; we found that all three types of constructs were important for summarization (Section 5.3), but program constructs were not the only factor important to summarization. We found that factors such as the cognitive model of the intended summary reader (Section 5.3.3) and space constraints (Section 5.4.4) could have an impact on whether a part of the code is important for inclusion in a summary. Finally, our study in Chapter 5 also differs from Rodeghero et al.'s study in that we looked at summarization as a more open ended problem, on what a natural summary looks like, rather than only on the selection aspect.

Studying what is a good concise representation for source code also relates to work in the search engine domain on what information should be included in a textual snippet in the search result [15, 18, 69, 96, 97]. The goal for this line of work is to more effectively help a user evaluate a search result. For example, snippets including query terms [15] or exact phrases from a web page being summarized [97] resulted in better user performance in search tasks. These results motivated us to focus on the generation of code fragments as the summary output and using query as a feature in both of our summarizers.

In addition, Cutrell and Guan investigated whether varying the amount of information in the textual snippets had an effect on users' decisions on the pages to visit. They found that for queries intended for finding a piece of information (called "informational queries" in the field of web search [8]), longer snippets resulted in better user performance. For queries intended for finding a particular site the user has in mind (called "navigational queries" [8]), shorter snippets resulted in better user performance. This result is consistent with our experiments (Section 6.5.4) that longer code fragment summaries (5 lines by 50 characters) were judged to be significantly better than shorter ones (3 lines by 50 characters).

2.3 Code Examples in Search Engines

Search engines typically generate a summary snippet (e.g., Figure 1.3 for Google and Figure 1.4 for Stack Overflow) by extracting the *text* surrounding the keywords. Code specific search engines usually generate a snippet of code surrounding the keywords (e.g., Figure 1.5 for the Black Duck Open Hub code search engine and Figure 1.6 for Codota). However, these summaries are limited because important parts of the code example do not always lie in a small window surrounding the keywords. Previous efforts have aimed to improve this situation by better enabling a programmer to evaluate whether a search hit contains a relevant API element [88], or a full code example [6, 31]:

Mica [88] augments Google search results with a side-bar that contains code-specific information for search hits containing code examples. The side-bar is essentially an index of API elements (such as methods, classes, and field names) referenced in those code examples. When an API element in the side-bar is moused-over, the interface of the side-bar highlights the search hits containing that API element. This interface allows programmers to better evaluate whether a search hit contains a code example containing an API element of interest.

Assieme [31] and Blueprint [6] both present full code examples from web pages returned by a search engine. Assieme is a web interface that displays full code examples that match the query, as well as statistics on the API elements referenced in an example. In contrast, Blueprint’s interface is integrated with the development environment. The interface displays code examples returned by Google in a consistent format: each search hit consists of the title of the web page from which the code example is extracted, the link to the page, and the full example. These two systems provide a code example centric view on results from a general search engine.

Our work in code fragment summarization is complementary to these three efforts on code example centric interfaces. Our code fragment summaries can be used in place of the full examples in Assieme or Blueprint, or to augment Mica’s interface.

2.4 Source Code Search Engines

Source code search engines attempt to synthesize code examples or find relevant APIs from large code repositories that match a programmer’s query [5, 11, 14, 32, 37, 38, 59, 63, 103]. These systems take advantage of some notion of repeated patterns when determining the important elements in code. Bajracharya et al. generate code examples from a large repository of source code [5]. Their approach attempts to retrieve the most important code entities from the repository given a query, using cosine similarity and the tf-idf term weighing scheme [80]. In their approach, a code entity is a vector of terms extracted not only from identifiers and comments (as in Haiduc et al. [28] described in Section 2.1), but also from code entities with similar API usage.

Another example is the eXoaDocs system that augments API documentation with code example summaries [37, 38]. The tool generates code examples by mining from the results of Koders, a source code search engine. The component responsible for summarization resembles our tools, also using syntactic and

query features. The eXoaDocs system distills a summary from a list of candidate results from the Koders code search engine.

Our summarization work differs in the purpose. All these approaches summarize multiple occurrences of code, including results from a code search engine or a large code repository, whereas our algorithm aims at summarizing single code fragments. This is analogous to the distinction between multi-document summarization and single-document summarization in the area of text summarization [66].

Code fragment summaries are a type of representation extracted from the code. The idea that code contains more than just code for the machine is also found in the literate programming methodology which encodes documentation as part of the code [41].

Chapter 3

Code Summarization Dimensions and Problem Formulation

The area of text summarization is well established [53, 66, 72]. There are several parallels between summarizing text and code that researchers in code summarization can exploit for problem formulations and the design of summarization solutions.

One such parallel is the definition of a summary. We start with a definition of textual summary proposed by Radev et al. [72]. A textual summary is a text:

- that “is produced from one or more texts,”
- that “conveys important information in the original text(s),” and
- that “is no longer than half the original text(s) and usually significantly less than that” [72, page 399].

In this chapter, we focus the design space discussion on summarizing a single code fragment as opposed to multiple ones (the first bullet of Radev et al.’s definition above). We also focus on the problem of generating code fragment summaries as opposed to text summaries describing or explaining code as in many efforts described in Chapter 2 [12, 28, 62, 64, 73, 79, 85, 86, 87, 98].

Section 3.1 presents five dimensions in the design space for the problem of code summarization. Two of the dimensions concern the input to the summarization problem; three concern the output, the summaries. These dimensions also have implications in the design and the implementation of the two summarization technologies presented in this dissertation (Chapters 4 and 6). With these dimensions in mind, we present different formulations of the summarization problem in Section 3.2.

3.1 Design Space

The five dimensions in the design space of code summarization, the first two regarding the input and the last three regarding the output summaries, are as follows:

3.1.1 Declarative Completeness

Does summarization take as input code with declarative completeness, i.e., in a complete program or compilation unit? This dissertation is specifically about summarizing code fragments. Among the efforts in code summarization we presented in Section 2.1 [12, 13, 28, 62, 64, 73, 79, 85, 86, 87, 98], all except the AutoComment tool [98] take code fragments as input. The capability to handle code fragments enables use cases such as summarizing code fragments on Stack Overflow (Section 1.2). There are challenges in parsing an incomplete compilation unit and applying program analysis techniques on an incomplete program. We describe in more details the challenges in parsing and applying program analysis on code fragments.

Challenges in Parsing: Most parsers do not properly parse code fragments because code fragments are typically incomplete compilation units and contain non-Java tokens such as "...". In the initial case study on code fragment summarization (Chapter 4), we handled incomplete compilation units by building

an abstract syntax tree (AST) for multiple cases (Section 4.2.1): the code fragment as it is, the code fragment put in an empty Java compilation unit stub, and the code fragment put in an empty method declaration stub in a Java class, and measured which case resulted in the most complete AST. In this context, an AST is more complete than another when there are more code constructs (e.g., method invocations, method declarations, exception blocks, and conditional statements) that were recognized by the parser. We also engineered the algorithm to remove invalid tokens such as “...”.

We improved on this approach in Konaila by modifying an existing Java grammar to accept code fragments (Section 6.2.1). This design decision was conceptually and implementation-wise cleaner than transforming a code fragment into a Java compilation unit by enclosing the fragment in a stub.

Challenges in Program Analysis: Analyses that are handled by existing compilers on complete programs become more challenging for code fragments. For example, extracting which API elements (e.g., methods) are used (e.g., called) from a code fragment involves determining the type bindings of object variables that are the target of methods. In languages such as Java, this task is normally handled by compilers. However, in the context of a code fragment, resolving type bindings is technically challenging and imprecise because a code fragment is generally a subset of the whole program, possibly without the necessary dependency information for the usual type binding resolution to work. One technique that can infer type bindings from code fragments is partial program analysis [19].

For example, suppose that we are presented with the code in Figure 3.1a. If we want to know which method is called at line 6, a syntactic analysis of only the code shown in Figure 3.1a can only tell us that a method named `add` with one parameter is called at line 6, but not which class declares `add` nor the type of the parameter. This is especially problematic when multiple classes declare methods with the name `add` and one parameter. Improving upon pure syntactic analysis, partial program analysis infers that method `A.add(String)` is

<pre> 1 import p.A; 2 class B { 3 void main() { 4 A a = new A(); 5 a.p1 = ``hello``; 6 a.add(a.p1); 7 a.remove(a.p1); 8 } 9 } </pre>	<pre> 1 package p; 2 public class A { 3 String p1; 4 void add(Object o) {} 5 void remove(Object o) {} 6 } </pre>
(a)	(b)

Figure 3.1: **(a)** An incomplete Java program demonstrating the difficulty to extract information about method calls **(b)** The part of the program missing in the incomplete program demonstrated in (a). We adapted this example from work by Dagenais and Hendren [19, p.1-2].

called by looking at the string in the assignment in line 5. This inference is not strictly correct: in this example, apparently, class A (Figure 3.1b) only has one method named `add` with one parameter, `A.add(Object)`; thus, the inference is more specific than the one provided by syntactic analysis on the full program consisting of classes A and B.

The specific challenges in applying program analysis techniques to code fragments in our research are in Section 4.2.3 (obtaining the type of a method that is called) and Section 6.3.4 (determining variable definition-use relationships within a method).

3.1.2 Query Context Dependence

Is the problem query-based summarization or not? In text summarization, researchers draw the distinction between generic summaries (without a query) and query-based or task-specific summaries [66]. Examples of generic summaries include the introductory paragraph of a news report and the abstract of a paper. The rationale for query-based summarization is that code fragments usually serve a purpose (for example, to illustrate a certain API usage scenario, bug workaround, etc.). Moreover, developers often use code examples to obtain the answer to a particular query. For these reasons, we formulated summarization problems in this dissertation (Section 3.2) in relation to a query.

3.1.3 Output Granularity

Granularity concerns the unit of code that is used to assemble the summary output. Granularity is also used as an analysis unit in a summarization solution.

Choosing an appropriate granularity is important: a granularity that is too fine (such as at the token level) unnecessarily increases the computational complexity of a summarization algorithm. For machine learning based summarization technologies that depend on training data, such as the one described in Chapter 4, a granularity that is too fine requires significantly more effort from human annotators in the construction of a summary oracle. Consequently, this situation reduces the relative number of instances an annotator can annotate. On the other hand, a granularity that is too coarse limits the ability of the algorithm (or human annotator) to selectively choose regions of code as important.

The granularity issue is also of importance in text summarization systems [76]. The granularity of these systems is seldom at the word level. Early text summarization systems employed a sentence-level granularity [53]. Because sentences can be long, later systems adopted a finer granularity that is based on elementary discourse units [56], which are essentially clauses in text.

What are the points in the granularity dimension of the design space for code fragment summarization?

Line: In the summarization case study in Chapter 4, we used line granularity. The justifications at the time we conducted the case study were two-fold: First, lines are the most common unit of code measurement and management, for example, tools that identify differences in code, LOC measures, CVS, editors, and debuggers work with lines. Second, since code fragments are written by a human, the practice of separating a code statement into several lines has significance in terms of comprehensibility of the code fragment. However, as we found out from the case study, the line granularity does not result in sensible

summaries, for example, when one of the lines from a multi-line statement was selected for a summary.

Statement: In their work on code-to-text summarization [86], Sridhara et al. defined the granularity at the statement level, as they focused on code within a Java method.

Statement with Improvements: Statement granularity typically results in similarly-sized code units. However, some statements in code can be long. Our summarization study (Chapter 5) found that participants broke down long statements into multiple units when making summarization decisions. We used this insight to refine statement-level granularity, to define *selection units* which are detailed in Section 6.2.¹

3.1.4 Summary Composition

Current textual summarizers distinguish between two types of summaries: *Extractive* summaries have content obtained solely from copying and pasting whole sentences from the original document, whereas *abstractive* summaries can contain text modified from the original document [53]. More specifically, extractive summaries (extracts) are defined as follows [66, page 105]:

Extracts are produced by concatenating several sentences taken exactly as they appear in the materials being summarized.

Here the definition assumes that the output granularity is sentences. Abstractive summaries (abstracts) are defined as follows [66, page 105]:

Abstracts are written to convey the main information in the input and may reuse phrases or clauses from it, but the summaries are overall expressed in the words of the summary author.

¹One may wonder about the relationship between the selection unit representation and an intermediate representation (IR) in compiler optimization. Our selection units are designed for human consumption, whereas an IR is designed to represent source code without loss of information.

Here the granularity is at the phrase or clause level. Applying these ideas to code fragment summarization, we also make a distinction between abstractive and extractive code fragment summarization:

Abstractive Code Fragment Summarization: In the abstractive code fragment summaries that humans generated in the study described in Chapter 5, we found all participants used modifications beyond changing white spaces: namely, modification involving trimming a line, compressing a large amount of code, and truncating code. Modifications associated with abstractive summaries were present in 90% (47 out of 52) of the code fragments; thus, these 90% of the code fragments had at least one abstractive summary provided by a participant. This catalog of modifications is the basis of Section 5.4. These results demonstrate the need for abstractive summarization.

Extractive Code Fragment Summarization: In the code summarization context, an extractive code fragment summary is a subset of code units selected verbatim from the original code fragment. Selecting important parts of a code fragment is the first step in any type of summarization system on source code. The granularity can span from tokens, to code statements, and to high-level code units. Beyond the benefit of being easier to implement, extractive code fragment summaries can be beneficial. Research in the search engine domain has shown that search engine textual summary snippet that contain exact phrases from the web page are more effective summaries [97]. When summarizing a code fragment, having the output also as a code fragment enables a larger overlap with the original fragment compared to having the output as natural language text.

3.1.5 Space Constraints

From the empirical study described in Chapter 5, we found that code summarization is not only about the selection of code content (an area investigated

by many researchers in other problems such as extracting or synthesizing code examples) but also about the presentation of the content. While the need for the presentation step corroborates findings from textual summary generation [76], one aspect of presentation found in code summarization but not in textual summarization is formatting. In a textual summary, summary sentences within a paragraph are given a one-dimensional total order and printed in a two-dimensional textual display with line wrap. In a code summary, however, selected code cannot be simply printed with line wrap because in code, formatting affects readability [10, 57]. One participant from the summarization study said (Chapter 5):

I don't like packing more stuff. I always want readability. That helps me to in one glance to assess whether the particular code is helpful or not. [...] If the code is packed, it's pretty hard. It would go for another example which has more clarity.

In fact, all of the sixteen participants from the summarization study employed some formatting practices to the code. These results point to the need of a two-dimensional formulation of the code summarization problem.

3.2 Problem Formulation

This dissertation focuses on query-based code fragment summarization. In the first attempt at attacking the code fragment summarization problem (the case study in Chapter 4), we used a simple formulation based on extracting important code lines.

In this formulation, a code fragment summary is a line-based extractive summary, which we defined as follows, based on the definition by Radev et al. as we saw in the beginning of this chapter:

a code fragment shorter than the original one, where any line in the summary is more informative (in the context of a specific query) than any other line not in the summary.

3.2 Problem Formulation

The selection of a code unit of a specific granularity can be formulated as a supervised classification problem, whether to include a code unit in a summary or not. Chapter 4 presents a case study on an approach to tackle this problem.

One lesson from the case study is that line granularity is not appropriate for generating code fragment summaries because the line granularity does not adequately capture atomic parts of a code construct, such as opening of a code block and the closing of the code block. Together with the importance on readability in a space constrained setting, we formulated a second summarization problem that better captured what a summarization solution should strive for:

Given as input a code fragment and a query (a set of keywords), produce a shorter version of the fragment that fits in a two-dimensional space (L lines by W columns) and that captures as much as possible of the essential elements of the original code related to the *query*, while remaining *readable*.

Integrating width into the summarization problem is further motivated by the utility of narrower summaries, as demonstrated by a participant (P13) from the empirical study in Chapter 5,

When it comes to horizontal space, like the `onResume` one [a method declaration], when I have to pick `onResume` or `onPause`, I just pick one or the other. If there is a way to put two columns, [I would put] `onResume` on the left and `onPause` on the right.

The design, implementation, and evaluation of an optimization based approach to generate summaries constrained by height and width are in Chapter 6.

Chapter 4

Generating Line-Based Extractive Summaries Using Machine Learning

This chapter describes a case study on one way to generate code fragment summaries: a supervised machine learning approach that classifies whether a line in a code fragment should be in a summary. In this formulation, a *code fragment summary* is a line-based¹) extractive² summary, which we defined in Chapter 3 as:

a code fragment shorter than the original one, where any line in the summary is more informative (in the context of a specific query) than any other line not in the summary.

We investigated whether features known to work well in text summarization could be applied to the problem of code fragment summarization. These techniques typically apply to sentences. The code fragment summarization features include syntactic constructs in a code fragment, the amount of overlap between tokens in the source code and query words, and features describing API calls.

¹See the output granularity design dimension in Section 3.1.3.

²See the summary composition dimension in Section 3.1.4.

```

Query: How do I react to changes in source files?"
1: IResourceChangeListener rcl = new IResourceChangeListener() {
2:     public void resourceChanged(IResourceChangeEvent event) {
3:         IResource resource = event.getResource();
4:         if (resource.getFileExtension().equals("escript")) {
5:             // run the compiler
6:         }
7:     }
8: }
9: ResourcesPlugin.getWorkspace().addResourceChangeListener(rcl);

```

Figure 4.1: A summary generated from a code fragment from the Eclipse Official FAQ [23]. The code fragment shown here is reproduced with permission under the Eclipse Public License.

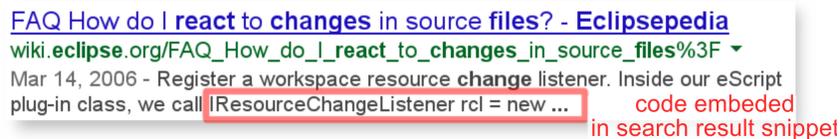


Figure 4.2: Top Google result on the query “eclipse open editor outside eclipse”

We first illustrate code fragment summarization with a Java code fragment that uses the Eclipse framework³ through its API (Application Programming Interface). Figure 4.1 illustrates how to programmatically react to changes in source files in Eclipse, the same code fragment Figure 4.2 attempted to demonstrate.⁴ Figure 4.1 also demonstrates an extractive summary generated using a machine learning algorithm. We consider this summary good because it contains the same lines that human annotators would have selected. The legend on the left hand side of Figure 4.1 indicates that at least two out of four annotators selected those lines. The summary also illustrates the effectiveness of the syntactic features: The summary contains an anonymous class declaration and the method signature, demonstrating an important call-back mechanism in the Eclipse API.

To generate summaries such as the one demonstrated in Figure 4.1, we report on four research questions concerning the construction and evaluation of the summary-line classifier:

³<https://www.eclipse.org/>

⁴The full code fragment is in Figure 4.1

RQ 1 - Feature Investigation Experiments: In the summary line classification problem, what is the effectiveness of using different combinations of features (syntactic, query related, and API calls related) in generating the summaries? We measured two aspects of classification effectiveness:

- **Within-API:** How well do the features perform when a summary-classifier is trained on and applied to the same API?
- **Cross-API:** How well do the features perform when a summary-classifier is trained on one API and tested on a different API?

With these two aspects in mind, we constructed two corpora of original-summary pairs:

- We used the first corpus on the Eclipse API which was constructed by four annotators and informed the design of the summarizer. The Eclipse API corpus also served as training and evaluation data⁵ of the classifier for both within- and cross-API.
- The second corpus on the Android API is for the cross-API question, allowing us to determine how well the classifier trained on the first corpus could generalize to another API.

We found that a selected set of AST features, combined with query-related features, provided the best performance. Based on results using a classifier trained within an API, the summaries had a average precision of 0.71 when we allowed summaries to be of the same length as the oracle.

We describe the two corpora used in the experiments for the within- and cross-API experiments in Section 4.1 and the features in Section 4.2. Section 4.3 presents the experimental settings and results on the effectiveness of different combinations of the features.

⁵The training and evaluation use different subsets of the corpus.

RQ 2 - Effectiveness Compared to Baselines: To interpret the 0.71 average precision in RQ 1, we investigated whether the summaries with this level of precision were better than some meaningful baselines. We found that the summaries generated by the classifier we determined in RQ1 were better than three baselines: a classifier that only depended on two query features that did not require code parsing, and two classifiers that either selected the first-N lines or the last-N lines as the summary. Section 4.4 describes the experimental settings and the results.

RQ 3 - Annotator Comparison: How do the summaries we generate compare to human-generated summaries? To further interpret the level of precision reported, we computed the level of agreement among the summaries generated by the four annotators, and determined whether using our automatically generated summaries in place of an annotator’s summaries would degrade the level of agreement. This analysis did not show a degradation in the agreement, implying that our classifier for the summary-line classification problem took as much advantage as it could from this data corpus. Section 4.5 describes the experimental settings and the results.

RQ 4 - Overall Summary Quality: What is the quality of the overall summary beyond the precision numbers? Section 4.6 reports on a qualitative analysis and an experiment using a metric called pyramid precision which corresponds better than precision in terms of the overall quality of the summary [67].

In the remainder of this chapter, we first describe the corpora (Section 4.1) and the features (Section 4.2) used in the construction of the classifier. We report on the experiments pertaining to the four research questions in Sections 4.3 to 4.6, respectively, and the lessons learned in Section 4.7. These lessons, which subsequently informed and addressed in later parts of research, constitute the first contribution of this dissertation. Part of this work appears in a conference publication [100].

4.1 Corpora

Section 4.1.1 describes the main corpus used in all the experiments for the four research questions. This corpus contains code fragments that demonstrate the API of the Eclipse framework, which developers can use to extend the IDE’s functionality with plug-ins. Section 4.1.2 describes the secondary corpus which is used in RQ1 for the cross-API experiments. The second corpus contains code fragments that demonstrate the API of the Android mobile platform.

4.1.1 Main Corpus: Eclipse Original-Summary Pairs

We extracted 70 code fragments from the official Eclipse FAQ website [23]. We defined the **query** as the question in the FAQ entry. For these 70 code fragments, four human annotators generated extractive summaries by selecting lines deemed important in a summary. Figure 4.3 shows a web-based tool we developed for this annotation task. We instructed the annotators to generate summaries averaged to three lines, with a maximum of six lines. We set aside 17 original-summary pairs as the **development set** for feature construction purposes. The remaining 53 code fragments were in the **evaluation set**. Figure 4.4a shows the distribution of the number of code fragments with different number of lines.

The Eclipse corpus consists of 70 code fragments produced from the 190 code fragment candidates in the Eclipse FAQ. For our corpus, we only considered those candidates that were in Java, longer than four lines, and were meant to completely answer the question. Fifty-three code fragments came from FAQ answers with only one code fragment candidate. The other 17 were extracted from the rest of the answers containing more than one code fragment candidate; this extraction involved manual inspection, either combining the code fragments for the same question (when the fragments in the answer were meant to be a single example but were separated because of interleaving text) or treating each fragment for the same question as separate (when each fragment was meant to

18. How do I insert text in the active text editor?

summary? merge with? additional?

<input type="checkbox"/>	line	<input type="checkbox"/>	1:	IWorkbenchPage page = ...;
<input type="checkbox"/>	line	<input type="checkbox"/>	2:	IEditorPart part = page.getActiveEditor();
<input type="checkbox"/>	line	<input type="checkbox"/>	3:	if (!(part instanceof AbstractTextEditor)
<input type="checkbox"/>	line	<input type="checkbox"/>	4:	return;
<input type="checkbox"/>	6	<input type="checkbox"/>	5:	ITextEditor editor = (ITextEditor)part;
<input checked="" type="checkbox"/>	line	<input type="checkbox"/>	6:	IDocumentProvider dp = editor.getDocumentProvider();
<input checked="" type="checkbox"/>	line	<input type="checkbox"/>	7:	IDocument doc = dp.getDocument(editor.getEditorInput());
<input type="checkbox"/>	line	<input type="checkbox"/>	8:	int offset = doc.getLineOffset(doc.getNumberOfLines()-4);
<input checked="" type="checkbox"/>	line	<input type="checkbox"/>	9:	doc.replace(offset, 0, pasteText+"\n");

I am familiar - I have seen Are you familiar with this code fragment?

Save this summary

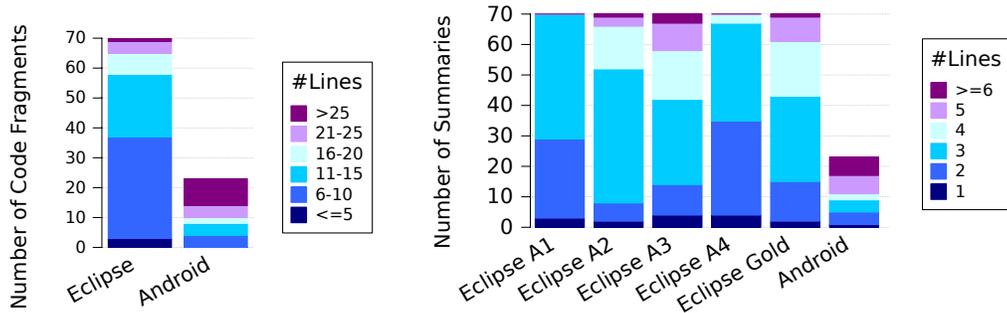
Saved summary:

```
ITextEditor editor = (ITextEditor)part;
IDocumentProvider dp = editor.getDocumentProvider();
IDocument doc = dp.getDocument(editor.getEditorInput());
doc.replace(offset, 0, pasteText+"\n");
```

Summary statistics:

	Lines in summary	Lines merged
This summary:	3	1
Average over all summaries:	2.419	1.389

Figure 4.3: Annotation tool demonstrating a summarization task on a code fragment from the Eclipse Official FAQ [23]. The code fragment is reproduced with permission under the Eclipse Public License.



(a) Length distribution for the originals (b) Length distribution for the summaries

Figure 4.4: Basic statistics of the two corpora of original-summary pairs

be an alternative solution to the question). We reserved these 17 compound code fragments as the *development set*. The choice of the 17 code fragments was due to the fact that we had to manually inspect them, disqualifying them for the use in the *evaluation set* meant to be unseen until the end of the research to avoid contaminating the models learned from the data with investigator biases.

Figure 4.4b shows the distribution of the length of the summaries generated by the four annotators, Eclipse A1 to A4. The Fleiss’ Kappa agreement statistic of the four annotators is 0.487 [44]. This statistic assesses how far away the observed agreement among multiple annotators is from the agreement if the annotators were to mark the summary lines randomly. A kappa of 0 indicates random markings and a Kappa of 1 indicates perfect agreement. A Kappa value between 0.4-0.6 is considered a moderate agreement [44]. However, since the goal of this annotation task is to elicit what a summary entails, and our annotation instructions allowed an annotator to judge what should be in the summary while not specifying its exact properties, the Kappa value of the four annotators is appropriate considering our task. This level of agreement reassures us that a summary code fragment is not a random set of lines.

To train and evaluate the summarizer, we constructed **gold standard**

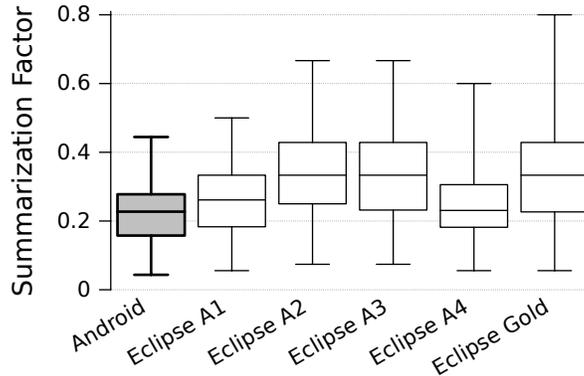


Figure 4.5: Summarization factor

summaries consisting of lines where two or more annotators agree on the in-summary annotations because of the annotator task’s subjectivity. The “Eclipse Gold” column in Figure 4.4b shows the distribution of the length of the gold standard summaries. On average, the size of a summary in the gold standard was 33.5% of the original code fragment (the right-most box-plot in Figure 4.5).

4.1.2 Secondary Corpus for Cross-API Experiments: Android Original-Summary Pairs

To explore whether the classifier could generalize to a different data set, we trained the classifier on the Eclipse summaries and applied it to a second corpus consisting of Android code fragments. This corpus consists of code fragments extracted from a technical book that demonstrates the development for the Android operating system for mobile devices [9]. We defined the **query** of a code fragment as the listing caption. For the **summaries**, we leveraged the author’s use of bold formatting style, intended to highlight certain code fragment lines. We did not need to consolidate multiple versions of summaries in the oracle when evaluating the quality of the predicted summaries for the Android fragments. In total, 22 original-summary pairs were selected for the corpus as follows.

For the Android corpus, we employed the following procedure to systematically choose the code fragments whose bold lines were summaries. The book contains 146 listings of code fragments with bold formatting, 41 of which were in Java. We eliminated four code fragments that were duplicates, leaving us with 37 fragment-summary pairs. To ensure the bold lines indeed form summaries as in the Eclipse data set, we manually inspected the bold lines from the 37 fragments. We found two conceptual differences between the Eclipse gold standard summaries and the summaries derived from bold lines in the Android corpus:

1. Thirteen out of the 37 fragments contain a bold line with only a curly brace. None of the Eclipse gold standard summaries contain such single-brace lines; in fact, none of the annotators marked any of these lines. The sole purpose for including a single-brace line in a summary is to match the brace. However, in line-based extractive summarization, it is obvious that a single-brace line never conveys more useful information than other lines in the code fragments. To make the Android corpus more similar to the Eclipse one, we excluded the single-brace bold lines as part of the oracle summaries in the Android corpus.
2. A significant number of Android fragments contain a “large chunk” of bold lines, either a whole method declaration (7 out of 37) and/or more than five consecutive lines (13 out of 37). Out of the 37 code fragments, 13 (35%) of the code fragments have fewer than 3 consecutive bold lines, 11 (30%) have between 3 and 5 consecutive lines, and 13 (35%) have more than 5 consecutive lines, whereas for the Eclipse data set, the ratios are 71%, 27%, and 1%. None of the 70 Eclipse gold standard summaries contain whole method declarations and only one contains more than five consecutive lines. We speculate that such cases with a “large chunk” of bold lines were intended for the author to refer to in the text rather than to summarize the code fragment. We eliminated

these original-summaries pairs in our experiments, leaving us with 22 code fragments in the Android corpus.

The length of the Android fragments in the final corpus was on average longer than the Eclipse fragments (Figure 4.4a). Figure 4.4b shows that if we treat the bold lines as summaries, the Android data contains longer summaries than the Eclipse data. Using the bold lines directly as a representation of a unique summary, on average, the size of a summary in the Android oracle was 21.6% of the original code fragment (the Android box-plot in Figure 4.5).

The Android data provided a generalizability challenge to the classifier developed and trained on the Eclipse FAQ. First, the code fragments demonstrate two different APIs. Second, the book author obviously did not use our annotation instruction to generate the bold lines, as our annotators for the Eclipse FAQ oracle did.

4.2 Features

The generation of extractive summaries in the textual domain is well established. There are several parallels between summarizing text and code fragments that researchers attempting to summarize source code can exploit. The selection of code lines can be formulated as a supervised classification problem, of whether to include a code line in a summary. Almost all automatic single-document text summarization algorithms for generating extractive summaries use features of the text to give a score to a sentence, for classifying whether it should be in a summary or not [33]. Many algorithms focus on query-based summaries, e.g., for the search result snippets. There are four types of sentence-independent features that have shown success in single-document summarization, as catalogued in a text summarization survey [33]: *cue phrase indicators* (such as “in this paper we show”) [91]; *query overlap criteria* (sentences with more overlap with the query tend to be important) [43]; *word frequency criteria* (if words appear unusually frequently in a piece of text, sentences containing these words tend

to be important) [51]; and *positional criteria* (topic and concluding sentences are likely candidates of important summary sentences) [43]. The first three types of features from text summarization naturally map to the source code domain:

- **Code Syntactic Features:** One way to think of cue phrases in the context of the source code is that syntactic structures are more likely to associate with important lines in a summary. An abstract syntax tree (AST) is a programming language dependent representation that captures the syntactic structure of a program.⁶
- **Query Features:** To exploit the query, we designed features that compute the amount of overlap between identifiers in the source code and words in the query.
- **Call Frequency Features:** The frequency criteria exploiting unusual words can be mapped to features exploiting unusual (infrequent) method calls.

The positional criterion, which takes advantage of topic and concluding sentences in text, does not apply to code fragments because code fragments are an arbitrary fragment of code without necessarily the enclosing structure such as the compilation unit. We confirmed this intuition after experimenting in the development set (described in Section 4.1.1). For this reason we did not further investigate positional features.

We investigated which set of AST, query, and call frequency features was the best at selecting summary lines. We used Support Vector Machines (SVM) to combine the features into a classifier. In Section 4.4, we present a comparison of an SVM classifier and a Naive Bayes classifier.

4.2.1 AST Features

We observed from the summaries in the development set that when a line contains a certain type of syntactic constructs, the line is more likely to be in

⁶The AST features are binary variables depending on whether a line exhibits a feature.

a summary. For example, a line containing an anonymous class declaration and instantiation tended to be in a summary in the oracle, whereas a line containing an *if* conditional expression tended *not* to be in a summary.

AST features take into account the syntactic structure of the code fragment. All AST features are discrete variables, with a binary value depending on whether a line exhibits a feature. We divided the 56 features into the following eleven groups:

Group 1 - Method Invocations (2 features): These two features indicate whether a line contains a part of a method invocation (*methodInvocation*) or contains more specifically the name of the method invocation (*methodInvocationName*). Distinguishing the two features is useful to capture a method invocation that spans multiple lines.

Group 2 - Methods and Signatures (5 features): The *methodDeclaration* feature indicates whether a line contains any part of a method declaration. Four remaining features capture various aspects of a method signature: *methodDeclarationName* indicates whether a line contains the name of the method declaration (as opposed to the body of the method declaration); *methodsPublic*, *methodsProtected*, and *methodsPrivate* indicate whether a line contains a method signature declared public, protected, or private, respectively. For default visibility, the three features *methodsPublic*, *methodsProtected*, and *methodsPrivate* all have the value *false*.

Group 3 - Field Declarations (3 features): The *fieldDeclaration* feature indicates whether a line contains any part of a field declaration, while *fieldDeclarationType* specifically indicates whether a line contains the type, and *fieldDeclarationName* indicates whether a line contains the name.

Group 4 - Control Flow Constructs (11 features): The *ifBlock*, *elseBlock*, *forBlock*, and *whileBlock*⁷ features indicate whether a line contains any part of the block in an *if*, *for*, or *while* statement, while *ifConditional*, *forConditional*, and

⁷We mistakenly did not include *do...while* loops. We corrected this oversight in Konaila's control flow salience filter (Section 6.3.3).

whileConditional only capture the conditional expression, and *ifKeyword*, *elseKeyword*, *forKeyword*, and *whileKeyword* only capture the keyword.

Group 5 - Return Statements (3 features): *returnStatementSimple* indicates whether a return statement returns a variable, boolean value, or null, while *returnStatementNothing* is for void return statements and *returnStatement* is for any return values.⁸

Group 6 - Variable Declarations (4 features): *variableDeclaration*, *variableDeclarationName*, and *variableDeclarationType* apply to any variable declaration, indicating whether a line contains any part of a variable declaration. The *primitiveDeclaration* feature applies to declaration of a variable of a primitive type.

Group 7 - Types and Signatures (12 features): Four features apply to the whole type (*typelsPublic*, *typelsProtected*, and *typelsPrivate*). For package visibility, the three features *typelsPublic*, *typelsProtected*, and *typelsPrivate* all have the value *false*. *superClassType* indicates whether a line contains any part of a class with a declared super-class. Eight of the features indicate whether a line contains various keywords in the class signature: *implementsKeyword*, *extendsKeyword*, *classKeyword*, *interfaceKeyword*, *typePublicKeyword*, *typeProtectedKeyword*, *typePrivateKeyword*, and *typeDeclarationName*.

Group 8 - Anonymous Classes (3 features): *anonymousClassDeclaration* indicates whether a line contains any part of an anonymous class declaration, while *anonymousClassSuperTypeName* more specifically captures the signature. *methodInvocationOnAnonymous* indicates whether a line contains a call with at least one argument that is an anonymous class creation.

Group 9 - Comments (3 features): These features (*javaDocComment*, *blockComment*, and *lineComment*) indicate whether a line contains a part of the three types of comments.

Group 10 - Exception Handling (7 features): Three features capture lines in exception blocks (*tryBlock*, *catchBlock*, and *finallyBlock*), while three others capture

⁸*returnStatement* is a super-set of *returnSimple*.

the keywords (*tryKeyword*, *catchKeyword*, and *finallyKeyword*). Finally, *thrownExceptionDeclaration* captures the `throws` clause in a method signature.

Group 11 - Other (2 features): The *importKeyword* and *assignment* features indicate whether a line contains an import statement or an assignment, respectively.

Generating these features required constructing an AST of a code fragment. Building an AST for incomplete programs such as a code fragment poses a challenge for many popular parsers, such as the Eclipse Java Development Toolkit parser we used. In our corpus, only fewer than 10% of the fragments form full compilation units. In addition, the Eclipse parser is not able to construct an AST when a code fragment contains invalid constructs such as "...", a common convention to indicate a part of the code not worth showing. To solve the first problem, we built ASTs for multiple scenarios: the code fragment as it is, the code fragment put in an empty Java compilation unit stub, and the code fragment put in an empty method declaration stub in a Java class, and measured which scenario resulted in the most complete AST. In this context, an AST is more complete than another when there are more code constructs (e.g., method invocations, method declarations, exception blocks, and conditional statements) that were recognized by the parser. We also engineered the algorithm to remove invalid Java tokens such as "...". With this algorithm, we were able to parse the code fragments and constructed corresponding ASTs for over 90% of the fragments. However, this result is not necessarily generalizable to code fragments extracted from arbitrary web pages, as we might not have exhaustively considered all possible invalid Java constructs that exist in any code fragments on the web. In the second summarization system we built, Konaila, we took a different approach in parsing, by specifying precisely the code fragments Konaila accepts (Section 6.2.1).

Because of the relatively large number of the AST features, we performed a Multiple Correspondence Analysis (MCA) to see whether the features were associated with each other. MCA applies to categorical features and is considered

4.2 Features

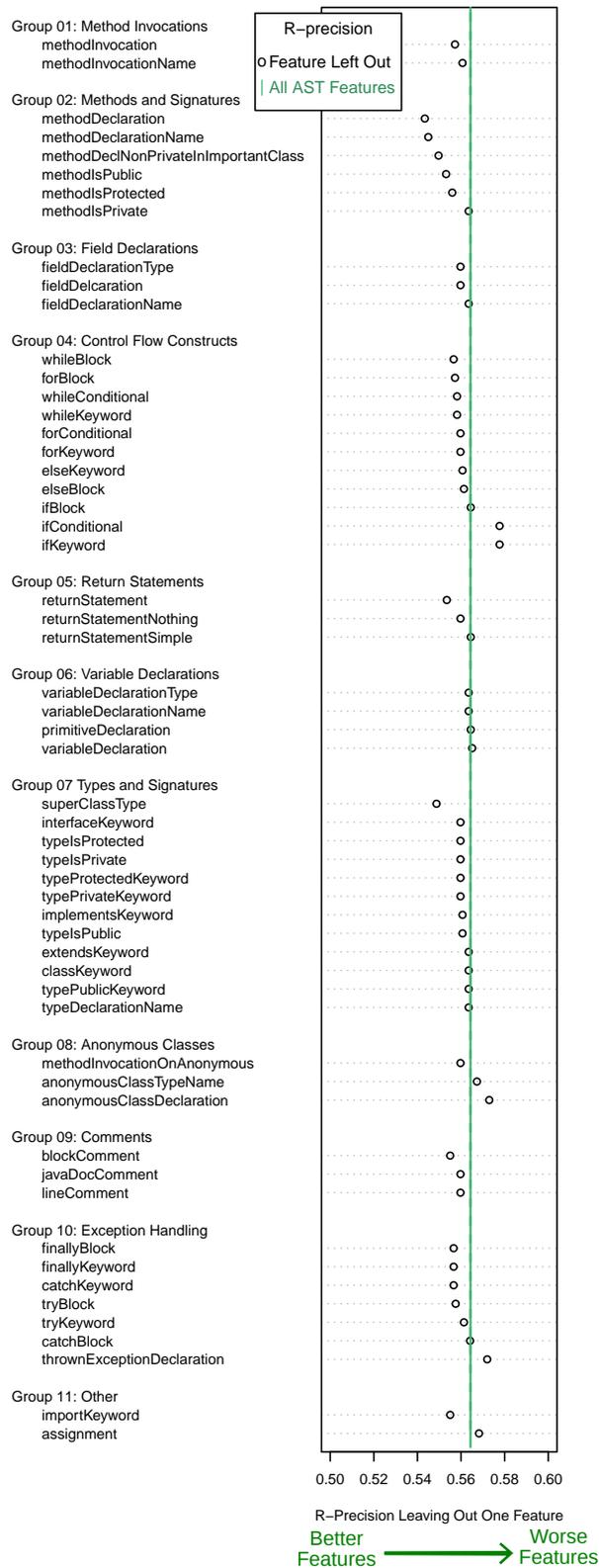


Figure 4.6: Leave-one-feature-out experiment on the AST features

the counter-part of Principle Component Analysis for categorical data. Intuitively, MCA projects the data from the original feature space (AST features in our case) to a lower dimensional space that maximizes the variance of the data. If a small number of dimensions capture close to 100% of the variance of the data, then many of the AST features are associated with each other, and the summary classification problem can benefit from a classifier using the smaller set of MCA dimensions rather than the full feature set. For this MCA analysis, we used the “mca” function from the MASS package in R.⁹ MCA revealed that using one dimension captured 9.59% of the variance, whereas two dimensions captured 17.8% cumulatively and 24.5% for three¹⁰ dimensions. We did not consider this level of variance sufficiently high to conclude that the summary classification problem could benefit from a classifier using the MCA dimensions.

Instead, we turned to an ablation study to find out which features were more important to the summary-line classification problem. In the ablation study we ran the classifier multiple times, each time leaving out one single feature. Figure 4.6 shows the prediction precision of each of the leave-one-feature-out run, indicating the relative strength of the features. The more a feature degraded R-precision when left out, the better it is. The solid vertical line indicates the prediction precision when all the 56 AST features were used in the classifier. Six features were obviously poor features whose existence degraded the performance of the classifier: *ifConditional*, *ifKeyword*, *anonymousClassSuperTypeName*, *anonymousClassDeclaration*, *thrownExceptionDeclaration*, and *assignment*. This analysis also showed that 13 features, when removing each one, did not degrade the performance: *methodsPrivate*, *fieldDeclarationName*, *ifBlock*, *returnStatementSimple*, all the four features in the variable declarations group, *extendsKeyword*, *classKeyword*, *typePublicKeyword*, *typeDeclarationName*, and *catchBlock*.

⁹<https://cran.r-project.org/web/packages/MASS/index.html>

¹⁰Figure 4.7 plots the percentage of variance for each successive MCA dimension added to the projected space. The percentage of variance reduced abruptly when adding the fourth dimension. Hence we chose to stop the analysis at three dimensions.

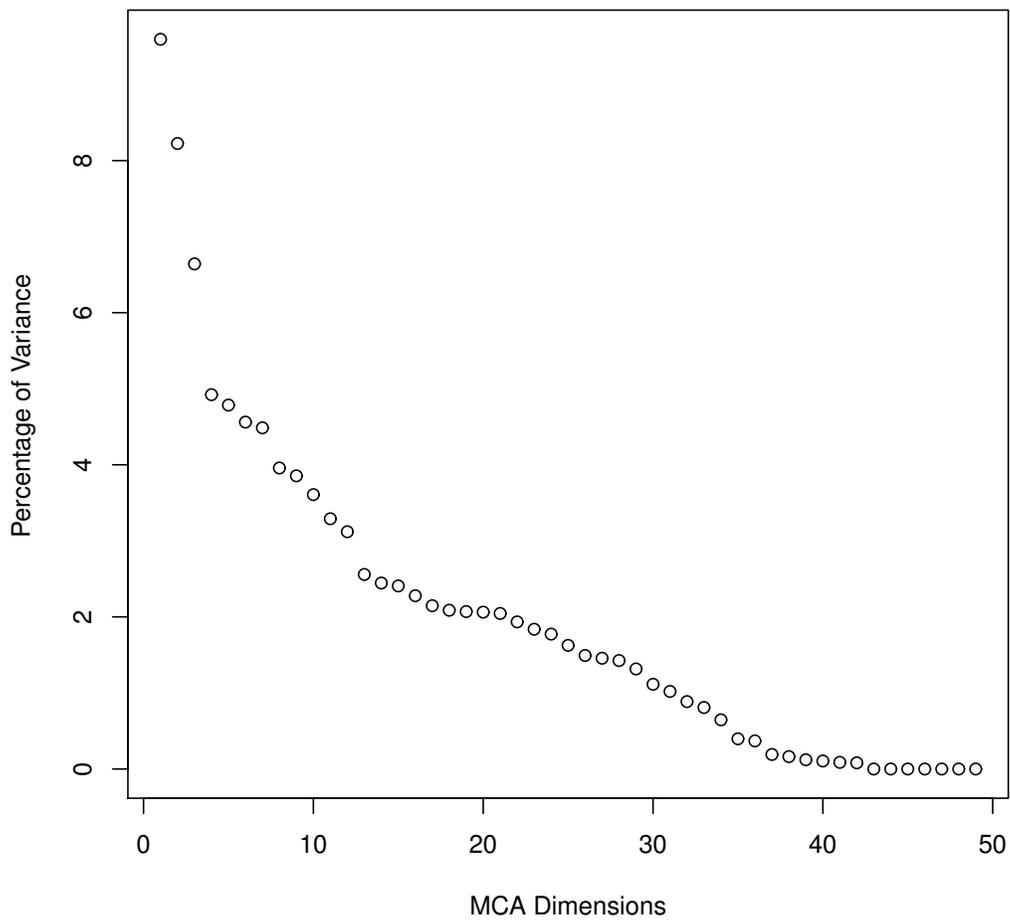


Figure 4.7: Percentage of variance for each successive MCA dimension added to the projected space.

The results of the ablation study raised the following more specific research questions when investigating the combination of features for RQ 1:

RQ 1.1: Would a more compact AST feature set (called ***ASTCompact***) perform better than the full set (called ***ASTAll***)? We defined the compact set to only consist of those features that degraded the performance (the features strictly left of the “All Features” line in Figure 4.6) from the leave-one-feature-out analysis.

RQ 1.2: Would a set larger than ***ASTCompact*** (called ***ASTReduced***) perform better than ***ASTAll***? We defined ***ASTReduced*** as the set of features that did not increase performance when left out (the features on or left of the “All Features” line in Figure 4.6).

RQ 1.3: What is the contribution of AST features (either ***ASTAll***, ***ASTCompact***, or ***ASTReduced***) to the overall classification performance?

4.2.2 Query-related Features

We also observed that annotators are more likely to include in the summary the lines containing the terms from the query. Analogous to the syntactic features, query related features are discrete variables, with a binary value depending on whether a line contains a feature. We constructed four query-related features. Two features (*methodInvocationNameContainsQuery* and *variableDeclarationNameContainsQuery*) that indicate whether an identifier (method invocation or variable name respectively) in a code fragment contains a query term or not, defined as follows. We constructed two additional features that looked beyond just individual lines: *mostTerms* is true when a line contains the most number of matching terms among all the lines in the same code fragment. *mostDiverseTerms* is analogous except it indicates the most number of distinct terms. When computing the

terms, we split the identifiers according to the common camel case identifier naming convention.¹¹ The research question we were interested in was,

RQ 1.4: How much do the four query features (denoted as *Query*) contribute to the prediction performance?

4.2.3 Call Frequency Features

Inspired by the success of the frequency criteria in text summarization that gives more weight to sentences with more unusual words, we wanted to see if the same idea could be applied to method calls: Were unusual methods in a code fragment more important for a summary?

One technical challenge in generating these features was to qualify the method calls in a code fragment correctly, because resolved fully-qualified type references to program elements, called type bindings, are seldom available in code fragments. Fully qualified names include the project and package information to help unambiguously identify type references (see Section 3.1.1). To find these missing type bindings, we used Partial Program Analysis (PPA), a technique for resolving such bindings [19]. Because PPA was designed to be used with entire files retrieved from code repositories, PPA requires programs with proper import statements. We used the Eclipse “Organize Import” facility¹² to attempt to infer import statements.

We explored three features that characterize method calls:

API calls: API calls that are rarely called in general but called in a given code fragment tend to be important. For example, the API method `PlatformUI.getWorkbench()` is commonly used by programmers in general whereas `IWorkbench.getHelpSystem()` is rarely used. To account for this insight,

¹¹For example, for the identifier “getCodeWords”, the algorithm produces three terms “get”, “code”, and “word” while for “ASTVisitor” it produces the terms “ast” and “visitor”.

¹²<http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fui%2Factions%2F0organizeImportsAction.html>

we created a feature that counted the number of times the method is being used *universally*, if a code line contains a call to an API method call. To capture this notion, we needed a large sample code corpus and analyze its method call frequencies statically. We compiled such a corpus from the three largest official Eclipse CVS repositories: Eclipse platform¹³ (188 KLOC in Java), Eclipse technology¹⁴ (217 KLOC), and Mylyn¹⁵ (112 KLOC), retrieved on July 2012. We generated a feature called *DfPackage* (with “Df” inspired by the word document frequency) by counting how many times another Java package in the sample code corpus calls a method contained in a line. We discretized the count into three values: *rare*, *common*, and *in_between*. The thresholds for these categories were determined in the development set. Determining a parameter value in the development set (Section 4.1.1) and using the same parameter value in the evaluation set is a standard practice in machine learning experiments [55, p.262]. The thresholds determined were above 150 for *common* and less than or equals to 50 for *rare*.

Local calls: We defined a feature called *DeclaredInCode* that is true if a line contains a method call whose method declaration is in the code fragment.

Java calls: We defined a feature called *Java* to indicate whether a line contains a call to the Java standard library JDK. This computation is a static analysis. In the presence of dynamic dispatch, we can only infer that the call is to a Java standard library JDK assuming the run-time type of the object matches its statically-declared type. (Section 3.1.1).

The research question we were interested was,

RQ 1.5: How much do the *CallFreq* features (*DfPackage*, *DeclaredInCode*, and *Java*) contribute to the performance?

¹³at CVS location “:pserver:anonymous@dev.eclipse.org:/cvsroot/eclipse”

¹⁴at CVS location “:pserver:anonymous@dev.eclipse.org:/cvsroot/technology”

¹⁵at CVS location “:pserver:anonymous@dev.eclipse.org:/cvsroot/mylyn”

4.3 Feature Investigation Results

For RQs 1.1-1.5, the performance results in this section were computed using a leave-one-out cross validation. In each cross validation fold which corresponded to a code fragment in the corpus, a summary was generated by a classifier. For each fold, we calculated R-precision, which is described below. The final result of the cross validation was the average of all the folds. Using a leave-one-out cross validation set-up maximized the use of the original-summary pairs for training the classifier, compared with the commonly used 10-fold cross validation.

4.3.1 Cross-Validation Settings

For the **within-API** results, we used the Eclipse corpus. To maximize the training data, we adopted a common practice in machine learning where we trained the summary-classifier on both the development (17 original-summary pairs) and the evaluation sets (53 pairs). As a result, the cross validation consisted of 53 folds, each fold yielded a prediction for one summary trained on 69 gold standard summaries (17 from the development set, plus 53 minus 1 from the evaluation set). This set-up made the most out of our training data but did not threaten classifier over-fitting because all the predictions still applied to the code fragments in the unseen evaluation set. For each fold (one code fragment), we used the gold standard summaries (Section 4.1) as the ground truth in the evaluation of the generated summaries. For the **cross-API** results, the training data was the Eclipse corpus while the test data was the Android corpus. More specifically, we trained the classifier on all 70 summaries in the Eclipse oracle. We then computed the performance metric on the 22 code fragments from the Android corpus.

4.3.2 Metric: R-Precision

To evaluate how well a generated summary resembles a summary in the oracle, we compared the two using **R-precision**, an evaluation metric from the field of information retrieval [55]. R-precision is similar to precision-at-k, a more familiar precision metric for summaries of length k. Precision-at-k determines out of the top k lines predicted by our classifier ($predicted_k$), how many are correct ($|oracle \cap predicted_k|$). R-precision differs from precision-at-k by allowing summaries of variable lengths. The R-precision of a code fragment evaluates the top R lines returned by our classifier ($predicted_R$), where R is the length of the summary oracle. More formally, R-precision is given by $\frac{|oracle \cap predicted_R|}{|predicted_R|}$. R-precision is more stable than precision-at-k, which is highly dependent on the total number of lines marked in the oracle. For example, in the Android book oracle, summaries can be as long as 12 lines (Figure 4.4b). In such cases, pre-defining generated summaries at three lines can never achieve perfect recall even for a perfect classifier.

4.3.3 Results

Answering the five research questions required comparing various combinations of the features. The comparison was determined by evaluating two aspects of the classification performance: **within-API** in which R-precision was calculated using the classifier trained and tested on the Eclipse corpus, and **cross-API** in which R-precision was calculated using the classifier trained on the Eclipse corpus, and tested on the Android corpus. Figure 4.8 summarizes the results comparing the AST feature sets (RQs 1.1-1.3). Figure 4.9 summarizes the results for the query feature set (RQ 1.4) and Figure 4.10 for the call frequency feature set (RQ 1.5).

Given the differences in the two corpora, we expected the average R-precision on the Android corpus to be significantly lower for two reasons. First, the generation of oracle of summaries from the Android book corpus and the Eclipse FAQ corpus were different (retaining bold lines marked by the authors

4.3 Feature Investigation Results

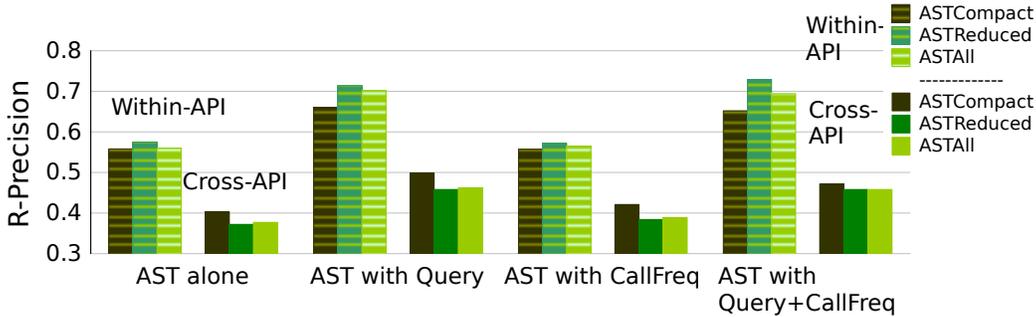


Figure 4.8: Comparing the AST feature sets. The statistically significant results for within-API were as follows: **ASTReduced+Query** (R-precision=0.714) > **ASTCompact+Query** (0.660), **ASTReduced+Query+CallFreq** (0.729) > **ASTCompact+Query+CallFreq** (0.652), **ASTReduced+Query+CallFreq** (0.729) > **ASTAll+Query+CallFreq** (0.694). For cross-API, the only statistical significant result was **ASTCompact+Query** (0.498) > **ASTReduced+Query**.

versus explicitly instructing annotators to provide summaries). Second, the nature of the code fragments was different: the code fragments in the Eclipse corpus were intended to answer FAQs and the code fragments in the Android corpus may not necessarily have been intended to answer a specific question. Degradation in performance when generalizing a classifier is inherent to any machine learning approach.

Results on AST Features (RQs 1.1-1.3)

Figure 4.8 presents an overview on the results for RQs 1.1-1.3 on the contribution of different AST sets (**ASTCompact**, **ASTReduced**, **ASTAll** in Section 4.2.1) to the prediction performance. The results for the within-API experiments are marked with bars with stripes, and the results for the cross-API experiments are marked with bars without stripes.

Research questions 1.1 and 1.2 concerned how the AST features performed without combining with the **Query** nor **CallFreq** features: whether the **ASTCompact** set (bars with the darkest shade) performed better than **ASTAll** (bars with the lightest shade), and whether **ASTReduced** (medium shade) performed better than **ASTAll**. The results relevant to these two questions were the group of bars

4.3 Feature Investigation Results

marked as “AST alone” in Figure 4.8. The R-precision of the three AST sets alone were *not* statistically significantly different from one another:

- For within-API predictions (bars with strips), the R-precision of each of the three sets (***ASTCompact***, ***ASTReduced***, and ***ASTAll***) was 0.558, 0.575, 0.560, respectively. Wilcoxon signed rank tests showed that there were neither a statistically significant difference between ***ASTCompact*** and ***ASTAll*** nor between ***ASTReduced*** and ***ASTAll***.
- Similarly, for cross-API predictions (bars without strips), the average R-precision numbers of the three sets were 0.403, 0.372, 0.377, respectively. There was neither a statistically significant difference between ***ASTCompact*** and ***ASTAll*** nor between ***ASTReduced*** and ***ASTAll***.

Research question 1.3 concerned how the AST features performed in combination with ***Query*** and/or ***CallFreq***. The results relevant to this question are in the group of bars marked as “AST with Query” and the group marked as “AST with Query+CallFreq” in Figure 4.8. *When combined with the query features (either with ***Query*** or ***Query+CallFreq*** features), ***ASTReduced*** performed the best overall for within-API predictions, while ***ASTCompact*** performed the best for cross-API predictions:*

- For within-API predictions, when combined with ***Query*** (the “AST alone” group of bars), ***ASTReduced***’s average R-precision (0.714) was statistically significantly better than ***ASTCompact***’s (0.660), with $p = 0.0105$, although ***ASTReduced***’s R-precision was not statistically significantly better than ***ASTAll***’s (0.702). When combined with ***Query+CallFreq***, ***ASTReduced*** (0.729) was statistically significantly better than ***ASTCompact*** (0.652) with $p = 0.00972$, as well as ***ASTAll*** (0.694) with $p = 0.0489$.
- For cross-API predictions, when combined with ***Query***, ***ASTCompact***’s average R-precision (0.498) was statistically significantly better than ***ASTReduced***’s (0.458), although not statistically significantly different from ***ASTAll***’s (0.462). When combined with ***Query+CallFreq***, the average

4.3 Feature Investigation Results

R-precision numbers of **ASTCompact**, **ASTReduced**, and **ASTAll** (0.471, 0.458, and 0.458 respectively) were not statistically significantly different from one another.

When combined with **CallFreq** (“AST with CallFreq”), the average R-precision numbers of the three AST sets were not statistically significantly different from one another. This is the case for both within-API predictions (the average R-precision numbers for **ASTCompact**, **ASTReduced**, and **ASTAll** were 0.557, 0.572, 0.564, respectively) and cross-API predictions (the average R-precision numbers for the three sets were 0.421, 0.384, 0.389 respectively). The pair-wise differences among the three sets were not statistically significant.

Overall, the cross-API results on the Android corpus were in the range of 0.3-0.5. We argue that this level of precision was acceptable given the challenge in cross-corpus predictions. A random classifier would generate summaries of R-precision of 0.216, the summarization factor (the average percentage of lines from the code fragments that are in a summary) for the Android book corpus (Figure 4.5).

Results on Query (RQ 1.4)

For understanding the contribution of the **Query** set to the performance of the summary-line classification problem, we report on different combinations of the feature sets. For the AST features, we employed **ASTReduced** following from the results from RQs 1.1-1.3.

With just **Query** alone, the average R-precision was 0.540 for within-API predictions and 0.341 for cross-API. Figure 4.9 presents the results on the performance contribution of **Query**. Adding the **Query** feature set (bars with a lighter shade) consistently increased the average R-precision compared with the ones without **Query** (bars with a darker shade) in all six settings:¹⁶

¹⁶The six settings account to the three combinations with **ASTReduced** and/or **CallFreq**, each combination with two settings, within- and cross-API.

4.3 Feature Investigation Results

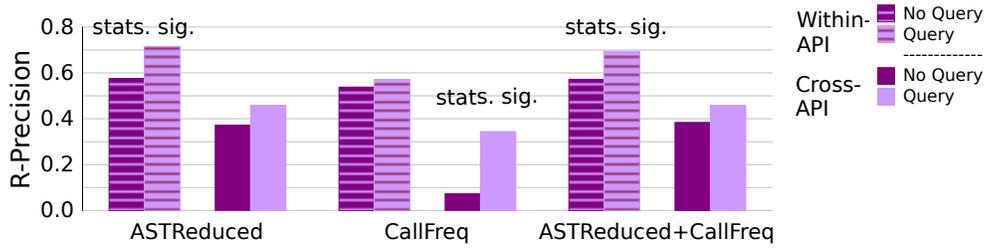


Figure 4.9: Six comparisons on the contribution of the **Query** feature set to prediction performance measured in average R-precision. From the leftmost bars to the right, the six pairs of average R-precision *without Query* and *with Query* are (0.575, 0.714), (0.372, 0.458), (0.538, 0.570), (0.073, 0.344), (0.572, 0.694), (0.384, 0.458). The pairs in bold had statistically significant differences.

- For the within-API predictions (bars with strips), adding the **Query** features set resulted in two statistically significant increases. When combining with **ASTReduced**, **Query** increased the average R-precision from 0.575 to 0.714 ($p = 0.00124$), and when combining with **ASTReduced+CallFreq**, **Query** increased the average R-precision from from 0.572 to 0.694 ($p = 0.00247$). However, the performance increase of **Query** when combined with **CallFreq** (from 0.538 to 0.570) was not statistically significant.
- For the cross-API predictions (bars without strips), adding **Query** resulted in one statistically significant increase, when combining with **CallFreq**. The increase was from 0.073 to 0.344 ($p = 0.00142$). This result contrasts with the result in the experiment comparing the AST feature sets (Figure 4.8) in which **CallFreq** did not increase the prediction performance. We suspect that there is an overlap between **CallFreq** and a subset of the AST features (i.e., method invocation features); therefore, adding **CallFreq** to **Query** was fruitful but only without the AST features.

4.3 Feature Investigation Results

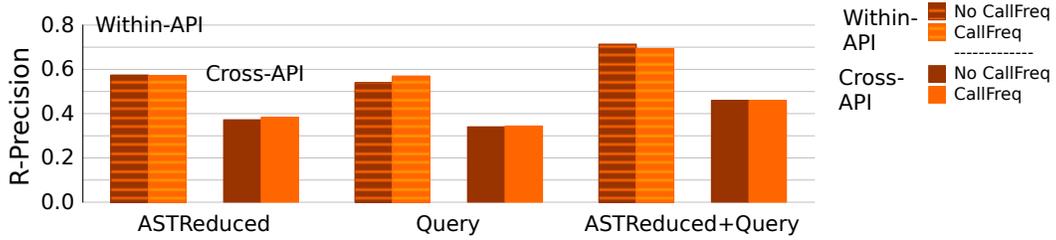


Figure 4.10: Six comparisons on the contribution of the **CallFreq** feature set to the prediction performance measured in R-precision. From left to right, the six pairs of R-precision *without* **CallFreq** and *with* **CallFreq** are (0.575, 0.572), (0.372, 0.384), (0.540, 0.570), (0.341, 0.344), (0.714, 0.694), (0.458, 0.458). None of the pairs were statistically significantly different.

Results on Call Frequency Features (RQ 1.5)

The average R-precision of the **CallFreq** feature set alone was 0.538 and only 0.073 for cross-API predictions. As for the results on the performance contribution of the **CallFreq** feature set, (Figure 4.10), **CallFreq** (bars with a lighter shade) did not contribute significantly to the classification performance.

- For the within-API results, **CallFreq** increased the average R-precision in one case, when combining with **Query** (from 0.540 to 0.570). However, **CallFreq** degraded the average R-precision when combining with **ASTReduced+Query** (from 0.714 to 0.694) and when combining with **ASTReduced** (from 0.575 to 0.572). Wilcoxon tests showed that these changes were not statistically significant.
- For the cross-API results, **CallFreq** increased the average R-precision also in one case when combining with **ASTReduced** (from 0.372 to 0.384), while **CallFreq** did not practically change the R-precision in the two other combinations. Wilcoxon tests showed that these changes were again not statistically significant.

Summary of the Feature Investigation Experiment

The conclusion from RQs 1.1-1.3 was that feature selection paid off both for the within-API and cross-API predictions. The **ASTReduced** set was a clear winner for within-predictions, while there was some evidence that **ASTCompact** generalized better than **ASTReduced**. This result indicates that the full AST set was over-fitting. The implication is that the features in **ASTCompact**, which was also a subset of the **ASTReduced**, were the most crucial in the construction of a generalizable summary classifier. These **ASTCompact** features covered ten of the eleven groups of AST constructs (Figure 4.6): method invocations, methods and signatures, field declarations, control flow constructs, return statements, types and signatures, anonymous classes, comments, exception handling, and import statements. On the other hand, variable declarations and statements containing assignments were less crucial in the construction of a generalizable summary classifier. For RQs 1.4 and 1.5, the conclusion was that the **Query** features were crucial in the classification performance, while the **CallFreq** features were not.

Given these results, for the rest of the experiments in this chapter (unless indicated otherwise), the classifier we employed consist of a combination of the **ASTReduced** and **Query** feature sets. Table 4.1 presents the time it took to generate these features. The machine used to generate the features had a 64-bit Intel Duo Core 3.33GHz processor with 7.7G of memory. On average, generating these features for a code fragment took 0.09 second, making it possible to deploy in a real application setting.

4.4 Effectiveness Experiment

For RQ 2, we compared the SVM classifier constructed with the best feature set, **ASTReduced+Query**, against three baselines:

- the *first-N-lines* classifier which constructs a summary of length N by selecting the first N lines of a code fragment,

Table 4.1: Running time for feature generation

	# code frags	# lines	Running time		
			total	per code frag	per line
Eclipse dev	17	225	1.6 s	0.09 s	7 ms
Eclipse eval	50	602	3.8 s	0.07 s	6 ms
Android	23	641	2.4 s	0.10 s	4 ms
	90	1468	7.8 s	0.09 s	5 ms

- the *last-N-lines* classifier which selects the last N lines,
- and the *SVM-query-only* classifier that uses the two query features (*most-Terms* and *mostDiverse*) that do not require any AST construction.

We also experimented with a Naive Bayes classifier (NB) using the **ASTReduced+Query** feature set. In this experiment, we focused on the within-API setting (Section 4.3.1) on the Eclipse corpus (Section 4.1.1).

Metrics

We conducted the performance comparison through a receiver operator characteristic (ROC) curve. An ROC curve depicts the trade-off between the true positive rate and false positive rate as we varied N from one line to four lines. This curve enables us to understand explicitly the performance trade-off among different summary lengths. The coordinate of a point on the curve is given by the average true positive rate (average of each of the true positive rate per code fragment) and average false positive rate (average of each of the false positive rate per code fragment). The *true positive rate* of a code fragment c is given by:

$$\frac{|\{\text{lines in gold st. summary of } c\} \cap \{\text{lines in gen. summary of } c\}|}{|\{\text{lines in gold st. summary of } c\}|}$$

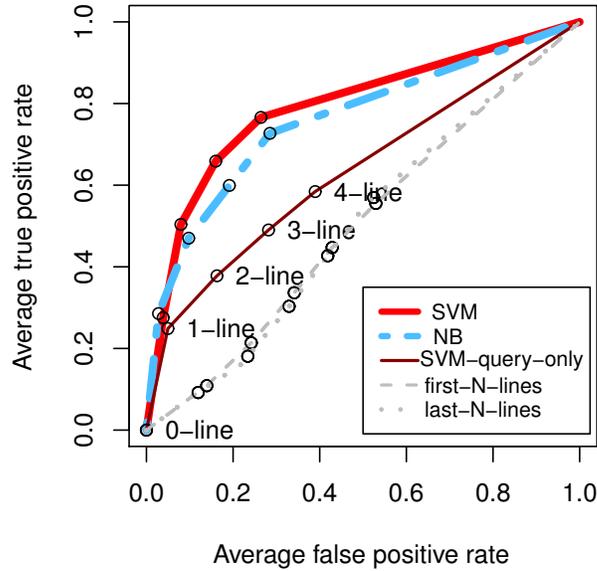


Figure 4.11: ROC curves

The *false positive rate* of a code fragment c is given by:

$$\frac{|\{\text{lines in } c \text{ not in gold st. summary}\} \cap \{\text{lines in gen. summary of } c\}|}{|\{\text{lines in } c \text{ not in gold st. summary}\}|}$$

Averaging the rates per code fragment (rather than for all lines) aligns better with the actual task of providing a summary for a code fragment (rather than being just an exercise of predicting summary-membership of lines). The closer the ROC curve is to the upper left corner (with fewer false positives and more true positives), the better the classifier. The area under the curve sums up this intuition: the better the classifier, the closer its area under a ROC curve is to 1.

Results

Figure 4.11 shows five ROC curves: two versions of our classifier (*SVM* and *NB*, the two thicker lines) and three baselines (*SVM-query-only*, *first-N-lines*, and *last-N-lines*, the three thinner lines). Our two classifiers have area under the

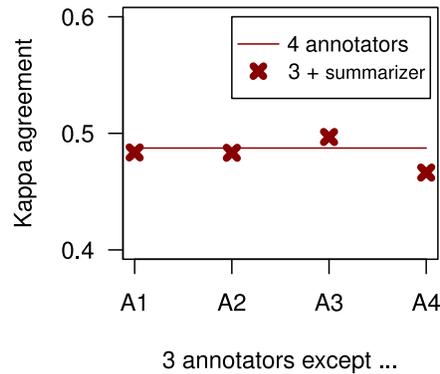


Figure 4.12: Agreement result

curve of 0.806 for *SVM* and 0.772 for *NB*. Both clearly lie above the *first-N-lines* baseline (the area under the curve is 0.493) and *last-N-lines* baseline (the area under the curve is 0.503). In addition, the two classifiers using both syntactic and query features out-perform the baseline using only query-related features, *SVM-query-only*, whose area under the curve is 0.629. The conclusion is that it is feasible to generate effective code fragments summaries using only syntactic and query-related features that are better than the baselines.

4.5 Generated Vs. Annotators' Summaries

To better interpret the level of precision reported, in RQ 3, we compared our automatically generated summaries with summaries constructed by the annotators. As in RQ 2, we based the results on this experiment on the within-API setting (Section 4.3.1) on the Eclipse corpus (Section 4.1.1). Figure 4.12 illustrates the Kappa agreement of the four annotators (kappa=0.487) and how the agreement changed when we left out the summaries provided by each one of the annotators and replaced the summaries of the left-out annotator with summaries generated by the classifier. In one case (A3), swapping in the generated summaries even *improved* the agreement, whereas in two cases the agreement decreased slightly and in the third case decreased more. The average of the four kappa statistics of the three-annotators-plus-our-classifier

settings was 0.484, almost the same as the agreement of the four annotators. The automatically generated summaries were *as similar to human-generated summaries as* summaries generated by different humans to each other.

4.6 Overall Quality

R-precision evaluates how many of the predicted lines matched the lines in the oracle summaries. However, the quality of a summary is not only affected by the number of correctly predicted lines, but also the quality of the summary as a whole, such as cohesion of the summary. For evaluating the summaries qualitatively, we wanted to specifically look for cohesion problems. All the features described in this chapter were not designed to handle cohesion. To gain a better understanding on what type of cohesion problems that could arise, we performed a qualitative analysis on the summaries (Section 4.6.1).

To evaluate beyond individual lines, we also experimented with the use of a metric called pyramid precision. Intuitively, R-precision weighs all the predicted lines equally, whereas pyramid precision puts more weight on lines selected by more annotators. Text summarization experiments have shown that pyramid precision corresponds better than precision in terms of the *overall* quality of the summary, when a human is asked to compare the overall quality of two summaries [67]. We report on a comparison of the pyramid precision results with the R-precision results in Section 4.6.2.

4.6.1 Qualitative Analysis

We present some insights into the quality of each summary as a whole through a qualitative analysis. Because judging the quality is a difficult task, we evaluated the summaries by comparing pairs of summaries of two settings: **AST+Query** and a baseline of only using the **Query** features. We chose the **Query** setting as a baseline because it was similar to the approach used in current search engines that extract code snippets in a result page.

We rated whether the **AST+Query** summary was better than, worse than, or the same as the **Query** summary. Of the 53 summaries in the evaluation set of the Eclipse corpus, for the purpose of the manual analysis, we eliminated 13 in which both feature sets produced exactly the same summaries, leaving us with 40 pairs for the analysis. In 27 out of 40 pairs, we arrived at a decision from considering the *individual* lines. For the rest of the 13 pairs, the decision required considering the summaries *as a whole*:

- In seven cases, a summary was worse because a line with a variable use should be in the summary together with a variable declaration. These cases are addressed in a subsequent summarization approach we proposed in Section 6.3.4.
- In three cases, the judgement depended on whether it was appropriate to show the structure (signatures) or not.
- In two cases, a statement spread into two lines were cut off in the summary.
- In one case, a summary was worse because redundant lines were included.

These four types of cases considering the quality of the summary as a whole are improvement opportunities for a summary classifier.

4.6.2 Pyramid Precision Results

Pyramid precision weights the more-agreed-upon summary lines more heavily, whereas the gold standard summaries used in R-precision disregard summary lines which are less agreed upon. This can be problematic for the code fragment summarization problem which naturally has low agreement. Pyramid precision emphasizes the lines more agreed upon by the annotators while de-emphasizing the lines less agreed upon. More specifically, pyramid precision produces a score that weighs more heavily the summary lines with higher agreement, and then normalizes the score by the highest possible score a summary of the same

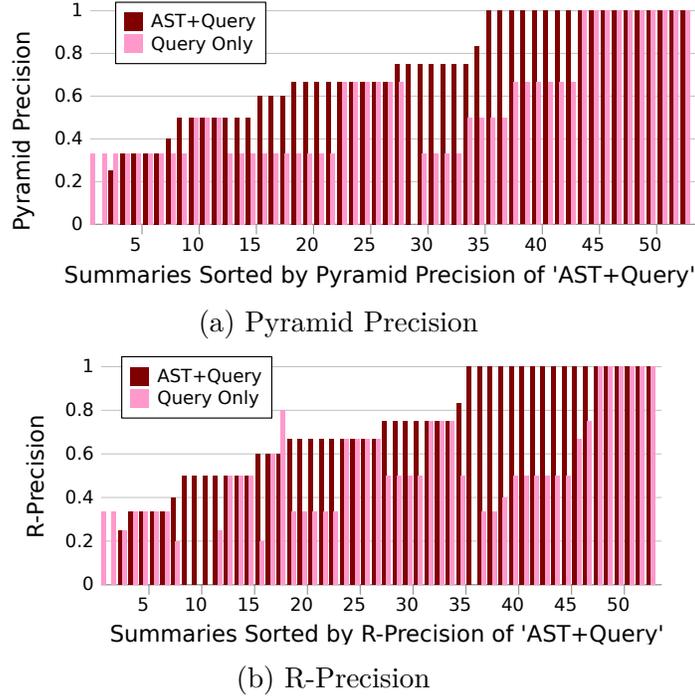


Figure 4.13: Comparing the **AST+Query** and **Query** summaries using two metrics

length can attain. The score of a generated summary s of length n , $score(s)$, is the sum of the scores of each line in a summary s , and the score of a line is defined as the number of annotators who marked the line as important in a summary. In our case, the best score for a code line is 4 (when all four annotators marked the line) and the worst score is 0 (when none marked it). Pyramid precision is $score(s)$ normalized by the score of the best possible summary s_{best} of length n : $\frac{score(s)}{score(s_{best})}$. The best possible summary is the set of the top n lines, if we sort all the lines in a code fragment by the number of annotators who marked the line as in-summary, or, $s_{best} = \max_{s \in S_n} score(s)$ where S_n is the set of all possible summaries of length n . Pyramid precision does not explicitly evaluate how a generated summary resembles any single one of the summary provided by an annotator, but how a generated summary resembles the best possible summary derived from the different versions of the summary provided by the annotators.

The average pyramid precision for 3-line summaries using **AST+Query**¹⁷ features for the within-API precisions was 0.745 as compared to 0.550 when using query features alone, with the full distribution in Figure 4.13a. To provide a point of reference, we also provide the R-precision distribution in Figure 4.13b. There are two code fragments with R-precision or pyramid precision=0. Surprisingly, there is only one other case where R-precision of query-based feature is higher (0.8) than **AST+Query** (0.6). There are seven cases where pyramid precision yield same results for both versions as R-Precision. Interestingly, the two metrics both appear to measure the same construct. The Pearson correlation coefficient of the two metrics on the **AST+Query** summaries was 0.857, indicating a linear relationship with statistical significance ($t = 11.9, p = 2.22 \times 10^{-16}$).

4.7 Chapter Summary

Selecting important parts of a code fragment is a crucial step in any type of summarization system. In this chapter, we reported on a case study on this selection step, via the design, implementation, and evaluation of a machine learning approach on the summary line classification problem. We report on four lessons learned which we subsequently addressed in the human study (Chapter 5) and the construction of Konaila (Chapter 6): the best performing feature set being the combination of syntactic and query-related features, and three limitations from the machine learning based line-based summarization approach. These limitations were in using line as the granularity, difficulty in obtaining training data with high quality, and only using features that were local to a line without considering dependencies among different parts of the code. The lessons learned from this case study constitute the first contribution of this dissertation.

¹⁷We used **AST** rather than **ASTReduced** for this comparison.

Combination of syntactic and query features, but not call frequency: We found in RQ 1 (Section 4.3) that a combination of syntactic constructs (*ASTReduced* or *ASTCompact* features) in a code example and the amount of overlap with a query (*Query* features) produced better summaries than using syntactic or query overlap alone for within-API and cross-API summary-line predictions. Not only did the call frequency features (*CallFreq*) not increase the quality of the summary, they were computationally costly to generate as the computation involved the use of a sophisticated program analysis algorithm (Partial Program Analysis [19]) to resolve type bindings in a code fragment. Without those features, the syntactic and query-related features are fast to generate (0.09s per code fragment), making it possible to deploy in a real application setting. These insights were key in the design of Konaila in Chapter 6.

Limitation on line granularity: As a first attempt at code fragment summarization, we formulated the problem as a line-based classification problem (Chapter 3). As the qualitative analysis highlighted (Section 4.6.1), line-based summaries did not always result in sensible summaries, for example, when only one of lines from a multi-line statement was selected for a summary. Fundamentally, the line granularity does not align with the structure of code. However, what is the right granularity? We report on additional insights on the granularity issue in the empirical study (Chapter 5), as well as a new unit of analysis called selection unit in our subsequent attempt at designing a summarization technique, Konaila (Chapter 6).

Difficulty in obtaining data of high quality: In RQ 3 (Section 4.5), we did not find a degradation in the annotators' agreement when we replaced the summaries from each of the four annotators with our automatically generated summaries. This result suggests that given this data, the classifier we reported in this chapter is an effective classifier for the line-based summarization problem. However, the Kappa agreement of the four annotators (0.487) was moderately low, indicating a significant disagreement among the annotators on which lines

are important for a summary. There is a limit to the performance one could expect from a supervised machine learning approach, a data-driven approach, that relied on this same set of annotations. In Chapter 6, we report on Konaila which does not depend on annotated data.

Limitations on local features: The three types of features we investigated were all features local to a line not considering dependencies among different parts of the code. As we saw from the qualitative analysis, this decision directly affected the coherence of the summary. In Chapter 6, we report on features that involved dependencies among code units, such as dependencies among a variable declaration and its usage.

Chapter 5

Understanding Code Fragment Summarization

To inform the development in source code summarization technology, we conducted a study to see how humans shorten code fragments. The goal of the study was to learn code summarization practices and their justification from human participants. We had two research questions:

1. **Selection:** Which parts of the code from an original code fragment should be selected for a summary, and why?
2. **Presentation:** How should the code be presented in a summary, and why?

This chapter describes the summarization study, which appears in a conference publication [\[102\]](#).

5.1 Study Set-Up

To answer the two research questions, we recruited 16 participants and asked them to shorten ten code fragments each. We instructed the participants to verbalize their thought process using the think-aloud protocol [\[47\]](#). For each

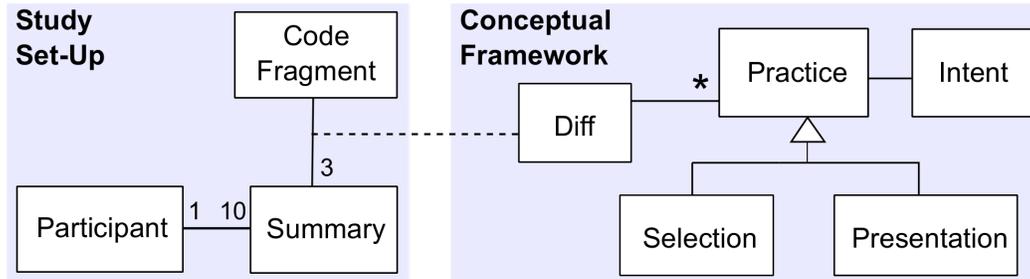


Figure 5.1: Study Set-Up and Conceptual Framework

code fragment studied, to be able to estimate differences in personal style, we asked three participants to shorten the code fragment, the result of which we call a *summary*. In total we collected 156 summaries on 52 code fragments and 26 hours of screen-recording with synchronized audio. We base the observations reported in this chapter on this data.

By analyzing this data and answering the two research questions, we learned how concrete code summarization practices lead to specific usability effects for a code fragment. This knowledge directly supports the design of tools to automatically extract and format code examples, such as Konaila (Chapter 6).

The rest of this section describes the details of the task, code fragment corpus, and participants involved in the study, also illustrated in the left part of Figure 5.1. The right part of Figure 5.1 illustrates the conceptual framework of the data analysis, which is described in Section 5.2. Sections 5.3 and 5.4 respectively present answers to the two research questions.

5.1.1 Summarization Task

We define of a code fragment summary by adapting the definition of a textual summary [72], as introduced in Chapter 3. A code fragment summary is smaller in size than the original code fragment and conveys important information in the original fragment. The major goal of a code fragment summary is to present the main ideas in the original fragment in less space.

We asked the participants to provide free-form¹ summaries. For each code fragment, a participant was instructed to write a summary of no more than three lines. To help participants envision the results, we asked them to make summaries as if they were to serve as content summaries in a result page for a search engine (for documents containing source code).²

To provide a summary of a code fragment, the participants used a data collection tool we designed for this study (Figure 5.2). The top section presents contextual information relating to the code fragment (Section 5.1.2). The middle section shows the original code fragment the participants were asked to summarize. Finally, the bottom section is a fixed-sized text box in which the participant was asked to enter the summary. In addition to the summary, for each line in the summary, we asked the participants to indicate which lines from the original code fragment the particular summarized line came from. This approach is used in experiments in the textual summarization community [67] for the purpose of evaluating how different summaries overlap. We used this information to determine the textual difference between the original and the summary described in Section 5.2.

We asked the participants to verbalize their thought process for the entire duration of their summarization activities. We recorded the verbalizations together with a video of the screen.

We chose to study the summarization practices in a lab setting with *access to the summary author*. The participants' verbalizations contained the rationale behind the decisions taken in generating the summaries. Such rationale is harder to reliably infer by the experimenters themselves in the absence of the author of a code example. We also designed the study to have *multiple* authors summarizing the same code example so that we could examine the variability among different code summary authors.

¹See the summary composition design dimension in Section 3.1.4.

²The summaries are part of the artifact companion package to the publication [102], made available at <http://annieying.ca/fse2014/fse2014-artifacts.zip>. The artifact package has been approved by the conference's Artifact Evaluation Committee.

Query: Passing Events Back to the Dialog 's Host

Android API: Dialogs

Code example:

```

1: public class MainActivity extends FragmentActivity
2:             implements NoticeDialogFragment.NoticeDialogListener{
3:     ...
4:
5:     public void showNoticeDialog() {
6:         // Create an instance of the dialog fragment and show it
7:         DialogFragment dialog = new NoticeDialogFragment();
8:         dialog.show(getSupportFragmentManager(), "NoticeDialogFragment");
9:     }
10:
11:     // The dialog fragment receives a reference to this Activity through the
12:     // Fragment.onAttach() callback, which it uses to call the following methods
13:     // defined by the NoticeDialogFragment.NoticeDialogListener interface
14:     @Override
15:     public void onDialogPositiveClick(DialogFragment dialog) {
16:         // User touched the dialog's positive button
17:         ...
18:     }
19:
20:     @Override
21:     public void onDialogNegativeClick(DialogFragment dialog) {
22:         // User touched the dialog's negative button
23:         ...
24:     }
25: }

```

1. Shortened code example:

Shortened code example:	2. Line#s of the original
...extends FragmentActivity implements NoticeDialogListener{	1
void showNoticeDialog() { // create instance and show it}	5-8
void onDialogPositiveClick(DialogFragment dialog) { //same for negative click }	14-25

Figure 5.2: Annotation tool featuring a summary by P6 (the “Shortened code example” box) and the original 25-line long code fragment. The code fragment is reproduced from Android API Guides [2] with permission under the Apache Software License, Version 2.0.

The summarization task was constrained by our decision to limit summaries to three lines. This forced the participants to make choices that were not necessarily the ones they would make if this experimental procedure had not been in place. Research in text summarization suggests that fixing the summary length in an experiment is crucial because summaries of different lengths directly affect the actual content of the summaries [35]. The artificial setup is a component of any lab study: although it decreases the ecological validity of the task, it has the major advantage that it supports the systematic analysis and comparison of the code summarization practices. We return to this issue in Section 5.2.

5.1.2 Code Fragments

Selecting code fragments to study summarization practices presents two challenges. First, the general idea of summarization is context-sensitive because of the requirement to assess the relative importance of the elements in the original code fragment. To distill a fragment to its essence, participants need a basic idea of what the fragment is about. Second, code summarization requires a non-trivial level of programming expertise: we cannot ask participants to summarize code they do not understand.

We addressed both challenges by selecting the code fragments from a well-defined corpus of programming documents: The Official Android API Guides [2]. This documentation contains a mix of natural-language text and code fragments meant to explain and demonstrate the usage of the Android API. Selecting code fragments from this documentation helps us address the first challenge above (context) by allowing us to draw from the structure of the text surrounding a code example to provide the context. It helps us address the second challenge by explicitly scoping the expertise required of participants. We discuss the implications of selecting code fragments from this corpus on the generalizability of the results in Section 5.2.

We extracted all code fragment candidates in the Android guide (1) that were enclosed in HTML `pre` tags; (2) that were non-XML, leaving only fragments with code (though not necessarily Java); (3) that had ten or more non-empty source code lines; and (4) whose closest enclosing heading started with a verb, e.g., “Passing events back to the dialog’s host.” Selecting candidates with a heading that starts with a verb is due to the necessity in providing a context for the code example (Section 5.1.3). We automatically determined these headings using a Part-of-Speech tagger.³ These four criteria produced 166 candidates. We randomly selected 52 fragments for the study.

5.1.3 Context Generation

We generated a context for each fragment using an automatic procedure based on the headings enclosing the code fragment. For each fragment, we constructed a context that consists of two parts. The immediately enclosing heading describes a specific purpose of what the code fragment is supposed to demonstrate. The rest of the enclosing headings point to the general area of the Android API the code fragment demonstrates.⁴

Figure 5.2 demonstrates a code fragment taken from a page with two levels of subsections: First, the page is titled “Dialogs” and the second-level heading is titled “Passing Events Back to the Dialog’s Host.” In this example, the title of the page, “Dialogs”, shows which part of the Android API the code fragment is taken from, as displayed in the annotation tool in Figure 5.2. The second-level heading provides a more specific purpose, which we marked as “Query” in the annotation tool. For the code examples with three levels headings, the third-level heading is displayed as “Query”, and the first two levels are displayed under “Android API”. This automatic procedure eliminates the threat of investigator bias in crafting the context, at the cost of a potential loss of precision in how well the context describes the fragment.

³<http://nlp.stanford.edu/software/tagger.shtml>

⁴These headings are part of the artifact we made available at <http://annieying.ca/fse2014/fse2014-artifacts.zip>.

Table 5.1: Participants’ Development Experience

Java \ Android	looked at Android API	developed an app	professional
1 year	P3,6,7,8	P4,5,10	
between 1 & 5 yrs	P9,14,15	P1	P11
between 5 &10 yrs		P2	P16
more than 10 yrs			P12,13

5.1.4 Participants

We assigned the 52 fragments to the 16 participants (P1 to P16) in a way that ensured that all fragments were summarized by exactly three participants. Twelve participants were assigned ten fragments and four were assigned nine fragments.

We required participants to have one year or more of Java programming experience, and have at least looked at the Android API. Of the 16 participants in the study, five were recruited from local professional programmer meet-up groups, one through personal contacts, and the remaining ten from the McGill School of Computer Science (nine graduate students and one undergraduate). Table 5.1 presents the participants’ Java and Android development experience. In total, seven had professional software development experience.

5.2 Conceptual Framework

The study produced two different types of data: shortened source code and the verbalizations of participants. We analyzed this data using a combination of quantitative and qualitative [81] methods. The basis for the analysis was the systematic extraction of the textual differences between code fragments and the corresponding summaries (“Diff” in Figure 5.1). We then refined the difference into a structured list of *summarization practices*. For this purpose we followed *coding* (or classification) techniques [81, Section 2.3], guided by the categories of operations in textual summarization [36]. We distinguished practices concerning the type of content *selected* and the way the content

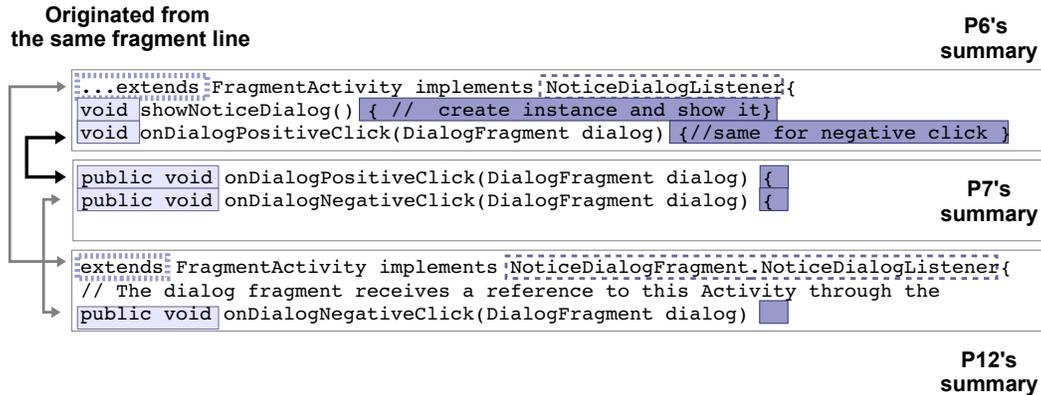


Figure 5.3: Summaries on the same fragment, with variations on the presentation highlighted

was *presented* in a summary (“Selection” and “Presentation” in Figure 5.1). The categorization enabled a quantitative assessment of the frequency and generality of each practice. To make hypotheses justifying the use of different summarization practices, we relied on a quantitative analysis of the distribution of each summarization practice across code fragments and participants. This aspect of the analysis was directly enabled by the choice to have each code fragment summarized by multiple participants. Finally, we inspected the transcripts of the participants’ verbalizations for evidence of the *intent* behind each practice.

We distinguished between selection and presentation because even summaries with content associated with the same part of the original fragment could have variations on how the summary content was presented. For example, P6’s and P7’s respective summaries (Figure 5.3) of the same fragment both included the signature of the method `onDialogPositiveClick` (marked by the arrow in bold), but P6 chose to leave out the keyword `public` and added a comment in the body (the dark shade in the third line), and P7 chose to provide the complete first line of the method declaration (the dark shade in the first line). P12, whose summary contained the signature of a different method, chose to provide the first line of the method declaration without the body (the third line). The decision of treating the removal of tokens (e.g., `public`)

as presentation, rather than selection (i.e., de-selecting `public`) is dictated by granularity. The granularity we chose for selection is at a higher level in the abstract syntax tree than individual Java tokens. A granularity at the token level would result in more decisions, complicating the conceptual framework of the analysis and the computational complexity of a summarization algorithm.⁵ This separation of content selection from presentation is typical in a natural language generation system, where the selection granularity is typically at the sentence level rather at the word level [76].

5.2.1 Links to the Evidence

This study reports primarily on evidence of a qualitative nature. A major challenge for reporting observations derived from qualitative data is linking to the evidence upon which an observation is based. In our case this amounts to providing, for each observation, the number of fragments where a practice is observed and the distribution of these fragments across participants. This amount of precision can quickly overwhelm the text to the point of unreadability. Instead, we use a new visual approach inspired by the idea of sparklines [93]. A sparkline is a small graphic embedded in the text, drawn without axes. In our context, a histogram presents the distribution of observations of a given practice for a participant (each bar) over the ten code fragments (the vertical axis). The 16 bars corresponding to participants are sorted in decreasing number of code fragments where the evidence was observed. The vertical axis represents the number of code fragments where the evidence was present. For example, the histogram for the practice *shortening identifiers* is , signifying that eight participants showed evidence of the practice in 8, 6, 5, 5, 4, 4, 3 and 3 code fragments respectively, and the rest of the participants did not use this practice. We can compare different practices in terms of the amount of evidence that was observed across the participants and code fragments. For example, the practice *shortening identifiers* was observed in more participants

⁵Section 3.1.3 contains additional discussion on granularity.

and more summaries than the practice *shortening API names* (—). This observation can be deduced by comparing the dark area of the two histograms.

The in-lined histograms are intended to provide a convenient and compact assessment of the amount of evidence for a practice. We provide more detailed links to the evidence in Table 5.2, which relates the histograms to the specific participants associated with the practices, as well as the number of code fragments observed and the number of occurrences in the summaries. Finally, each quote explaining the rationale for a practice is annotated with the corresponding participant identifier, whose characteristics can be found in Table 5.1.

5.2.2 Threats to Validity

The threats to validity for this study concern the reliability of the observations for the purpose of informing source code summarization technology. We consider the risk that a reader wishing to rely on these observations could be misled.

The corpus of code fragments is limited to 52 fragments in one technology. It is not representative of any defined population of code fragments besides the Android documentation. However, the contributions we provide in this chapter do not involve generalization from a sample to a population. We make no claim about how often the practices we noted are used *in general*, and we do not think such a projection would be particularly useful. Instead, the implications of our results concern the goodness of fit of a certain practice to achieve a particular selection or presentation goal, which is independent from frequency counts. We indicate frequency counts for each practice to be transparent about the strength of the evidence for the observations, without implying that they can be extrapolated.

One threat of using frequency counts as a measure of the strength of the evidence is that not all practices are equally likely to be observed in the 52 fragments. It is possible that our data misses some useful summarization

practices, for example if they target special source code patterns that were not part of our code fragment corpus.

Our use of a grounded approach means that the data is collected directly from participants and, as such, is influenced by them. The corresponding threat is that a participant with an unusual background or behaving strangely could corrupt the data. Our experimental protocol required participants to justify most of their decisions, allowing us to discover such potential problems. We observed that all participants appeared to complete the task in earnest. To avoid injecting our own bias, we did not attempt to judge the quality of the summaries.

As mentioned in Section 5.1.1, the summarization practices we observed were employed in a context where participants were required to produce a short (three-line) summary. This decision was necessary to obtain comparable data. At the same time, it also means that different practices might be useful in contexts where the desired output summary is not constrained by size.

5.3 Selection Practices

Selection practices concern how participants decided on which content to include in a summary, e.g., whether to include a specific method declaration that matched the query terms or whether to exclude exception handling code. We observed three types of selection practices, each using a distinct type of information: language constructs from the code example itself, query terms, and human considerations, such as the programming expertise of the reader.

Understanding these practices can help determine what type of content should automatically be selected (or filtered out) when presenting code examples in contexts where summarization is appropriate (e.g., in search results).

5.3 Selection Practices

	# of Selected	# of Eligible for Selection	
Compilation Unit	0 /	3	
Package Declaration	0 /	21	
Import Declaration	0 /	21	
Class Declaration	50 /	54	
Class Signature	12 /	50	
Class Body	50 /	50	
Method Declaration	184 /	210	
Method Signature	124 /	184	
Method Body	126 /	184	
Conditional Structure	65 /	165	
Try Statement	9 /	12	
Try Block	9 /	9	
Catch Block	0 /	9	
Finally Block	0 /	3	
Comment	33 /	702	

Figure 5.4: How often a construct was in a summary

5.3.1 Practices Related to Language Constructs

Certain types of language constructs were consistently included (e.g., content of a class) or excluded (e.g., exception handling code) in a summary. Figure 5.4 shows how frequently a language construct appeared in a summary (“# of Selected”). To put these frequency numbers in context, we also provide how frequently the construct was eligible for selection (“# of Eligible for Selection”). This second set of frequency numbers represents either the occurrences of the construct in a code fragment shown to a participant (e.g., `try` exception handling blocks do not occur as often as method declarations), or the number of times the parent node was selected in a summary (e.g., a method signature can only be selected when a method declaration is selected). Finally, the pie charts show the ratio “# of Selected” divided by “# of Eligible for Selection”.

The first two practices involve methods. All participants selected methods (██████████), as P14 justified, “First, I want to know the functions I have to use.”:

Practice - Including (or Excluding) the Method Signature: Depending of the code fragment a method signature can be included or excluded.

Including the method signature (██████████) was considered as part of keeping the structure of the code. As one participant put it, “because there’s a lot of them [code], it can be anything. It’s the structure. The main part is the class `BillingReceiver` which extends `BroadcastReceiver`, the method that overrides inside. The rest can be ignored.”^{p9} Another participant chose to show the structure of the code rather than the control flow structure: “The `switch` is more about how the method functions. What are the possible functions and outcome. [...] I was just given `switch`, I have no idea of what it is.”^{p7} This structure can be important in code on the Android platform with a substantial amount of call-backs. There were fewer cases in which only the method signature was kept (██████████), while more fragments had both the signature and the body selected for the summary (██████████). One reason for keeping the method body was that the fragment has more computation-intensive code. For example, a fragment about the usage of the gyroscope had more than half of the lines on mathematical computations. For

that fragment, all three participants included at least two statements from the method body.

Participants collapsed the method by displaying the content of the method *without* the signature (). One participant who eliminated the method signature said that the declaration was a common API call-back, saying, “This handler is pretty much for any activity.”_{P4} Another reason was that the participants expected the user of a summary could find the signature through the IDE.

Practice - Including Overriding Methods: Of the method declarations with an explicit `@Override` annotation (43 methods), most of the methods (36) were included in a summary by at least one participant. One participant even called it a “regular pattern”_{P13} to include overriding methods. The seven methods not included by anyone were in code fragments with other choices of methods. However, the override annotation itself was rarely kept, only in seven code fragments ().

Annotating a method declaration with `@Override` provides a Java compiler a way to catch spelling mistakes, when a developer intends to override a method from a super-class but the method does not exist. The annotation is also meant to make it obvious when a method declaration overrides a method in the super-class. While no participants cited the former intention of the `@Override` annotation, a few participants said that the annotation could remind the reader of the summary that the method declaration was overriding a method in the super-class: “What’s I’m trying to tell the user is that he has to override the `onCreateDialog` method.”_{P3} On the other hand, user-defined methods not involving in an event call-back do not tend to be included in the summary, as P9 put it: “The `main` [and other methods containing application code] can change.”

Practice - Excluding Exception Handling Blocks: None of the exception handling code, enclosed in `catch` or `finally` blocks, appeared in a summary. There were several intents behind the practice of excluding exception handling code. First, exception handling code was not unique to an example (“Try-catch is part of almost all standard code.”_{P5}) or too obvious to the reader (“Anyone working with sockets knows it will throw exceptions. I will remove the catches, and the try.”_{P2})

Second, the code inside the `try` block was kept (while the `catch` clause was removed) to show one case of the code: “The first thing you should do in an example is [to] assume everything is OK.”_{p2} Third, participants expected that missing exception handling code would be suggested by an IDE: “[Missing] this exception you would have Eclipse complaining about it.”_{p11} Another participant also indicated: “The first one [to remove is] the try-catch. The IDE, or the compiler itself will fix it, because you cannot compile it. [...] If he copies my [summary], he needs to fix the issue.”_{p2}

Practice - Keeping Only One Case in a Parallel Structure: Some code fragments contained code with multiple cases. In the case of `if` or `switch` statements, more than one third of the instances only had one block selected for a summary. Keeping one case and dropping the others also happened with method calls (“I can just remove one of the buttons. Instead of having the cancel button, I can just have the OK button.”_{p4}) or method declarations (“`onStop` is basically the reverse of `onResume`. It will be OK to just display everything in `onResume` [...]”_{p9})

5.3.2 Practices Based on Query Terms

Not surprisingly, participants used terms from the query to determine whether a part of the code was relevant enough to include in a summary. Thirteen out of 16 participants explicitly mentioned the importance of the query in the decision of content selection. For example, “`startForeground` [a method declaration] actually starts the foreground service. Since the query [“Running as a foreground service”] doesn’t have anything to do with the media player, even though it’s part of the API being used, [...] I left it out.”_{p15}

All 16 participants verbally justified that they selected code elements because identifiers in the elements matched terms in the query. Of the 70 distinct methods from the 52 code fragments ($70 \times 3 = 210$ instances of methods eligible for selection from Figure 5.4), twenty-two contained at least one query term (or stemmed). Of these 22, only one was not selected by any

participant. Method signatures that did not appear in a summary were often irrelevant to the query: “This [method declaration not selected] probably does not have to do with the intent [the query].”_{P4} Both from the verbalizations and from the summaries, we conclude that all participants used the query in content selection for summarization.

5.3.3 Practices Considering the Human Reader

Summaries are targeted to humans. Participants explicitly considered the expertise of the programmer.

Practice - Including Easy-to-Miss Code: Four participants mentioned including *easy-to-miss* parts of the code in the summary, e.g., the method declaration `onResume` “is something people tend to forget.”_{P11} Another participant made a similar comment and explicitly qualified the advice with the participant’s own personal experience: “It reflects my own knowledge of this class [...]. If you set the layout in the wrong place, you can end up with a lot of problems. I want to be specific there.”_{P10} Another participant expressed frustration at not being able to include a call to the super class which was deemed easy to miss: “I’m still not happy to remove the `super`. If someone looks at the short one, copy and start. He could miss it.”_{P2}

Practice - Accounting for Programming Expertise: Seven participants justified not including parts of the code that were too obvious to the reader. The code might be obvious because of (1) previous languages used (“This is C-style where you handle one byte at a time. It would be pretty obvious this is how you do it”_{P11}) (2) the assumption on the knowledge of Java (“The sockets [...] is not specific to Android. It’s exactly the same in standard Java”_{P2}) , or previous knowledge of the Android API (“`onCreate` is a method where if he knows a little bit of Android development, he won’t get a lot from this [`onCreate`] anyways.”_{P5}) In general terms, one participant explicitly distinguished the different needs of an expert and a novice of an API: “Someone who’s very experienced [...] may be

looking for something very specific, [such as] methods [...]. Someone who is a complete novice would probably look at something very explanatory.”_{P3}

Practice - Using the Query to Infer Expertise: Participants used the query to infer the level of expertise on the API of the query poser, and then excluded the part of the API deemed obvious. One participant commented on the decision of not including certain method declarations for a query about Near Field Communication (NFC), a topic the participant deemed advanced. “If someone is doing NFC, [...] someone already knows what `onPause` [or] `onResume` is, so I don’t need to stress it. This is more advanced stuff than how the activity behaves.”_{P11} The same participant continued on commenting on the abstraction level inferred from the query: “Especially, when you query something very specific, in this case like NFC, there isn’t really other stuff to care about.” Interestingly, P11 was one of the participants who deemed `onResume` easy to miss in another fragment.

5.4 Presentation Practices

Presentation practices relate to decisions about *how* the selected content appeared in a summary. We observed participants made changes to the selected original content to make it fit into the space allowed for the summary through practices in three general categories: trimming a line when needed (Section 5.4.1), compressing a large amount of code (5.4.2), and truncating code (5.4.3). Beyond presentation decisions for the purpose of fitting the desired content into the space, we found that formatting decisions were personal and related to readability of the summary (5.4.4). Despite the task’s focus on reducing code, we observed participants improved the code, e.g., by clarifying comments (5.4.5).

The presentation practices we collected provide practical insights into how source code fragments can be formatted in various situations including on search results pages, in forum posts, and in tutorial documents.

5.4 Presentation Practices

Table 5.2: Evidence of the presentation practices

Presentation practices	#Participants (out of 16)	#Fragments (out of 52)	# Instances
Trimming a Line When Needed	10	P2,4,6,8,10,11,13,14,15,16	33 95
Shortening Identifiers	8	P2,4,6,8,10,11,15,16	29 72
Shortening API Names	4	P6,10,11,15	5 7
Eliding Type Information	10	P2,4,6,8,10,11,13,14,15,16	9 16
Compressing a Large Amount of Code	13	P2,3,4,5,6,7,8,9,10,11,14,15,16	28 46
Shortening Multiple Statements	10	P3,4,6,7,8,9,10,11,15,16	15 51
Shortening Method Declarations	7	P4,6,8,10,11,14,16	10 11
Shortening Control Structures	8	P2,3,5,6,8,10,11,16	12 14
Truncating Code	12	P1,2,4,5,6,8,9,10,12,13,15,16	28 63
Eliminating a Parameter	9	P1,4,5,6,8,9,10,15,16	16 28
Truncating a Signature	9	P2,4,5,8,10,12,13,16	18 35
Formatting for Readability	16	all	52 140
Indenting	8	P1,2,4,7,8,9,13,14	20 27
Treating Lines as Separate	15	all except P10	52 135
Improving Code	9	P1,2,4,6,8,10,11,12,16	9 35
Fowler's Refactorings	4	P1,2,8,11	4 5
Generalization	4	P2,4,8,16	5 8
Clarification	6	P6,8,10,11,12,16	14 22

**#Summaries
(out of 156)**

5.4.1 Trimming a Line When Needed

Ten participants (P1-P10) performed transformations for the purpose of trimming a line, such as shortening variable names or removing a type qualifier. These operations happened when the content needed to fit into a line pre-allocated for that content.

Practice - Shortening Identifiers: We expected participants to shorten variable and parameter names, because these changes do not change the semantics of the program. Eight participants (P1-P8) did so in 29 (56%) code fragments. We observed four ways to shorten a name: (1) using acronyms, e.g., from `sharedPreferences` to `sp`, (2) shortening words in an identifier, e.g., from `defaultValue` to `defaultVal`, (3) using discourse aggregation for reducing the complexity [75] by dropping words (e.g., from `defaultValue` to `default`) or paraphrasing (e.g., from `tagFromIntent` to `intenttag`), and (4) using a combination of these operations, e.g., `mInputStream` to `in`. These observations concur with Eshkevari et al.’s taxonomy on identifier renaming in a code base [24].

Practice - Shortening API Names: We expected that the names of API calls and overridden methods would remain the same in a summary. As P6 asserted, “I am assuming overridden methods cannot have their names changed.” Surprisingly, we observed changes to these API elements, by four participants (P1-P4). P6’s justification on the shortening of the name of an API method was that the name is “abnormally long”_{P6}: “`unregisterOnSharedPreferenceChangeListener`, what kind of name is that?”_{P6} P6 renamed the method to “`unregister...`”. Note that the context was important in this case, as the API call is expected to be made inside a class that inherited from `SharedPreferencesChangeListener`, which defined the `unregister` method.

Practice - Eliding Type Information: Java requires a variable to have a declared type in an unambiguous name-space and explicit down casts. In the context of summarization, ten participants (P1-P10) relaxed this requirement and elided type information in the following situations: (1) Four participants eliminated a type qualifier; e.g., “I’m removing the name-space. [...] Someone

can [put a] `import static.`"_{P2} In addition, the type information is a piece of information expected to be found easily: "[For] the flag, if they are in the definition of the type, they can see which flags are in the type."_{P12} (2) Five participants removed the variable type in assignments in six assignments that were selected over five summaries. (3) In the only selected line that contained a type cast in the whole corpus, all three participants selected that line and removed the type cast in consensus. (4) Two participants shortened a type reference or a primitive type: e.g., from `Object` to `obj`.

5.4.2 Compressing a Large Amount of Code

Twelve participants () employed more complex abstraction and aggregation practices that greatly reduced the code from its original size. These changes involved compressing a block of code that contained one or more method declarations, control statements, or multiple statements and replacing the code with ellipses or a comment. Four participants (P5,7,9,15) only employed ellipses when compressing a large block of code, four (P4,11,14,16) only employed comments, and four (P3,6,8,10) employed both.

It is inevitable that when a block of code deemed important exceeded the space available for summaries, the participant needed to somehow compress the code. We observed that participants either compressed the code using ellipses ("...' [indicates] additional important things"_{P8}) or comment. Ellipses and comments also could "abstract a particular block."_{P8} Certainly, comments conveyed more information than ellipses. However, choosing comments or ellipses was affected by the trade-off between information and space: All the comments in the summaries were longer than three characters.

Practice - Shortening Multiple Statements: Ten participants () shortened multiple statements including the whole method body. The use of comments versus ellipses was split almost evenly: Six of the participants (P3,4,6,8,10,11) used comments 22 times and seven (P3,6,7,8,9,10, 15) used ellipses 29 times. P15 who only used ellipses to summarize multiple statements

5.4 Presentation Practices

```
1 /* implement SensorEventListener @override onAccuracyChanged(), ↔
   onSensorchanged() */
2 @override onCreate(), onAccuracyChanged(), onResume(), onPause() ↔
   onSensorChanged(){...}
3 // remember to override all inherited methods appropriately
```

Figure 5.5: All three summaries on the same example contained a comment listing overriding methods

```
1 while(cur.moveToNext()) {...}
2 if (checked)... else...
3 if (resultCode == Activity.RESULT_OK && requestCode == ↔
    PICK_CONTACT_REQUEST) { //code for activity }
```

Figure 5.6: Sample of summarized control structures

said, “Most of the time I put ‘.’ when there are lines in between. If you don’t put that in, it’s less clear there’s other stuff in there.”_{P15}

Practice - Shortening Method Declarations: Seven participants ([P1](#)) aggregated whole method declarations by replacing the whole declaration with comments or with ellipses. Unlike in abstracting multiple statements, most participants (six out of the seven) used comments rather than ellipses (one out of seven) to abstract method declarations. The ten comments demonstrated three different ways to abstract content: (1) listing the method declarations (eight comments), e.g., lines 1 and 2 in Figure 5.5; (2) aggregating lexically [75] through the use of the quantifier “all” (one comment), as in, e.g., “all inherited methods” in line 3 in Figure 5.5; and (3) aggregating semantically [75] (one comment), e.g., the comment `//same for negative click` which referred to the code for handling the positive click. Lexical aggregation is a way to summarize a list of elements with a few words rather than explicitly listing the methods [75].

Practice - Shortening Control Structures: Eight participants ([P1](#)) shortened control structures. Four participants replaced a block in a conditional statement or a `switch` statement, or in a `while` or `for` loop, with a comment or ellipses. Figure 5.6 illustrates three such examples. Beside using ellipses and comments, five participants (P2,6,8,11,16) compressed the whole structure through program semantics preserving transformations. Participants

either turned an `if` statement into a more compact conditional expression (with operators `?` and `:`), or turned a `switch` statement into an `if`: “`switch` is going away because [...] they become too big. I’m just going to put an `if`.”_{P2}

In brief, ellipses and comments were approaches to shorten a large of piece of code. This result concurred with one of Nasehi et al.’s findings [65] on concise code examples that contain “place-holders, such as comments or ellipses, which usually transforms the code to a solution skeleton.” We found that to summarize method declarations, almost all participants employed comments instead of ellipses. The majority of the comments were simply listing the name of the method declarations or using lexical aggregation.

5.4.3 Truncating Code

Code truncation transformations involve shortening a line while violating syntax. Twelve participants () performed such truncation. These code truncation transformations affected code compilability, which some participants considered important. One factor that affected the presentation of code was a participant’s view on the importance of making the code compilable. This view varied between the participants. Those who were more indifferent to the importance of compilability tended to perform code truncating operations violating syntax, using ellipses, or cutting off parts of a statement or having unmatched brackets or parentheses.

Four participants mentioned the importance of compilable summaries. The most common reason was that participants wanted to copy and paste the code directly: “It’s very important for the code to compile correctly. [...] I’m a very lazy person. I would Google [...] the snippet, [...] copy it, and pretend it’s mine.”_{P6} Another reason was for understandability: “Having something compilable allows me to actually see the effects, and that will help me to understand the code better.”_{P8} Compilable code was also important because otherwise the summary could look “sloppy.”_{P2}

```
/*convert ns to s */ omega = sqrt(X*X + Y*Y + Z*Z); if (omegaMagnitude > EPSILON)
{ X /= omega; ... } theta = omega * deltaT / 2.0f; deltaR[0] = sin(theta) * X...
deltaR[3] = cos(theta); SensorManager.getRotationMatrix(new float[9], deltaR);
```

Figure 5.7: A summary without formatting, by P10

It was not always possible to make the code compile. When the code was not compilable, participants wanted to minimize the non-compilable parts: “I want to copy the least amount of code or through the least number of places. Copy code with the least number of changes [that] would [make the code] work.”_{P6} One participant wanted to clearly mark the non-compilable parts of the code: “It’s important to either compile on its own, or if it does not compile, it is readily identifiable what needs to be done to make it compile.”_{P8} P8 invented a language construct, a pair of angle brackets to indicate variables not declared in the summary: “Here I am going to add uncompileable code [replacing the variable name with <NAME>].”_{P8}

On the other hand, one participant did not see the importance of having compilable or runnable code: “For such a short and abstract example, [...] we are not talking about runnable code.”_{P2} Less so, P4 said, “If it’s not compilable, Eclipse or whatever editor you use will give some hints. This expects a pointer, or this expects an object of this class, inherit this class, the kind of auto-fix suggestions that Eclipse give.”_{P4}

Practice - Eliminating a Parameter: It was sometimes desirable to eliminate a parameter which is deemed to be a detail: “When we search for something, we don’t want too much stuff that is irrelevant, [for example,] the parameters.”_{P9} Nine participants () shortened the parameter list in a method call or a method declaration. When eliminating a parameter, participants chose to replace a parameter with ellipses (eight participants over thirteen code fragments) as well as simply eliminating a parameter (three participants over four code fragments). P1 justified the use of ellipses: “Here [where the parameters were eliminated] you have to put ‘...’ because there are multiple parameters.”

Practice - Truncating a Signature: Because many of the method and class declarations are part of the call-back mechanism of the Android

framework, we expected that when such a method or class signature was selected for a summary, the signature would be kept intact. For both method declarations and class declarations, leaving the signature intact was indeed the most common way for a signature appeared in a summary. However, a significant number of participants () had summaries with the method or class signature modified. These changes involved Java keywords (such as `public` or `static`), identifier names, or the whole signature replaced by a comment. One participant justified the removal of keywords, saying, “`public class something, extend something [...]. This is rudimentary. [...]. All this stuff is meaningless.`”^{p6} This justification corroborates with conclusions from work on statistical modeling of source code [30], pointing out that source code contains redundancy.

5.4.4 Formatting Code for Readability

Participants explicitly expressed the importance of two different readability dimensions that related to formatting: indenting and treating lines as separate. Figure 5.7 illustrates a summary with little formatting, i.e., without any indentation, and with no separate lines. In Table 5.2, for formatting practices we report the number of summaries exhibiting the practice instead of the number of instances because formatting practices apply to the whole summary, not necessarily to a specific line.

Practice - Indenting Code: Indentation in code can increase readability: “It’s easier to see the layers, the level of importance. You look at [the code] from top to bottom.”^{p9} Two participants mentioned that indentation is a standard coding convention and is required by languages such as Python. For example, “interpreted languages requires indentation, [...] Python [and] CoffeeScript.”^{p2} Eight participants () intentionally indented at least one summary. We did not count cases when the indentation in the summary was not intentionally put in, e.g., indentation that were simply copied and pasted from the code that contained indentations.

Practice - Keeping Lines as Separate: Keeping the summary as separate lines can be seen as desirable, whereas wrapping lines and putting two lines into one can be seen as undesirable. P11 declared, “it’s ugly,” when a comment expected to fit a line wrapped around to the following line. P2 considered eliminating a line break between the class signature and the method signature as undesirable: “The class definition and the method on the same line. That will be really crazy.” All participants (██████████) treated at least one summary with all separate lines, i.e., not wrapping lines and not putting two lines into one in a summary.

Participants’ views on readability was divided. Half of the participants (P2,4,8,9,11,12,13,15) explicitly expressed importance in readability. P2 despised original code with poor readability: “I look at this and I’m scared. Oh my god, what’s happening here? There’s not a break line.” The other half of the participants included P3 who did not think readability is important because summaries are short: “Since it’s just three lines of code, [...] I don’t think he [the reader] would mind the formatting.”_{P3} P10 thought packing more information is more important than readability: “If it [the summary] is more readable and as a consequence there is less information, you still would not know whether you want to click [the link to the whole example.” Despite the eight participants who did not think readability was important, the two formatting practices were used by all participants (██████████).

5.4.5 Improving Code

We observed three types of transformations that improve the code: refactoring, generalization, and clarification. Nine participants (██████████) took the effort to improve the code: “Can I, interesting, well I guess I can. I should. From my experience, I will just do this [refactoring].”_{P2} Because the main objective was to shorten the code, we found any improvements surprising, especially when some improvements, such as adding in clarifications as comments, lengthened the code.

```
1 if (item.isChecked()) item.setChecked(false); else item.setChecked(true);  
2 item.setChecked(!item.isChecked());
```

Figure 5.8: P4 refactored code from line 1 to line 2

Practice - Fowler’s Refactorings: Four participants () applied two different types of refactoring to control flow structures [26]. P2 and P11 applied refactorings in the spirit of the “Consolidate Conditional Fragments” refactoring on the same code fragment. P2 eliminated unnecessary control flow branches, turning line 1 in Figure 5.8 into line 2. P1 and P8, on two code fragments, applied the “Consolidate Duplicate Conditional Fragments” refactoring, moving part of code that is in all branches of a conditional expression to the outside of the expression.

Practice - Generalization: Four participants generalized a value or variable specific to the examples to something more likely applicable to other contexts (). P2 and P4 generalized a constant specific to the examples to a variable, e.g., from the constant `Intent.CATEGORY_ALTERNATIVE` to the variable `myCategory`. P8 and P16 replaced a variable specific to the example with a place-holder. The place-holder employed by P8 was an invented notation, an angle bracket (e.g., replacing the variable `R.id.menu_search` with `<NAME>`) or with a comment for P16, (e.g., replacing the string constant `"landscape"` with `/*orientation*/`). P8 and P16 essentially treated the summary as a closure and noted the free variables with the place-holder and comment. The terminology of this type of generalization is called conceptual aggregation in the natural language generation domain [75].

Eight participants explicitly mentioned that some parts of the code were important for the example to work, but too specific to the example. One participant shortened a path because “someone [the query] assumes the data is an image, but it doesn’t need to be an image.”_{P2}

Practice - Clarification: Six participants () added to the summary clarifications that were not present in the original code. Five clarified names of the variables; e.g., P16 justified replacing the variable `cur` with a

more descriptive name, `queryResult`, especially important for a variable that is the input or the result of a piece of code: “Intermediate variables don’t matter, but what needs to be fed in and what needs to come out, those two variables [`cur` and `cr`], [matter.]”_{P16} We observed 16 comments in nine different code examples. Fifteen of those comments reiterated what the code summary presented, while one comment (`//Start and stop download when activity is in foreground`) clarified that the call-backs `onResume` and `onStop` were run in the foreground, an insight not explicit in the original code. Some of these clarification transformations are found in automatic algorithms to expand and improve identifiers [45].

5.5 Discussion

Most of the work on code example generation and summarization has focused on the content selection aspect. Language constructs have been used for code fragment summarization (Chapter 4) and code example synthesis [38]. Systems for extracting [5], synthesizing [11, 38], or summarizing (Chapter 4) code examples have made heavy use of query terms. These practices of using language constructs and query terms were used by the participants, a result concurring with existing work. Section 5.5.1 discusses some novel types of information we observed beyond the existing use of code itself and query terms. Sections 5.5.3 to 5.5.4 focus on implications related to the presentation practices. Finally, Section 5.5.5 discusses implications on programming environment design.

5.5.1 Selection Beyond Code and Query Terms

Accounting for expertise information to determine which content should be included can be a promising type of information to complement existing code example search engines that are based heavily on the code itself and the query as the input to the analyses. Existing measures to quantify expertise include the use of commit logs and interaction history [60, 27]. These measures all share

the assumption that the more a developer changes the code or calls a method, the more expertise of the corresponding code or API method the developer has. We observed that participants either assumed the reader had a certain expertise or inferred expertise from the query (Section 5.3.3). In information retrieval, the foundational research on inferring intention is whether a query is informational or navigational [46].

5.5.2 Most Summaries are Abstractive

Current textual summarizers generate two types of summaries: *Extractive* summaries have the content obtained solely from copying and pasting whole sentences from the original document, whereas *abstractive* summaries can contain text modified from the original document [53].

If all participants were to provide extractive summaries, we would only observe selection practices and formatting practices (modifications involving white spaces) in the summaries. However, all 16 participants () employed modifications beyond changing white spaces, namely, modification involving trimming a line (), compressing a large amount of code (), and truncating code (). As we saw in Section 5.4, the participants made changes to the selected content to make it fit into the space allowed for the summary.

Modifications associated with abstractive summaries were present in 90% (47 out of 52) of the the code fragments; thus, these 90% of the code fragments had at least one abstractive summary provided by a participant.

5.5.3 Abstractive Summary Generation

The code-shortening transformations found in the field of code transformation and example generation and extraction typically generate syntactically correct code and preserve program semantics. For example, shortening identifiers using acronyms is used in Buse and Weimer’s code example synthesizer [11]. Knowing which words to shorten in an identifier or using discourse aggregation

require a deeper understanding of the linguistic aspects of the identifier, such as part-of-speech information. We observed the shortening of API names only in exceptional cases, when the context was clear and when the name was long.

Two presentation practices involving compressing a large amount of code (Section 5.4.2), *shortening multiple statements* and *shortening method declarations*, did not necessarily result in source code. In the *shortening method declarations* practice, we observed summaries with both code and natural language. Overall, seven participants (██████████) injected additional natural language (in the form of comments or place-holders described in Section 5.4) into the code summaries. This motivates a novel type of transformations that mix code and text. The only work we know of in this area is the natural summaries generated by Rastkar et al. [73]. Their summaries describe a commit as part of a software concern. The patterns found in their summaries include listing the method declarations changed in a commit and using lexical aggregation to describe a commit (e.g., “All of the methods involved in implementing ‘Undo’ are named `undo`”). We have observed both patterns (listing and lexical aggregation), as shown in Figure 5.5.

The presence of improvement transformations was surprising. However, the proportion of these transformations were small. The five instances of refactoring of control flow structure were out of 65 on conditional structures selected, and the ten instances on generalization and clarification on values and variables were a small fraction of the total number of statements containing a variable. The 16 comments were only inserted by two participants, and most of the comments were redundant because they reiterated the code. Also, generalization and clarifications are challenging to generate even in a natural language generation system. Extraneous to the main goal of a summarizer, improvement transformations should be of lower priority for a summarizer to address.

5.5.4 Silhouette of a Summary is Important

Formatting practices determine how much space a summarizer has for a summary. We observed that all participants employed some formatting in their summaries. The formatting included respecting indentation and keeping lines as separate lines. These results uniquely apply to the problem of summarizing and presenting source code, as opposed to text. Text summarizers typically use the space as a contiguous stream of characters with no indentation.

The amount of space in the summary could affect readability. “I don’t like packing more stuff. I always want readability. That helps me, in one glance, to assess whether the particular code is helpful or not. [...] If the code is packed, it’s pretty hard. It would go for another example which has more clarity.”^{P4} We did not derive any specific practice from this behavior due to the difficulty of objectively defining what “packing more stuff” means. We nevertheless observed that there did not appear to be a strong correlation between the length of a code fragment and the length of its summary (Pearson $R = 0.0758$, $p = 0.347$).⁶ This result indicates that many other factors could influence the density of a summary.

In addition to the formatting practices we have presented, P13 explicitly mentioned the preference of having short lines so that the code was formatted in a narrow, vertically long style: “My style is longer vertically than other people. When I see other people putting wide lines, it bothers me because important bits of code can get lost of the right side of the line. I spent a lot of my career taking other people’s code, becoming the maintenance guy, as I have to learn the code base. I’d like

⁶The data points we used for the Pearson correlation calculation were the length of the 156 summaries and the length of their corresponding code fragment. One concern with this calculation was that the data points (summary lengths and the corresponding code fragment lengths) were not independent because there were three summaries per code fragment. To see whether this situation affected the result, we took three random samples from the data points, sampling without replacement, so that each sample contained only one summary per code fragment (i.e., Select randomly one summary for each fragment, calculate the Pearson correlation coefficient on this set, select one summary from the remaining two, calculate the correlation on the second set, and then calculate the correlation with the final set.) The Pearson correlation coefficients on the three samples were $R = 0.0361$ ($p = 0.799$), $R = 0.127$ ($p = 0.371$) and $R = 0.102$ ($p = 0.473$) all indicating that there did not appear a strong correlation between the length of a code fragment and the length of its summary.

the important stuff to be in the front of the line.” This idea suggests a different design to the search result page. Instead of having wide summaries ordered vertically in a search result page, an alternative design is to have narrower and longer summaries. This design idea contributed to the motivation of the two-dimensionally constrained summaries we introduced in Section 3.2; Chapter 6 describes Konaila which generates such summaries.

5.5.5 Programming Environment Design

We have not observed any summarization practices involving reordering code beyond comments or white space. This implies the possibility of populating a collapsible code editor initially with the summary and the rest of the code as being expandable. One early support in programming environments for displaying code in a collapsible manner is RPDE [29]. The RPDE framework “provides an composition facility [...] to construct a two-dimensional image,” containing “a set of objects defined by the user as the focus set, the objects required to establish the structural relationship among the objects in the focus set, and as much nearby material as will get on the display” [29]. Modern programming environments, such as Eclipse,⁷ support folding editors where segments of code can be collapsed or expanded. What can be collapsed and what is initially collapsed are hard-coded.

The preference for copying-and-pasting code (Section 5.4.3) indicates that a search interface should support copying and pasting compilable code. One possible user interface is to provide a code fragment summary as a search result clue, and provide a separate widget for copying and pasting the full, compilable code.

⁷<http://www.eclipse.org/articles/Article-Folding-in-Eclipse-Text-Editors/folding.html>

5.6 Chapter Summary

This study elicited selection and presentation practices we observed from 156 concise code representations obtained from 16 participants. The goal of the study was to inform the design of concise representations of source code and automatic summarization algorithms. The selection practices we observed reinforce the existing usage of code and query terms in content selection in the summarization domain. The selection practices revealed the importance of the human reader, as we observed that participants targeted summaries to the expertise level inferred from the query. Moreover, participants did not simply copy and paste parts of the the original fragment to the summary verbatim; all 16 participants employed practices to modify the content, mostly with the intent to make it more concise but also make it more compilable, readable, and understandable. The practices directly inform the design and the generation of concise source code representations. More specifically, insights from the formatting practices (Sections 5.4.4 and 5.5.4) motivated the formulation of two-dimensionally constrained summaries (Section 3.2). Selection practices presented in Section 5.3 motivated the design of the four salience filters in Konaila (Section 6.3), our second attempt at building a code fragment summarizer. In this study, we also gained understanding in the granularity human selected code content for a summary; this understanding directly informed the definition of selection units (Section 6.2.2), the granularity used in the Konaila summarizer.

Chapter 6

Optimization-Based Code Fragment Summarization

This chapter reports on the design, implementation, and evaluation of a technique targeted to the summarization problem we formulated in Chapter 3:

Given as input a code fragment and a query (a set of keywords), produce a shorter version of the fragment that fits in a two-dimensional space (L lines by W columns) and that captures as much as possible of the essential elements of the original code related to the *query*, while remaining *readable*.

The technique is based on optimization and is embodied in a tool we implemented called Konaila. Konaila first estimates the value of a code unit, and then uses optimization to maximize the value of the content selected while constraining the chosen content to be formatted within an L by W space.¹ Konaila estimates the value of a code unit to a summary using four salience filters: query relevant call filter, call-back filter, control flow filter, and variable definition-use filter. The filters take advantage of multiple aspects of the input code fragment and the query: the similarity of the words in a code unit to the query, the extent to which the syntax of the code unit is indicative of

¹Regarding the space constraints, see Section 3.1.5.

importance, and the extent to which a code unit is involved in dependencies among a variable declaration and its usage. The motivation for using query- and syntax-related features is based on the relative success in using the combination of these two types of features in our machine learning case study (Section 4.3) and the selection and presentation practices (Sections 5.3 and 5.4) from our empirical study on summarization practices. We also incorporated into the salience filters knowledge from existing research [42, 79, 82, 85, 89]. Because Konaila handles incomplete code fragments and some pseudo-code, it can generate summaries of code fragments on the web such as those from Stack Overflow.

The evaluation of Konaila involved eleven human raters² evaluating altogether a total of 364 sets of summaries generated from code fragments found on Stack Overflow. Raters also judged that Konaila’s summaries were better at capturing the original elements of the code related to the Stack Overflow question while remaining readable, compared to a competitive baseline that included code units that maximally fill the given space. Raters judged that Konaila’s summaries were better than summaries generated without using optimization, indicating that optimization is essential in the effectiveness of the summarization technique. We observed that, according to raters, 52.1% of Konaila’s summaries captured as much as possible the original elements of the code related to the Stack Overflow question while remaining readable. Even though 52.1% may seem low, it is statistically significantly better than the baseline summaries, and we have to interpret this number with two considerations: the inherent subjectivity in summarization and the large number of decisions involved in summarization.

The inherent subjectivity in the problem of selecting the most salient content has been well documented for text summarization [49, 67]. For code

²We refer to the participants in our summary evaluation tasks as *raters* rather than *annotators*, the terminology we used to refer to the four programmers who created the finer-grained corpus for the machine learning based experiments in Chapter 4. The difference in terminology is due to the difference in the goal, i.e., an annotator determines what should be in a summary while a rater determines how good a summary is.

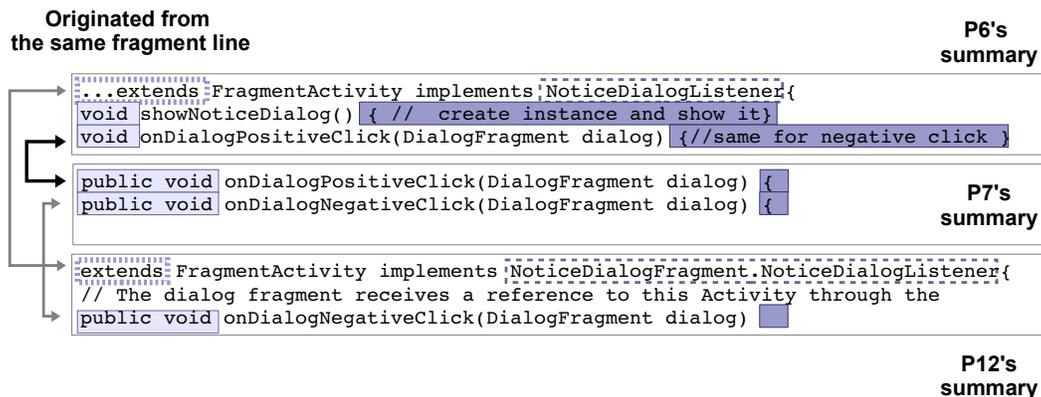


Figure 6.1: Even summaries on the same fragment have a lot of differences due to the inherent subjectivity of the summarization task.

fragment summarization, we also observed the subjectivity issue from our study of summarization practices. For example, Figure 6.1 (reproduced from Section 5.2, Figure 5.3) shows that even when three participants shortened the same code fragment, the three summaries were different, to the extent that there was not a line in the original code fragment that was selected by all three participants. The conclusion is that there is no universal “ground truth” summary. Rather, one can consider all three summaries as correct. The situation of not having a ground truth is different from other problems in mining software repositories, such as determining whether a change results in a bug [39]; one can use the bug fix history to find which methods were changed to fix a bug and use that as the ground truth. Ultimately, subjectivity leads to difficulty in obtaining reliable human judgements, and therefore researchers in text summarization used two types of evaluation procedures: the notion of correctness that is typically constructed using *multiple* raters (such as the use of gold standard summaries in Section 4.1.1), or an evaluation strategy that involves *comparing* two full summaries. Our evaluation design is based on the latter type because we wanted to elicit feedback on the summary as a whole.

Figure 6.1 also illustrates the complexity of the summarization problem. Even when two participants selected the same line (e.g., the method signature `onDialogPositiveClick` marked by the black arrow), there were a lot of variations

```
1 Display display = new Display();
2 final Shell shell =
3     new Shell(display, SWT.SHELL_TRIM);
4 shell.setLayout(new FillLayout());
5 Browser browser = new Browser(shell, SWT.NONE);
6 browser.addTitleListener(new TitleListener() {
7     public void changed(TitleEvent event) {
8         shell.setText(event.title);
9     }
10 });
11 browser.setBounds(0, 0, 600, 400);
12 shell.pack();
13 shell.open();
14 browser.setUrl("http://eclipsefaq.org");
15 while(!shell.isDisposed())
16     if(!display.readAndDispatch())
17         display.sleep();
```

Figure 6.2: A code fragment reproduced from the Eclipse Official FAQ [23] and with permission under the Eclipse Public License

```
1 Browser browser = new Browser(shell, SWT.NONE);
2 browser.addTitleListener(new TitleListener() {...});
3 browser.setUrl("http://eclipsefaq.org");
```

Figure 6.3: A summary automatically generated by Konaila from the code fragment in Figure 6.2

as to how the line was presented. There is a large number of decisions involved in summarization.

In the remainder of this chapter, we start with an overview of Konaila (Section 6.1). We describe the design and implementation of Konaila in Sections 6.2 to 6.4. Section 6.5 presents the evaluation study and Section 6.6 summarizes this chapter.

6.1 Overview of Konaila

We highlight the key insights in the summarization algorithm with an example based on the code fragment in Figure 6.2. The code fragment is taken from the Eclipse Official FAQ corpus we constructed for the machine learning case study (Section 4.1.1). The code fragment is designed to answer the question “How do I display a Web page in SWT?” which we took as the query.

Step 1 - Scoring Salient Selection Units (Section 6.3): Konaila determines which code units (called selection units, defined in Section 6.2) are salient candidates for inclusion in the summary. In our example, Konaila chooses candidates based on two salience filters: one favouring calls (Section 6.3.1) and the second favouring event call-backs (Section 6.3.2). Konaila also applies a variable-definition-use filter that boosts the scores of units involved in a variable-definition-use relationship. For the code fragment in Figure 6.2, the top three candidates are:

- the event call-back “changed” on line 7,
- “final Shell shell = new Shell(display, SWT.SHELL_TRIM);” (lines 2-3), and
- “Browser browser = new Browser(shell, SWT.NONE);” (line 5).

Step 2 - Applying Length Constraints and Maximizing Scores (Section 6.4): Konaila eliminates extraneous white-space and determines the number of lines used by the unit when the space is configured at width W . For example, to generate Figure 6.3, we set $W = 50$ characters. Even though the `Shell` declaration statement is a good candidate, when considering that it occupies two lines when formatted to 50-character wide, it is no longer worth-while given that other candidates occupy only one line. This dilemma demonstrates the importance of Konaila’s approach of taking into account the amount of space the unit takes (Section 6.4.1).

With each candidate having a score and a length (in lines), Konaila uses Knapsack optimization [17]. This optimization procedure selects a set of units that maximizes the score of the selected set while limiting the length of the set to be within L lines (here $L = 3$). Certain code units require appropriate context. For example, it is undesirable to include the event call-back on line 7 without the enclosing type on line 6. We made an adjustment to the input to the Knapsack optimization procedure to handle such dependencies (Section 6.4.2). Taking context into account diminishes the importance of the call-back relative to the `Browser` variable declaration statement (line 5).

6.2 Parsing and Unit of Summarization

There are two challenges in parsing code fragments on the web. First, we need to construct a grammar to handle code fragments that are typically incomplete compilation units and that contain non-Java tokens such as “...” (Section 6.2.1). Second, we have to determine the unit of granularity appropriate for summarization (Sections 6.2.2).

6.2.1 Parsing Code Fragments

To handle the first challenge, we modified an existing Java grammar to accept code fragments. This design decision was conceptually and implementation-wise cleaner than transforming a code fragment into a Java compilation unit by enclosing it into an empty type stub, method stub, etc. This stub approach was used in the summarization case study in Chapter 4. Instead, we made changes to a Java grammar. First, we augmented the entry point for Java code fragments from *CompilationUnit* to *MethodDeclaration*, *BlockStatements*, *ClassBodyDeclaration*, etc. We also added tokens such as ellipses, “...”, as valid tokens. Appendix A details the changes to the production rules to the Java grammar to handle code fragments. For implementation, we used the ANTLR parser generator³ and the Java grammar the ANTLR package provides. Currently our focus is on summarizing Java code fragments.

Konaila is only applicable to code fragments that satisfy the grammar, because the ability to parse a code fragment is a requirement for Konaila to perform subsequent summarization steps. Inevitably, there is no grammar that can parse all possible code fragments, some of which may be non-Java code. In Section 6.5.1, we report that out of 64,443 code fragments we extracted from a set of Stack Overflow answers, Konaila correctly accepted 57,261 (90%) Java code fragments, and correctly rejected 1,916 (3%) that we verified separately to be XML fragments. Konaila rejected the remaining of the 5,266 code fragments

³<http://www.antlr.org/>

(7%). We analyzed a random sample of 20 (see Section 6.5.1 for the details) and observed that three were malformed Java code, but we could consider to expand our definition to include in our code fragment grammar. The other seventeen were not source code (e.g., console messages) or were written in different programming languages.

6.2.2 Definition of Selection Units

We defined *selection units* as atomic units to consider for inclusion in a summary. Choosing an appropriate granularity is important: a granularity that is too fine (such as at the token level) unnecessarily increases the number of decisions a summarization algorithm has to make; a granularity that is too coarse does not allow the algorithm to selectively choose important parts of the code.

We define⁴ a selection unit as one of the following Java constructs:

- a statement; when a statement contains a *body*, for example, a block or a single statement in an `if` statement, the unit is the statement excluding the content of the body;⁵
- method signature, type signature, interface signature;
- a field declaration, a package declaration, or an import declaration;
- a comment;⁶
- an anonymous class creation, a method invocation or a constructor invocation extracted from a statement that spans multiple lines.

The motivation behind the extraction is that Java syntax and developer coding styles could allow the formation of overly long statements. For example, Figure 6.4 shows a 30-line statement taken from a code fragment used in our study on summarization practices (Chapter 5). Intuitively, we want to break this statement into multiple units, including:

⁴Appendix B contains additional details on this definition.

⁵We used a place-holder, ..., to represent the body as we exclude it. For the complete definition of a statement selection unit, please refer to Appendix B.2.

⁶Comments include block comments, JavaDoc comments, and line comments.

6.3 Scoring Salient Selection Units

- `new Thread(...).start();`
- `new Runnable() { ... }`
- `@Override public void run() { ... }`
- `int incr;`
- etc.

The extraction algorithm is described in Section [B.3](#) in the appendices. For example, the code fragment in Figure [6.2](#) consists of 14 selection units:

- `Display display = new Display();`
- `final Shell shell = new Shell(display, SWT.SHELL_TRIM);`
- `shell.setLayout(new FillLayout());`
- `Browser browser = new Browser(shell, SWT.NONE);`
- `browser.addTitleListener(new TitleListener() { ... });`
- `public void changed(TitleEvent event) { ... }`
- `shell.setText(event.title);`
- `browser.setBounds(0,0,600,400);`
- `shell.pack();`
- `shell.open();`
- `browser.setUrl("http://eclipsefaq.org");`
- `while (!shell.isDisposed())`
- `if (!display.readAndDispatch())`
- `display.sleep();`

6.3 Scoring Salient Selection Units

The procedure for selecting salient units for inclusion in a summary is motivated mainly by existing research [[42](#), [79](#), [82](#), [85](#), [89](#)] and our summarization study

```
9 new Thread(  
10 new Runnable() {  
11     @Override  
12     public void run() {  
13         int incr;  
14         // Do the "lengthy" operation 20 times  
15         for (incr = 0; incr <= 100; incr+=5) {  
16             // Sets the progress indicator to a max value, the  
17             // current completion percentage, and "determinate"  
18             // state  
19             mBuilder.setProgress(100, incr, false);  
20             // Displays the progress bar for the first time.  
21             mNotifier.notify(id, mBuilder.build());  
22             // Sleeps the thread, simulating an operation  
23             // that takes time  
24             try {  
25                 // Sleep for 5 seconds  
26                 Thread.sleep(5*1000);  
27             } catch (InterruptedException e) {  
28                 Log.d(TAG, "sleep failure");  
29             }  
30         }  
31         // When the loop is finished, updates the notification  
32         mBuilder.setContentText("Download complete")  
33         // Removes the progress bar  
34         .setProgress(0,0,false);  
35         mNotifier.notify(id, mBuilder.build());  
36     }  
37 }  
38 // Starts the thread by calling the run() method in its Runnable  
39 ).start();
```

Figure 6.4: An example of a long statement taken from a code fragment used in our study on summarization practices (Chapter 5). The code fragment is from the Android API Guides [2], demonstrating the task of “displaying a fixed-duration progress indicator.” The code fragment is reproduced with permission under the Apache Software License, Version 2.0.

in Chapter 5. We studied 156 summaries generated by 16 programmers, and interviewed the programmers to ascertain the rationale behind their decision to include a type of code unit in one summary but leave it out of others, and why they chose to present a code unit in a certain way in a summary. With these insights, we constructed four salience filters: the query relevant call filter (Section 6.3.1), the call-back filter (Section 6.3.2), the control flow filter (Section 6.3.3), and the variable-definition-use filter (Section 6.3.4). For each selection unit, these four filters together determine a score.

More formally, each of the filter has a corresponding salience function, $salience_{call}$, $salience_{callBack}$, $salience_{controlFlow}$, and $salience_{defUse}$. Each function takes as input a selection unit vector \vec{u} containing selection units $1, \dots, n$ in a code fragment. The output is a score vector \vec{s} , which is a vector containing non-negative scores for selection units $1, \dots, n$. Konaila first sums up the scores from the call, call-back, and the control flow filters:

$$\vec{s}_{sum} = salience_{call}(\vec{u}) + salience_{callBack}(\vec{u}) + salience_{controlFlow}(\vec{u})$$

Konaila then applies the variable-definition-use filter to the sum to get the final scores:

$$\vec{s}_{final} = salience_{defUse}(\vec{u}, \vec{s}_{sum})$$

If $\vec{s}_{final} = \vec{0}$, Konaila uses a default scoring function (Appendix C). The default function favours selection units that are signatures, contain calls, and match a query term.

6.3.1 Query Relevant Call Filter

This filter emphasizes selecting query relevant calls in a code fragment heavy on API calls:

6.3 Scoring Salient Selection Units

```
1 // Create a constant to convert nanoseconds to seconds.
2 private static final float NS2S = 1.0f / 1000000000.0f;
3 private final float[] deltaRotationVector = new float[4]();
4 private float timestamp;
5
6 public void onSensorChanged(SensorEvent event) {
7     // This timestep's delta rotation to be multiplied by the current ←
8     // rotation
9     // after computing it from the gyro sample data.
10    if (timestamp != 0) {
11        final float dT = (event.timestamp - timestamp) * NS2S;
12        // Axis of the rotation sample, not normalized yet.
13        float axisX = event.values[0];
14        float axisY = event.values[1];
15        float axisZ = event.values[2];
16
17        // Calculate the angular speed of the sample
18        float omegaMagnitude = sqrt(axisX*axisX + axisY*axisY + axisZ*axisZ);
19
20        // Normalize the rotation vector if it's big enough to get the axis
21        // (that is, EPSILON should represent your maximum allowable margin of ←
22        // error)
23        if (omegaMagnitude > EPSILON) {
24            axisX /= omegaMagnitude;
25            axisY /= omegaMagnitude;
26            axisZ /= omegaMagnitude;
27        }
28
29        // Integrate around this axis with the angular speed by the timestep
30        // in order to get a delta rotation from this sample over the timestep
31        // We will convert this axis-angle representation of the delta ←
32        // rotation
33        // into a quaternion before turning it into the rotation matrix.
34        float thetaOverTwo = omegaMagnitude * dT / 2.0f;
35        float sinThetaOverTwo = sin(thetaOverTwo);
36        float cosThetaOverTwo = cos(thetaOverTwo);
37        deltaRotationVector[0] = sinThetaOverTwo * axisX;
38        deltaRotationVector[1] = sinThetaOverTwo * axisY;
39        deltaRotationVector[2] = sinThetaOverTwo * axisZ;
40        deltaRotationVector[3] = cosThetaOverTwo;
41    }
42    timestamp = event.timestamp;
43    float[] deltaRotationMatrix = new float[9];
44    SensorManager.getRotationMatrixFromVector(deltaRotationMatrix, ←
45        deltaRotationVector);
46    // User code should concatenate the delta rotation we computed with ←
47    // the current rotation
48    // in order to get the updated rotation.
49    // rotationCurrent = rotationCurrent * deltaRotationMatrix;
50 }
51 }
```

Figure 6.5: A code fragment reproduced from Android API Guides [2] (used in the summarization study in Chapter 5) with permission under the Apache Software License, Version 2.0. The code fragment has more than half of the lines on mathematical computations.

Motivation

Even though method calls in general have been shown to be essential elements in the comprehension of full programs [42, 82], there is mixed evidence on the universal importance of method calls for the code fragment summarization task. On the one hand, our feature comparison study (Figure 4.6) shows that the method call feature is effective in classifying whether a code line is important for inclusion in a summary. On the other hand, Rodeghero et al.’s eye-tracking study [79] did not find method calls attracted significantly more eye gaze nor fixation as compared to other code constructs during a summarization task.

What is clear is that API method calls are particularly important in code fragments. Code fragments in accepted answers on Stack Overflow usually contain API calls: Subramanian and Holmes’s study on these highly regarded code fragments (21K code fragments from a set of accepted answers on Stack Overflow) reveals 75K calls to various APIs [89]. In addition, in our context, the importance of method calls increases when the name of the call contains a match to a query term, as we found from summarization practices based on query terms in Section 5.3.2. Another condition we observed from the study regarding method calls is that calls are not important in code fragments heavy on computations. For example, the code fragment in Figure 6.5 demonstrates the usage of the gyroscope on an Android phone. The code fragment’s focus is not on the API calls involved, and as such the call in line 41 is arguably not the most important part of the code fragment.

Salience Function

The salience function of the query relevant call salience filter, $salience_{call}(\vec{u})$, is defined as:

$$\begin{cases} \vec{0}, & \text{if } \frac{\sum_{i=1}^n \vec{u}.hasCallsAndConstructor[i]}{n} < 20\% \\ 3 \cdot \vec{u}.matchesQuery + \min(\vec{3}, \vec{u}.nbrCalls) + 3 \cdot \vec{u}.hasConstructor, & \text{otherwise} \end{cases}$$

6.3 Scoring Salient Selection Units

The feature vectors are as follows:

- $\vec{u}.hasCallsOrConstructors$ is a vector, where each element i is 1 or 0 depending on whether selection unit i contains a method or a constructor call;
- $\vec{u}.matchesQuery$ is a vector, where each element i is 1 or 0 depending on whether selection unit i contains an identifier that matches a query term;
- $\vec{u}.nbrCalls$ is a vector, where each element i is the number of method calls (as opposed to constructor calls) a selection unit i contains; and
- $\vec{u}.hasConstructor$ is a vector, where each element i is 1 or 0 depending on whether selection unit i contains a constructor call.

The idea is to assign a zero score vector unless a code fragment is API centric, which is defined as a code fragment containing at least 20% of method or constructor calls. The procedure to determine whether a code fragment is API centric is based on an analysis of the hand-generated summaries and participant interviews in the summarization study in Chapter 5. We empirically determined that the percentage of selection units with method or constructor calls is indicative of whether a code fragment is API centric. Using a threshold of 20% is sufficient to filter out the code fragments that are not API centric such as the gyroscope code fragment in Figure 6.5. Using the code fragment in Figure 6.2 as an example, it is API centric as $\sum_{i=1}^n \vec{u}.hasCallsAndConstructor[i] = 13$ out of $n = 14$ selection units have a method or constructor call (i.e., all the selection units except the method signature “`public void changed(TitleEvent event) { ... }`”).

Intuitively, when a code fragment is API centric, the salience function assigns a higher score to a selection unit that contains multiple method and constructor calls with matches to a query term, while it assigns a lower score to a selection unit that contains fewer calls or does not match a query term. For the code fragment in Figure 6.2, the query is “how do I display a web page in SWT”. The top three selection units have $salience_{call}$ score of 6 because they

contain a matching query term (underlined below) and a constructor call, each of these two features contributing a score of 3:

- `Display display = new Display();`
- `final Shell shell = new Shell(display, SWT.SHELL_TRIM);`
- `Browser browser = new Browser(shell, SWT.NONE);`

6.3.2 Call-Back Filter

This salience filter emphasizes selecting the signature of API event call-backs:

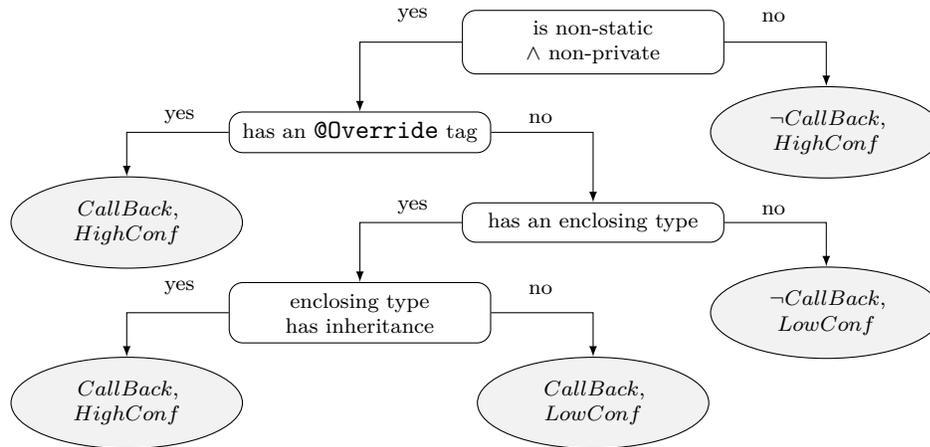
Motivation

Method signatures have been shown to be salient anchors in summarization [79], as we reported in the related work discussion in Section 2.2. As we found from the “Including Overriding Methods” summarization practice in Section 5.3.1, the signature of event call-backs in a code fragment are particularly important to include in a summary. The call-back mechanism is used by a software library or platform to call a piece of code in the application, usually as a reaction to asynchronous events to which an application has subscribed. The application typically responds to this event notification by overriding specific API methods of the library or platform. For example, `onDialogPositiveClick` in Figure 6.1 taken from the summarization study is a call-back in the Android API; a programmer has to override it and registers the call-back in order for the Android platform to notify the application when a dialog’s positive click button has been pressed. One participant called the inclusion of overriding methods a “regular pattern” (Section 5.3.1). From the hand-generated summaries we analyzed, the majority (36 out of 43) of the method declarations with an explicit `@Override` tag were included in a summary by a participant (Section 5.3.1). In contrast, user-defined methods not involved in an event call-back do not tend to be included in the summary (Section 5.3.1).

Input: method signature

Output: $\{Callback, \neg Callback\}$, $\{HighConf \text{ or } LowConf\}$

1: **function** ISCALLBACK



end function

Figure 6.6: Is a method signature a call-back?

Saliency Function

The main intuition for this saliency function is that method signatures that are event call-backs should have a high score, whereas user-defined signatures should have a low score. With only a code fragment, not the whole program, we can only infer without complete confidence whether a method signature is a call-back. We constructed a call-back inference procedure that takes as input a method signature and classifies whether it is a call-back, with either high or low confidence. There are strong clues in two cases: First, a method signature that is a static method or a private method can never be a call-back. Second, a method signature with an `@Override` tag is a call-back with high confidence, as annotating a method signature with the tag provides a Java compiler a way to catch spelling mistakes, when a developer intends to override a method from a super-class but the method does not exist. The annotation is also meant to make it obvious when a method declaration overrides a method in the super-class or interface. A weaker clue to inferring whether a method is involved in a call-back is when method signature is declared `public` and is

not declared `static`, but does not have the enclosing type in the code fragment. Figure 6.6 shows the full procedure for determining whether a method signature is a call-back.

The salience function of the call-back filter is $salience_{callBack}(\vec{u})$, defined as:
 $\vec{u}.callBackVal + 10 \cdot \vec{u}.isAnonClassCB + 10 \cdot \vec{u}.isAnonClassDecl + 3 \cdot \vec{u}.queryBoost$

The feature vectors are as follows:

- $\vec{u}.callBackVal$ is a vector where each element i is defined as:

$$\begin{cases} 10, & \text{if } isHighConfCallBack(u_i) = 1 \text{ and } \sum_{j=1}^n isHighConfCallBack(u_j) = 1 \\ 5, & \text{if } isHighConfCallBack(u_i) = 1 \text{ and } \sum_{j=1}^n isHighConfCallBack(u_j) \geq 2 \\ 3, & \text{if } isLowConfCallBack(u_i) = 1 \text{ and } \sum_{j=1}^n isLowConfCallBack(u_j) \geq 2 \\ 0, & \text{otherwise} \end{cases}$$
 - $isHighConfCallBack(u_i)$ returns 1 if selection unit i contains a method signature inferred to be a call-back with high confidence (Figure 6.6) and 0 otherwise.
 - $isLowConfCallBack(u_i)$ returns 1 if selection unit i contains a method signature inferred to be a call-back with low confidence (Figure 6.6) and 0 otherwise.
- $\vec{u}.isAnonClassCB$ is a vector where an element i is 1 if selection unit i contains a method signature inferred to be a call-back with high confidence (i.e., $isHighConfidenceCallBack(u_i) = 1$) and defined in an anonymous class, and 0 otherwise
- $\vec{u}.isAnonClassDecl$ is a vector where an element i is 1 if selection unit i contains an anonymous class instantiation, and 0 otherwise.
- $\vec{u}.queryBoost$ is a vector where $\vec{u}.queryBoost[i] =$

$$\begin{cases} 1, & \text{if } (\vec{u}.callBackVal[i] > 0 \vee \vec{u}.isAnonClassDecl[i] > 0) \wedge \vec{u}.matchesQuery[i] = 1 \\ 0, & \text{otherwise} \end{cases}$$

Intuitively, $salience_{callBack}$ assigns the highest score to a selection unit when there is only one method signature inferred as a call-back with high confidence,

as the signature is the single piece of important structure in the fragment. Otherwise, the function assigns a lower score to a selection unit. For the code fragment in Figure 6.2, the *salience_{callBack}* score of selection unit “`public void changed(TitleEvent event) { ... }`” is 15 and has a positive score from two features:

- The unit is inferred as a call-back with high confidence (i.e., $isHighConfCallBack = 1$). Because the unit is the only call back in the code (i.e., $\sum_{j=1}^n isHighConfCallBack(u_j) = 1$), the score for *callBackVal* is 10.
- The unit is in an anonymous class (i.e., $isAnonClassCB = 1$), contributing a score of 5.

6.3.3 Control Flow Filter

This salience filter emphasizes on control flow constructs in control flow heavy code:

Motivation

Control flow statements are important, but not necessarily universally important for a summary. For example, Sridhara et al.’s code-to-text summarizer [85] considers code of the form “`if(x==null) return null;`” unimportant. From our analysis of hand-generated summaries reported in Chapter 5, 65 out of 165 of the control flow statements were deemed important enough to be included in a summary (Figure 5.4). One case where control flow constructs are salient is when code fragments contain a lot of computation heavy statements. For example, Figure 6.5 has more than half of the lines on mathematical computations and multiple control flow statements but only one API call. In other cases where a code fragment contains multiple conditional cases, such as an *if* or a *switch* construct, participants tended to include one of the cases, (the “Keeping Only One Case in a Parallel Structure” summarization

practice reported in Section 5.3.1). We observed that when the conditional expression contains a constant, the code tends to be more relevant to a summary (Section 5.4.2), e.g., “if (resultCode == Activity.RESULT_OK)”.

Saliency Function

For this filter, the intuition is that control flow constructs are important only when the code fragment has sufficiently many control flow constructs. We empirically determined that two values are indicative of this intuition: the number of units containing a control flow construct (*for*, *if*, *while*, *switch*, and *do-while*) and the percentage of non-comment units within a loop (a *for* loop, a *while*, or a *do-while*). We empirically determined the thresholds as follows: when a code fragment’s number of control flow constructs is at least 3 or the percentage of units within a loop is greater than 20%.

The saliency function, $saliency_{controlFlow}(\vec{u})$ is defined as follows:

$$\begin{cases} \vec{0}, & \text{if } \sum_{j=1}^n \vec{u}.hasCF[j] < 3 \text{ and } \frac{\sum_{i=1}^n \vec{u}.inCF[i]}{n} \leq 20\% \\ \min(\vec{2} \cdot \vec{u}.nbrCallsInCF) + \vec{u}.constructorInCF + 2 \cdot \vec{u}.constantInCF - 2 \cdot \vec{u}.nullInCF, & \text{otherwise} \end{cases}$$

where the feature vectors include the following:

- $\vec{u}.hasCF$ is a vector, where $\vec{u}.hasCF[i]$ equals to 1 or 0 depending on whether selection unit i contains a control flow construct or not;
- $\vec{u}.inCF$ is a vector, where $\vec{u}.inCF[i]$ equals to

$$\begin{cases} 1, & \text{if selection unit } i \text{ is not a comment and is structurally contained within a loop} \\ 0, & \text{otherwise} \end{cases}$$
- $\vec{u}.nbrCallsInCF$ is a vector, where $\vec{u}.nbrCallsInCF[i] =$

$$\begin{cases} \vec{u}.nbrCalls[i], & \text{if the selection unit } i \text{ contains a control flow construct} \\ 0, & \text{otherwise} \end{cases}$$
- $\vec{u}.constructorInCF$ is a vector, where $\vec{u}.constructorInCF[i] =$

$$\begin{cases} \vec{u}.hasConstructor[i], & \text{if the selection unit } i \text{ contains a control flow construct} \\ 0, & \text{otherwise} \end{cases}$$

- $\vec{u}.constantInCF$ is a vector, where $\vec{u}.constantInCF[i]$ for selection unit i equals to

$$\begin{cases} 1, & \text{if } i \text{ contains a constant that is not } \text{null}, \text{ and a control flow construct} \\ 0, & \text{otherwise} \end{cases}$$
- $\vec{u}.nullInCF$ is a vector, where $\vec{u}.nullInCF[i] =$

$$\begin{cases} 0, & \text{if all of } \vec{u}.nbrOfCallsInCF, \vec{u}.constructorCF, \text{ and } \vec{u}.constantInCF \text{ are zero} \\ 1, & \text{if the selection unit } i \text{ contains a } \text{null} \text{ and a control flow construct} \\ 0, & \text{otherwise} \end{cases}$$

Intuitively, the control flow constructs that contain multiple method calls and constant literals (except for null literals, to avoid code such as “`if (X==null)`”) have the highest score. The control flow constructs that contain fewer number of method calls or that contain no literals have a lower score. For Figure 6.2, there are no selection units with a control flow construct hence the $salience_{controlFlow}$ score vector is $\vec{0}$.

6.3.4 Variable-Definition-Use Filter

This salience filter emphasizes statements involved in simple intra-method variable definition-use relationships:

Motivation

Understanding data-flow relationships in a program is essential in the comprehension and evolution of the code. For example, an important question asked by programmers evolving code is “Where is this variable or data structure being accessed?” [82, page 440]. Answering this question requires understanding of the data-flow of a program. We have also observed this information need from our code fragment summarization task, as one participant explicitly noted: “I am looking where exactly the inflater object is used. This is important because it’s used in the `setView` method.”

Saliency Function

There are many existing techniques for extracting data-flow relationships in a full program [92]. Konaila uses one of the simplest types of data-flow relationships, the relationship between a variable definition and its direct uses (as opposed to a “def-use chain” [1, page 399]): i.e., which statement contains the variable declaration of each variable usage. Intuitively, if a highly scoring statement reads or writes to a variable, the score of the statements containing the definition or uses of this variable should be increased. For example, the query relevant call pattern (Section 6.3.1) determines that the statement “`final Shell shell = new Shell(display, SWT.SHELL_TRIM);`” (lines 2-3 in Figure 6.2) is salient. The saliency of statements using the variable `shell` should be increased accordingly.⁷

The process to obtain relationships between a variable definition and its direct uses is called variable type binding resolution. Typically, type binding resolution is provided by a compiler. However, because a code fragment is often an incomplete compilation unit and is seldom a complete program, in this case typical compilers are unable to provide this information (see Section 3.1.1). We used Partial Program Analysis which uses heuristics to complete an AST from the Eclipse compiler with the missing type bindings [19]. A PPA enhanced AST indicates which variable declaration corresponds to each of the variable uses. For example, the code fragment in Figure 6.2 contains three variables and one formal parameter, “`display`”, “`shell`”, “`browser`”, and “`event`”, defined in lines 1, 2, 5, and 7 respectively. A PPA enhanced AST on this code fragment can identify the uses of these variables and parameter. Our definition-use component uses this information to extract definition-use relationships among the selection units.

We first obtain from the PPA procedure a list of variable declarations including formal parameters, $V_{def} = (v_1, \dots, v_x)$. For the code fragment in

⁷The decision to increase the score for *both* definition and uses is that we want to emphasize if one of the code units is important.

Figure 6.2, V_{def} corresponds to the definition of variables “display”, “shell”, “browser”, and “event.” For each variable $v_i \in V_{def}$, the procedure returns a list of uses which are a set of variables $V_{use_i} = (v_{i1}, \dots, v_{iy})$. For example, for the variable “display” in Figure 6.2, the list of variables that “use” “display” are in “if (!display.readAndDispatch())” and “display.sleep();”. Konaila then transforms V_{def} and $V_{use_1}, \dots, V_{use_x}$ into a list of lists of selection units, as follows. The i -th slot in the outer list corresponds to v_i and refers to the list of selection units each of which contains the definition of variable v_i or one of the variable uses V_{use_i} . We call this list of list of selection units $U_{defUses} = (U_1, \dots, U_k)$. For Figure 6.2, $U_{defUses}$ has four lists, each corresponds to one of “display”, “shell”, “browser”, and “event.” For “display”, the list contains three selection units: (1) “Display display = new Display();”, (2) “if (!display.readAndDispatch())”, and (3) “display.sleep();”. The list for “shell” has seven selection units, the list for “browser” has four, and the list for “event” has two.

The salience function $salience_{defUse}$ is defined in Figure 6.7. The inputs are $U_{defUses} = (U_1, \dots, U_x)$, \vec{s}_{sum} (the score vector output of the sum of the three prior salience functions, $salience_{call}$, $salience_{callBack}$, and $salience_{controlFlow}$), and a map from a selection unit to its s_{sum} . For each U_i (a list of selection units that contains the definition v_i or one of the uses v_{i1}, \dots, v_{iy}), $salience_{defUse}$ propagates the maximum salience score of any unit in the def-use list to all the other elements in the list.

6.4 Optimization and Dependency Handling

The selection of the units to include in a summary does not exclusively depend on whether an individual unit is salient (the score). In our study on summarization practices, we found that participants took into account the amount of space a unit occupies. When two selection units have similar values, Konaila should select the one occupying the fewer number of lines. In addition,

Input:

- $U_{defUses} = (U_1, \dots, U_k)$: where U_i is a list of selection units each of which contains a variable/parameter definition v_i or one of its uses v_{i1}, \dots, v_{iy}
- $s_{sum}^{\vec{}}$: sum of *salience_{call}*, *salience_{callBack}*, and *salience_{controlFlow}*
- m : maps a selection unit u to its corresponding score s_{sum}

Output: \vec{s}_{final} : a score vector containing non-negative scores for selection units $1, \dots, n$

```

1: function SALIENCE_DEFUSE( $U_{defUse}, s_{sum}^{\vec{}}, m$ )
2:    $\vec{s}_{final} \leftarrow \vec{s}_{sum}$  ▷ Initialize the scores
3:   for  $i = 1, \dots, k$  do ▷ For each variable/parameter  $i$ 
4:      $U_i = U_{defUses}[i]$ 
5:      $s_{max} \leftarrow \max_{u_j \in U_i} (m.getScore(u_j))$  ▷ Max score from the units in  $U_i$ 
6:     for  $u_j \in U_i$  do ▷ where  $j$  is the original index in  $s_{final}$ 
7:        $\vec{s}_{final}[j] \leftarrow \vec{s}_{final}[j] + s_{max}$ 
8:     end for
9:   end for
10:   return( $\vec{s}_{final}$ )
11: end function

```

Figure 6.7: The function *salience_{defUse}*

participants considered the combination of units taking coherence into account, as opposed to simply using a greedy algorithm to select the highest scoring units to fill the available space for the summary.

To take into account the space constraints, Konaila uses optimization that maximizes the score of the selected set of units while limiting the length of the set to be within L lines. Konaila also considers three types of dependencies for the coherence of the units selected for inclusion in a summary. The first type of dependency concerns variable definition-use relationships among the selection units, because a variable usage is more likely to make sense with its definition (a declaration statement or a formal parameter of a method). The second and third types of dependencies concern structural context: a method signature is more likely to make sense with its enclosing class signature (if it exists); and `super` or `this` constructor calls are more likely to make sense with their enclosing constructor, if the enclosing constructor exists.

Section 6.4.1 describes the computation of the length of a unit. Section 6.4.2 describes the optimization problem and the changes we made to the optimization problem to handle dependencies.

6.4.1 Determining the Length

First, Konaila determines the number of lines a selection unit occupies after applying our Java code formatting component to each selection unit. Formatting is important for readability not only for source code in general [10], but also for code fragments and their summaries as we found in our study of summarization practices. As we reported in Section 5.4.4, the practice of indenting code is essential for readability: “It’s easier to see the layers, the level of importance. You look at [the code] from top to bottom.”_{p9} In addition, the practice of keeping lines separate is desirable, as opposed to wrapping lines and putting two lines into one; all participants kept lines separate for some of the summaries.

Our formatting component uses the Eclipse Code Style API [22], which re-formats code according to a profile with preferences specified in an XML file

or via the Eclipse UI. We created a profile that aggressively eliminates white spaces and compresses the placement of curly braces and parentheses while respecting the summarization practices regarding indentation and separate lines. One of the preferences is for specifying the line width (in characters) of the formatted code (Appendix D), which we use to specify W , the desired width of the summary. Other preferences concern white spaces within Java constructs and the placement of the braces (Appendix D).

6.4.2 Knapsack Solution and Dependency Handling

Without considering dependencies, the formulation of the Knapsack problem [17] is as follows: Given the set of candidate selection units, each with a salience score s_i and a line length l_i , choose a subset with maximum sum of scores within the line limit L . This formulation is the so-called “0-1 Knapsack” [17], the binary version of the Knapsack problem in which choosing an item (the selection unit) is a binary decision (whether to include the item or not, the x_i ’s) rather than allowing multiple copies of an item:

$$\begin{aligned} & \text{maximize} && \sum_{i=1}^n s_i x_i \\ & \text{subject to} && \sum_{i=1}^n l_i x_i \leq L, \\ & && x_i \in \{0, 1\} \end{aligned}$$

The goal is to maximize the value (score) of the chosen items while satisfying the line limit of the summary. We used a standard implementation [17, page 335] that uses dynamic programming to find the solution to this optimization problem.

To handle the coherence among selection units, we do not take each selection unit as an independent item. Instead, we combine the units involved in dependencies as compound items and use the compound items as input to the Knapsack problem, essentially re-defining the notion of a Knapsack “item”.

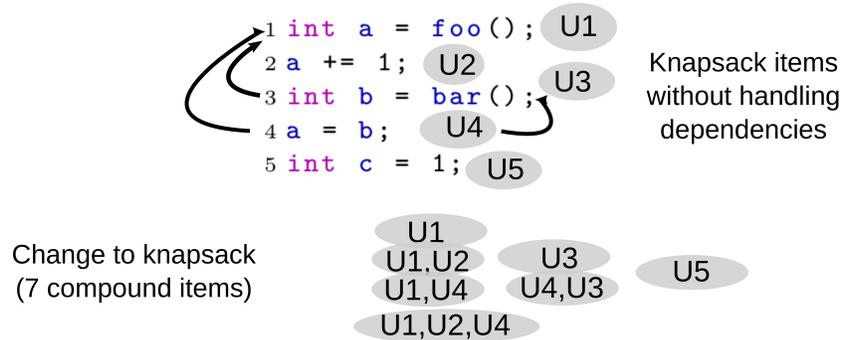


Figure 6.8: An illustration to the change made to the Knapsack algorithm for a code fragment

The pseudo-code for this pre-processing step to the Knapsack items is in Figure 6.9. Intuitively, we do not allow an individual selection unit i involved in a dependency to be defined as an individual Knapsack “item”, essentially ruling out the option to orphan i . We construct a compound item by composing i with a dependency parent p .

Figure 6.8 illustrates a code fragment with two dependencies, U1 (selection unit 1 in line 1) whose dependents are selection units U2 and U4, and U3 that has one dependent U4. Without considering dependencies, there would be five Knapsack “items”, each corresponds to a selection unit in the example (the top left bubble in Figure 6.8). When considering dependencies, if U2 is selected, U1 should be considered to be selected; and if U4 is selected, at least one of U1 or U3 should be considered to be selected. More formally, when a parent p has multiple dependents (i.e., U1 whose dependents are U2 and U4), we enumerate the power set of the dependents (i.e., $\{U2, U4\}$ is $\{\{\}, \{U2\}, \{U4\}, \{U2, U4\}\}$ - line 4 of the pseudo-code) and compose it with p (i.e., U1) as a compound item (yielding four compound items involving U1 in as illustrated in Figure 6.8 - line 6 of the pseudo-code).

Input:

- *units*: set of selection unit objects
 - *u.score*: integer attribute of each $u \in units$, score of u
 - *u.lines*: attribute of each $u \in units$, number of lines u occupies (Section 6.4.1)
- *dependencies*: list of dependencies objects
 - *d.parent*: attribute of each $d \in dependencies$, parent selection unit of c
 - *d.dependents*: attribute of each $d \in dependencies$, list of dependent selection units of c
- L : desired number of lines of the summary

Output: $selected \subset units$: units for inclusion in the summary

```

1: function KNAPSACKWITHPREPROCESSING
2:   compoundItems  $\leftarrow \emptyset$ 
3:   for all  $d \in dependencies$  do
4:     for all  $ps \in \text{POWERSET}(d.dependents)$  do
5:       new(compoundItem)
6:       compoundItem.units  $\leftarrow ps \cup d.parent$ 
7:       compoundItem.lines  $\leftarrow \sum_{u \in \text{compoundItem.units}} u.lines$ 
8:       compoundItem.score  $\leftarrow \sum_{u \in \text{compoundItem.units}} u.score$ 
9:       compoundItems  $\leftarrow \text{compoundItems} \cup \text{compoundItem}$ 
10:    end for
11:  end for
12:  unitsNotInDep  $\leftarrow units \setminus \bigcup_{d \in dependencies} (d.dependents \cup d.parent)$ 
13:  allItems  $\leftarrow \text{compoundItems} \cup \text{unitsNotInDep}$ 
14:  selected  $\leftarrow \text{KNAPSACK}(\text{allItems}, L)$ 
15: end function

```

Figure 6.9: Constructing the input (compound items) to encode dependencies for the knapsack routine

6.5 Evaluation

Task 50 of 52

Stack Overflow Question: Reading List of Node using Xpath

```
public class Parser {
    public static void main(String[] args) throws Exception, Exception {
        final DocumentBuilderFactory factory = DocumentBuilderFactory
            .newInstance();
        final DocumentBuilder builder = factory.newDocumentBuilder();
        final Document doc = builder.parse("src/sitesesh.xml");
        final XPathFactory xPathfactory = XPathFactory.newInstance();
        final XPath xpath = xPathfactory.newXPath();
        final XPathExpression expr = xpath.compile("/sitesesh/mapping");
        Object node = expr.evaluate(doc, XPathConstants.NODE);

        System.out.println(node);
    }
}
```

Summary 1	Summary 2	Summary 3
<pre>final DocumentBuilder builder = factory.newDocumentBuilder(); final XPathExpression expr = xpath.compile("/sitesesh/mapping"); System.out.println(node);</pre>	<pre>final XPathFactory xPathfactory = XPathFactory.newInstance(); final XPath xpath = xPathfactory.newXPath(); System.out.println(node);</pre>	<pre>final XPathExpression expr = xpath .compile("/sitesesh/mapping"); Object node = expr.evaluate(doc, XPathConstants.NODE);</pre>
Given the limited space, this summary captures as much as possible of the original elements of the code related to the Stack Overflow question, while remaining readable.	Given the limited space, this summary captures as much as possible of the original elements of the code related to the Stack Overflow question, while remaining readable.	Given the limited space, this summary captures as much as possible of the original elements of the code related to the Stack Overflow question, while remaining readable.
<input type="radio"/> Completely agree <input type="radio"/> Generally agree <input type="radio"/> Generally disagree <input type="radio"/> Completely disagree	<input type="radio"/> Completely agree <input type="radio"/> Generally agree <input type="radio"/> Generally disagree <input type="radio"/> Completely disagree	<input type="radio"/> Completely agree <input type="radio"/> Generally agree <input type="radio"/> Generally disagree <input type="radio"/> Completely disagree

Save Ratings and Next

Figure 6.10: Snapshot of the web interface used in the evaluation study. The interface presents the three summaries in a random order for each task (a task corresponds to evaluating the three summaries for one fragment). For this task, **Summary 1** is the **Baseline**; **Summary 2** is the **Greedy** summary; and **Summary 3** is the **Optimized** summary (**Konaila**). The code fragment is reproduced from Stack Overflow with permission under the Creative Commons License, Version 2.0.

6.5 Evaluation

To evaluate the quality of the summaries produced by Konaila, we conducted a study with 11 participants to obtain ratings on sets of summaries triplets generated from the same code fragments. We compared summaries produced in three different ways: Optimization-based summarization (Konaila), Baseline (including code units that maximally fill the given space), and Greedy search (greedily choosing candidate selection units with the highest value). Each participant rated 52 sets of summaries. Half of the summary sets were 3 lines by 50 columns and the other half were 5 lines by 50 columns. Figure 6.10 shows the web interface we designed for the study.

The main goal of the evaluation was to determine whether Konaila could generate effective summaries compared to a competitive baseline. We also aimed to determine whether there was any utility in the optimization step (a main contribution of Konaila) by comparing Konaila to the summaries without the step. The three types of summaries generation techniques evaluated were as follows:

Optimization: This is Konaila as described in this chapter.

Baseline: The goal was to generate a competitive baseline: summaries that maximally filled the space. The algorithm first parsed the code to obtain the set of selection units (Section 6.2) and then determined the number of lines when the space was configured at width W (Section 6.4.1). The Baseline algorithm then selected the units with the highest character per line ratio until all available lines were filled. The baseline summaries, like the summaries using the other two generation techniques, uses the formatting profile presented in Section 6.4.1 and does not pack characters densely to the detriment of readability.

Greedy: The goal was to test whether there was any utility in the optimization step (Section 6.4.2). Like the Baseline algorithm, the Greedy algorithm determined the selection units and length in lines for each unit. On top of this, the Greedy algorithm uses the four salience filters to determine the salience scores of the candidate units. With the scores and the line lengths of the candidate units, the Greedy algorithm selects the units with the highest scores until all available lines are. The Greedy algorithm however does not employ optimization nor takes into account hard dependencies (e.g., Section 6.4).

Given that both comparison algorithms (Baseline and Greedy) are conceptually similar and competitive, if the ratings of the Optimized summaries are better than that of the Baseline or Greedy summaries, we can conclude that Konaila's optimization step has value.

6.5.1 Data: Code Fragment Selection

We chose code fragments from Stack Overflow for two Java technologies: Hibernate and Spring. The motivation behind choosing these technologies was that they are popular Java technologies and are different enough from the set of code fragments we selected for the development of the summarization algorithm (the Eclipse official frequently asked questions [23] and Android official API Guides [2]).

In addition, we wanted to choose representative code fragments as the population of the code fragments from which we selected a sample for the experiment. Because one of our motivating applications was for presenting search engine results, we wanted the code fragments in the population to be the ones that a search engine would be likely to return. We thus chose the population of code fragments as follows: We took candidate code fragments extracted from Stack Overflow threads created between September 1, 2008 and March 31, 2015 with the tag “hibernate” or “spring”, in addition to the tag “java”. We selected a window of time large enough so that we could capture high scoring questions that were asked in the past. During that period, there were 27,525 threads created with at least one code fragment enclosed in `<code>` tags. These 27,525 threads contained a total of 64,443 code fragments.

Close to 90% of these code fragments (57,261) satisfied the code fragment grammar we illustrated in Section 6.2. Of the remaining 7182 code fragments that did not satisfy the grammar, 1916 were XML fragments (3% out of 64,443). For the rest (7% out of 64,443) it was not immediately clear the reasons behind the rejection by the grammar; a random sample of 20 from the 7% of code fragments revealed the following types of fragments: malformed Java code fragments (3), invalid XML (9), JavaScript (2), output console messages (2), and a mix of Java and either XML or console messages (3).

From the 27,525 threads, we further eliminated 9,313 threads with all “bad” answers, i.e., threads whose highest scoring answer was 0. From each of the remaining 18,212 threads, we selected one code fragment, the first one from the

highest scoring answer. In the end, we had 4527 code fragments, the population to be sampled for generating the evaluation tasks.

6.5.2 Study Set-Up

For each code fragment, we presented a participant with the task of evaluating the three summary versions. The three summaries were displayed on a web interface horizontally (see Figure 6.10). The top part of the interface is the code fragment extracted from a Stack Overflow *answer post* (as explained in Section 6.5.1). On top of the code fragment is the *title* of the corresponding Stack Overflow question post.⁸ The middle part presents the three summaries in a random order for each task. For this task in Figure 6.10, “Summary 1” is the Baseline; “Summary 2” is the Greedy summary; and “Summary 3” is the Optimized summary (Konaila).

We asked the participants to rate each summary according to the following statement: “Given the limited space, this summary captures as much as possible of the original elements of the code related to the Stack Overflow question, while remaining readable.” The ratings were on a Likert scale with four options: Completely Agree, Generally Agree, Generally Disagree, and Completely Disagree.⁹ We asked the participants to provide a Likert rating per summary rather than to order the three summaries because we wanted to gather the individual ratings in addition to the ordering.

6.5.3 Participants

Of the eleven participants, seven were professional programmers and four were students. One of the professional programmer had submitted patches to both the Hibernate and Spring projects.

For ten of the participants, we randomly assigned 52 fragments from the 4527 fragments to the participants in a way that ensured that the participants’

⁸The full text of the Stack Overflow question post was not available to the raters.

⁹As recommended by Fowler [25].

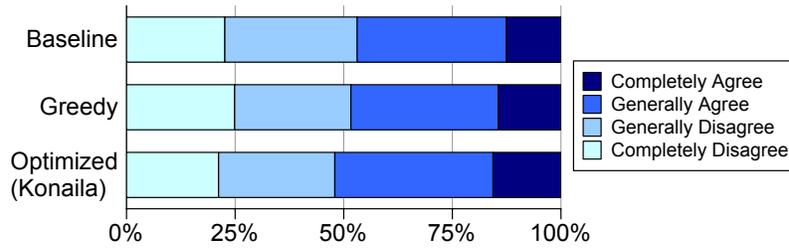


Figure 6.11: Distribution of the Likert ratings

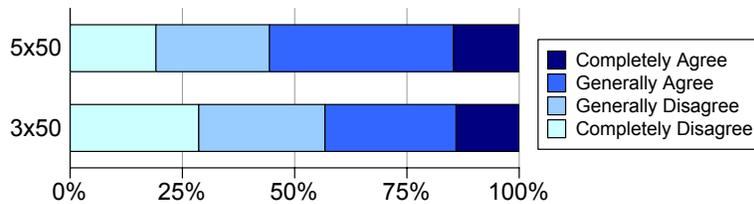


Figure 6.12: Distribution of the Likert ratings between the 3x50 and 5x50 summaries for the Konaila summaries

API knowledge of Hibernate and/or Spring matched the code fragments, that half of the summaries were of 3x50 size and the other half of 5x50 size, and that two participants rated a selected fragment (one for the 3x50 size and the other for 5x50). We set aside one participant for the agreement assessment as described in Section 6.5.5.

We required participants to have one year or more of Java programming experience, and to have at least looked at either the Hibernate or Spring API. The median number of years of Java experience was 4 and the average was 4.73. One participant only had experience with Hibernate, three only with Spring, and seven with both. We recruited the participants from local professional programmer meet-up groups and companies, the McGill School of Computer Science, and through personal contacts. We compensated the participants with \$25 CAD.

6.5.4 Results

After the completion of the data collection, we realized that short code examples almost always generated identical summaries. Therefore, we eliminated the code fragments with fewer than ten non-empty lines from the results from the input pool, leaving us with 364 evaluation tasks (from 520 tasks). The median length of the input fragment was 14 lines, while the mean was 18.5 lines long and standard deviation was 10.3.

Figure 6.11 presents the distribution of the Likert ratings as a stacked bar chart. We observe that according to raters 52.1% of Konaila’s summaries captured as much as possible the original elements of the code related to the Stack Overflow question while remaining readable. Even though 52.1% may seem a marginal proportion above half, we have to interpret this number with the consideration that summarization evaluation is an inherently subjective and complex task.

The median rating of the Optimized summaries was Generally Agree, whereas for Baseline and Greedy summaries, the median ratings were both Generally Disagree. To check whether this rating difference was statistically significant, we used a Wilcoxon signed-rank test. The test is non-parametric and appropriate for ordinal data such as Likert ratings. The null hypothesis is that there is no rating difference between the Optimized-Baseline summary pairs. For each Optimized-Baseline summary pair i , the computation involves taking the difference δ_i in the ratings, and then ranking the differences for all the pairs. The null hypothesis is that the rank of the median δ is equal to 0. The δ would have the largest value if the Optimized summary’s rating were Completely Agree and the Baseline summary’s rating were Completely Disagree; and the smallest if the ratings of the two summaries were flipped. The ranks of tied δ ’s were replaced by the average of the ranks of the ties. We eliminated pairs of summaries that were the same because participants did not need to make a comparison decision; including such pairs would wrongly bias the results towards the null hypothesis. We only included the ratings from

6.5 Evaluation



The image shows a screenshot of a Stack Overflow question titled "Stack Overflow Question: How to use MySQL prepared statement caching?". Below the question are three summaries labeled Summary 1, Summary 2, and Summary 3. Summary 1 is the Greedy summary, Summary 2 is the Baseline, and Summary 3 is the Optimized summary (Konaila).

```
Stack Overflow Question: How to use MySQL prepared statement caching?
Connection conn = DatabaseUtil.getConnection();
PreparedStatement stmtUpdate = conn.prepareStatement("UPDATE foo SET bar=? WHERE id = ?");
for(int id=0; id<10; id++){
    stmtUpdate.setString(1, "baz");
    stmtUpdate.setInt(2, id);
    int rows = stmtUpdate.executeUpdate();
    // Clear parameters for reusing the preparedStatement
    stmtUpdate.clearParameters();
}
conn.close();

Summary 1
PreparedStatement stmtUpdate =
    conn.prepareStatement("UPDATE foo*...");
stmtUpdate.setString(1, "baz");
int rows = stmtUpdate.executeUpdate();
stmtUpdate.clearParameters();

Summary 2
Connection conn =
    DatabaseUtil.getConnection();
PreparedStatement stmtUpdate =
    conn.prepareStatement("UPDATE foo*...");
for(int id = 0; id < 10; id++) {}

Summary 3
PreparedStatement stmtUpdate =
    conn.prepareStatement("UPDATE foo*...");
for(int id = 0; id < 10; id++) {
    stmtUpdate.setInt(2, id);
    int rows = stmtUpdate.executeUpdate();
}
```

Figure 6.13: For this task, **Summary 1** was the **Greedy** summary; **Summary 2** was the **Baseline**; and **Summary 3** was the **Optimized** summary (Konaila).

the ten participants but not the eleventh one from the agreement assessment because including that participant would introduce repeated measures of the same item in the test data.

Results showed that the median rank difference was statistically significantly greater than 0 for the Optimized versus the Baseline summaries ($p = 0.0211$), meaning that the ratings on the Optimized summaries were statistically better than the corresponding Baseline summaries. For the Konaila versus Greedy summaries, the median rank difference was again statistically significantly greater than 0 ($p = 0.0489$). For the Greedy versus Baseline summaries the null hypothesis was not rejected, meaning that these summary pairs were not significantly different in terms of the ratings.

This is a promising result: participants' ratings on the Optimized summaries were overall better than the Baseline summaries. However, using the Greedy did not significantly improve the quality of the of the summaries. *This demonstrates the utility of the optimization step and taking into account the context of the selection units, Section 6.4.2) of the summarization algorithm.*

To compare ratings among the two sizes on the Optimized summaries (Figure 6.12), we again used a Wilcoxon signed-rank test to determine whether the median rank of the ratings on 5x50 summaries are statistically significantly

better than the median rank of the ratings on the 3x50s. The test showed that for the baseline summaries, *5x50 sizes were statistically significantly better ratings than the 3x50 ones* ($p = 0.0125$). While it is not surprising to observe that larger summaries are better than smaller ones, that larger summaries should have better quality than small ones, it was worth-while to verify that the code summarization algorithm behaved according to intuitions from text summarization [33].

The screen-shot from Figure 6.10 shows an example in which the Konaila summary (Summary 3) had a higher rating than the Baseline (Summary 1) by two raters (one from the ten participants and the other one being the “agreement” rater which will be described in Section 6.5.5). The Baseline summary

```
1 final DocumentBuilder builder =
2     factory.newDocumentBuilder();
3 final XPathExpression expr =
4     xpath.compile("/sitemesh/mapping");
5 System.out.println(node);
```

contains three code units that are unrelated, whereas the Konaila summary

```
1 final XPathExpression expr = xpath
2     .compile("/sitemesh/mapping");
3 Object node = expr.evaluate(doc,
4     XPathConstants.NODE);
```

demonstrates the utility of the variable definition-use dependency for generating a coherent sequence of units, i.e., the definition and use of the variable `expr`.

Figure 6.13 demonstrates another example where two raters both judged the Konaila summary (Summary 3) as better than the Baseline summary (Summary 2). The Konaila summary

```
1 PreparedStatement stmtUpdate =
2     conn.prepareStatement("UPDATE foo"...);
3 for(int id = 0; id < 10; id++) {
4     stmtUpdate.setInt(2, id);
5     int rows = stmtUpdate.executeUpdate();}
```

demonstrates that the control flow salience filter was able to correctly assert the importance of the loop for the summary.

6.5.5 Agreement

To investigate how much different participants agree or disagree on the summary ratings, we asked the eleventh participant to perform evaluation tasks that overlapped with the other ten participants in the following way:

We randomly selected 52 evaluation tasks from the pool of tasks performed by the ten participants. The resulting tasks consisted of a mix of 27 3x50 and 25 5x50 summaries. Of the 52 code fragments, 24 were Hibernate and 28 Spring code fragments. This eleventh participant also used the same web interface as the other ten participants to complete the 52 evaluation tasks.

For each code fragment and the three corresponding summaries, we converted the ratings into two pair-wise comparisons: whether the ratings of the Optimized summaries were better or the same than the corresponding Baseline summary, and similarly for Optimized vs. Greedy summaries. We tabulated these numbers into a 2x2 contingency table, with each dimension distinguishing whether the rating of an Optimized or a Greedy summary was better or the same as the corresponding Baseline summary. Using this definition, the agreement between the first participant with the rest of the ten participants had a Cohen's Kappa of 0.465 [16]. The level of agreement of a Kappa between 0.4 and 0.6 is considered a moderate agreement [94]. Given the inherent subjectivity in summarization, we were satisfied that this value indicated reasonable reliability for the results.

6.6 Chapter Summary

In this chapter, we reported on Konaila, an optimization-based algorithm that makes use of four salience filters motivated from our formative research. Based on 364 summaries rated by ten participants, Konaila's summaries were statistically significantly better at capturing the original elements of the code related to the Stack Overflow question while remaining readable, compared to a competitive baseline that included code units that maximally fill the given

space. In addition, the optimization step is an essential part in the effectiveness of Konaila's summaries. The contribution of this chapter is that an approach such as Konaila, based on the use of meaningful code units, a two-dimensional formulation, simple-to-compute features (based on code constructs, the overlap with the given query, and simple variable definition-use analysis. We observed that raters agreed that these summaries captured the essential elements of the original code.

Chapter 7

Conclusion

Code examples are important in software development because of their ability to concretely demonstrate a solution and their ubiquity in both formal documentation and forums on the web. The motivation for summarizing code examples, or more generally, code fragments, is to enable their more effective use, by generating code fragment summaries that can benefit scenarios such as providing cues for long fragments in a documentation index or displaying search results. The code fragment summaries we generate can replace those summaries that treat code as text from search engines (e.g., Figure 1.3) and Stack Overflow (e.g., Figure 1.4), and ineffective summaries from code search engines (e.g., Figure 1.5).

In this dissertation, we presented research in code fragment summarization that makes three contributions to the field of software engineering. The first contribution is the lessons learned from a case study on the generation line-based summaries using a supervised machine learning approach. We had relative success in using a combination of features that take advantage of the syntactic structure of the code and the query. In addition, we found three limitations in the problem formulation and the approach: the limitation in using line granularity; the difficulty in obtaining data; and limitations on features local to a line without considering dependencies among different parts of the code.

The second contribution is the catalog of summarization practices consisting of three types of selection practices and five types of presentation practices, based on empirical evidence on how human summarize code. The practices reinforce the importance of the syntactic structure and the query in the selection of summary content. The practices also reveal that generating summaries is not only about shortening the code but also about making it compilable, readable, and understandable.

The final contribution is the design, implementation, and evaluation of an optimization-based approach for constructing summaries constrained in both height and width. We based the design decisions on the lessons learned from the case study and the catalog of summarization practices from the empirical study:

- We formulated the summaries to respect a novel width constraint in the summarization problem formulation that took into account the importance of readability as found in the empirical study.
- Due to the limitations of line-granularity and together with insights from the summarization practices, we defined *selection units* with a granularity roughly at the code statement level. These units were our atomic code units for summarization.
- Given the difficulty in obtaining training data and the inherent subjectivity in the summarization problem, we devised a rule-based approach that does not depend on data.
- We employed syntactic- and query-based features, motivated by the success of this combination of features demonstrated by the case study. In addition, Konaila uses a feature that model dependencies of the selection units, specifically, variable definition-use dependencies.

With this design, we implemented and evaluated Konaila. The summaries were statistically significantly better at capturing the original elements of the code related to the Stack Overflow question while remaining readable, compared to

a competitive baseline that included code units that maximally fill the given space. In addition, the optimization step is an essential part in the effectiveness of Konaila’s summaries.

Beyond these three contributions, this dissertation also provides a broader perspective of code fragment summarization via the description of five dimensions in the design space of code summarization.

7.1 Future Work

We envision future work both in our specific research and in the general area of code fragment summarization.

Machine Learning in Konaila

One immediate future work idea on Konaila would be to use a machine learning approach in two places in the Konaila algorithm. The first one would be to learn from data for what conditions a summarization filter (Section 6.3) is applicable to an code fragment. For example, we only apply the query relevant call summarization pattern when a code fragment as API centric, i.e., when 20% of the selection units contain method or constructor calls. To learn the thresholds, we would need a corpus of code fragments where each code fragment is annotated with whether or not a salience filter is applicable. A data-driven approach could also help in determining the thresholds in the salience functions (Section 6.3). For this problem, we would need to collect a corpus of original-summary pairs of code fragments. The granularity would be at the selection unit level (Section 6.2), rather than at the line level as used in the case study. The particular field of machine learning that could be useful is preference learning [34].

Experimentation in an Application Context

External validity of code fragment summaries needs to be appraised in the context of a deployed application. Additional validation could involve a controlled experiment where half of the participants are assigned to perform programming-related tasks with Konaila’s code fragment summaries (i.e., the treatment group), while the other half of the participants are assigned to use the baseline summaries (i.e., the control group). Some examples of programming-related tasks include selecting relevant code examples from a search engine result page modified with code fragment summaries, and selecting a relevant code example from a document index enhanced with code fragment summaries. The efficacy of each group can be measured using the task completion time, the amount of information needs satisfied by the summary, and user satisfaction. With these experiments we can gain a better understanding on whether code fragment summaries are useful in a more realistic application scenario.

Inherent Difficulty in Summarization

Evaluating summaries is a difficult problem because the notion of correctness of a summary is elusive. In computational linguistics, the evaluation criterion of certain problems (e.g., machine translation [70]) can be defined relatively more precisely, while in text summarization, it is difficult to obtain reliable human judgment on which content should be included in a summary [49]. To alleviate this problem in text summarization, the notion of correctness is typically constructed via summary lines more agreed upon by the annotators. Similarly, in software engineering, the evaluation criterion of certain problems is more precise than others. For example, there is a ground truth when it comes to whether a method causes a crash. There is no need to consult multiple annotators.

In three of our experiments (Sections 4.3 to 4.5), the notion of correctness we used was the gold standard summaries which consist of lines that were marked as in-summary by at least n annotators. In our experiments, n was two out of

the total of four annotators (Section 4.1.1). However, this definition assumes that a code fragment has a single universally correct summary. The moderately low agreement among the annotators suggests a different assumption on the correctness of a summary: no single correct summary exists. Rather, each annotator's version of the summary is correct. This issue has been recognized in text summarization [67]: experiments have shown that the human summaries themselves scored poorly when the correctness is defined by the other three of the four annotators. Correspondingly, pyramid precision was proposed as a metric [68]. Pyramid precision assumes that there is no single best model and weighs more agreed upon units more heavily than less agreed upon units. We used this metric in one of the experiments (Section 4.6.2). Initial results found that pyramid precision had a linear relationship with R-precision. This result needs to be expanded, replicated, or refuted in future work.

Building on this realization, one line of future work is to ascertain which evaluation metrics for code fragment summarization are reliable. Do evaluation metrics in text summarization reliably measure the quality of code fragment summaries? What are the strengths and weaknesses of these metrics? To adequately interpret results from a summarization experiment, researchers in the field of software engineering need to recognize the inherent subjectivity in the code fragment summarization task.

Personalization

When attempting to quantify the correctness of a summary using an oracle generated by multiple annotators, the conventional treatment is that there is only one correct summary which uses a set of summary lines more agreed upon by the annotators. However, if we assume that each annotator's version of the summary is correct, a better summarizer would be one that personalizes the summary for each individual. The personalization direction is also apparent from our empirical study of summarization practices. Participants had the human reader in mind, for example, by including code deemed easy to miss

by the reader and excluding code deemed obvious assuming the reader had previous knowledge of the API (Section 5.3.3). Incorporating the cognitive model of a reader into a summarizer is a promising area of future work.

The implementation of systems taking into account this cognitive model involves building a representation that captures the knowledge of the human themselves. Our book chapter [101] describes building such a representation, called developer profiles. Using expertise modeling, a developer profile can capture code deemed easy to miss and code deemed obvious. Existing measures on expertise—specifically, API-related expertise—of a programmer are mostly based on the number of times an API method has been used in the programmer’s code. This information is then used for recommending the right programmer with a desired expertise [27, 52, 60, 104]. To identify which programmer added or modified an API method, such algorithms typically use source code version history to see which programmer added, deleted, or modified which API method call to the client source code.

Comparing Code-to-Code Summarization with Code-to-Text

Is the format of code fragment ultimately the optimal output format? One interesting direction is to compare the efficacy of code fragment output versus textual output (e.g., code comments), when summarizing a given code fragment. In which usage scenarios are a code fragment output more effective, and in which scenarios are a textual output more effective? Such questions were also central in experiments comparing whether a visual representation or a textual representation is more effective in search engine results (e.g., [4]), or a combination of text and image thumbnails [90, 99].

There is some evidence that a mix of code and text is desirable. From the empirical study on summarization practices, we saw that when programmers were asked to shorten the code, participants employ natural language text. In the *shortening method declarations* practice (Section 5.4.2), we observed summaries with both code and natural language. Rastkar et al. [73] found

some success in summarizing a commit using a combination of code elements and natural language (Section 5.5.3). The patterns found in their summaries include listing the method declarations changed in a commit and using lexical aggregation (a way to summarize a list of elements with a few words rather than explicitly listing the methods [75]) to describe a commit. For example, “all of the methods involved in implementing ‘Undo’ are named `undo`” [73] is an example of lexical aggregation. We observed both patterns (listing and lexical aggregation) in our empirical study (Figure 5.5). This mix of different media (code and text) is also related to work in natural language generation that combines graphics and text [95].

7.2 Closing Remark

Given code examples are sought-after, effective, and abundant on the web, there is more need than ever for programmers to search for these code examples effectively. While searching for images on Google Images ¹ we can take advantage of image thumbnails in the search results and while searching for videos on You Tube ² we can take advantage of video snapshots. There is limited knowledge on how to summarize code examples, or code fragments in general. Our research in code fragment summarization is a step towards this direction that I believe will ultimately benefit programmers working with code examples.

¹<http://images.google.com/>, or any on-line image search service

²<http://www.youtube.com>, or any an on-line video hosting service

Bibliography

- [1] W. A. Andrew and P. Jens. *Modern compiler implementation in Java*. Cambridge, 2002.
- [2] Android Documentation. API Guides. <https://developer.android.com/guide/index.html>.
- [3] M. Asaduzzaman, A. S. Mashiyat, C. K. Roy, and K. A. Schneider. Answering questions about unanswered questions of stack overflow. In *Proceedings of the Working Conference on Mining Software Repositories*, pages 97–100, 2013.
- [4] A. Aula, R. M. Khan, Z. Guan, P. Fontes, and P. Hong. A comparison of visual and textual page previews in judging the helpfulness of web pages. In *Proceedings of the International Conference on World Wide Web*, pages 51–60, 2010.
- [5] S. Bajracharya, J. Ossher, and C. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, pages 157–166, 2010.
- [6] J. Brandt, M. Dontcheva, M. Weskamp, and S. Klemmer. Example-centric programming: integrating web search into the development environment.

BIBLIOGRAPHY

- In *Proceedings of the International Conference on Human Factors in Computing Systems*, pages 513–522, 2010.
- [7] J. Brandt, P. Guo, J. Lewenstein, M. Dontcheva, and S. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 1589–1598, 2009.
- [8] A. Broder. A taxonomy of web search. *SIGIR Forum*, 36(2):3–10, 2002.
- [9] B. Burd. *Android application development all-in-one for dummies*. For Dummies, 2011.
- [10] R. Buse and W. Weimer. A metric for software readability. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 121–130, 2008.
- [11] R. Buse and W. Weimer. Synthesizing API usage examples. In *Proceedings of the International Conference on Software Engineering*, pages 782–792, 2012.
- [12] R. P. Buse and W. R. Weimer. Automatic documentation inference for exceptions. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 273–282, 2008.
- [13] R. P. Buse and W. R. Weimer. Automatically documenting program changes. In *Proceedings of the International Conference on Automated Software Engineering*, pages 33–42, 2010.
- [14] S. Chatterjee, S. Juvekar, and K. Sen. SNIFF: A search engine for Java using free-form queries. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, pages 385–400, 2009.

BIBLIOGRAPHY

- [15] C. Clarke, E. Agichtein, S. Dumais, and R. White. The influence of caption features on clickthrough patterns in web search. In *Proceedings International SIGIR Conference on Research and Development in Information Retrieval*, pages 135–142, 2007.
- [16] J. Cohen et al. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.
- [17] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT press, 1990.
- [18] E. Cutrell and Z. Guan. What are you looking for? an eye-tracking study of information usage in web search. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 407–416, 2007.
- [19] B. Dagenais and L. Hendren. Enabling static analysis for partial Java programs. In *Proceedings of the Conference on Object Oriented Programming Systems and Applications*, pages 313–328, 2008.
- [20] R. DeLine, M. Czerwinski, B. Meyers, G. Venolia, S. Drucker, and G. Robertson. Code thumbnails: Using spatial memory to navigate source code. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*, pages 11–18, 2006.
- [21] E. Duala-Ekoko and M. Robillard. Asking and answering questions about unfamiliar APIs: An exploratory study. In *Proceedings of the International Conference on Software Engineering*, 2012.
- [22] Eclipse Documentation. Code Formatter Preferences. <http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fpreferences%2Fjava%2Fcodestyle%2Fref-preferences-formatter.htm>.
- [23] Eclipse Documentation. Official FAQs. https://wiki.eclipse.org/The_Official_Eclipse_FAQs.

BIBLIOGRAPHY

- [24] L. M. Eshkevari, V. Arnaoudova, M. Di Penta, R. Oliveto, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of identifier renamings. In *Proceedings of the Working Conference on Mining Software Repositories*, pages 33–42, 2011.
- [25] F. J. Fowler. *Improving survey questions: Design and evaluation*, volume 38. Sage, 1995.
- [26] M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley Professional, 1999.
- [27] T. Fritz, J. Ou, G. Murphy, and E. Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *Proceedings of the International Conference on Software Engineering*, pages 385–394, 2010.
- [28] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *Proceedings of the Working Conference on Reverse Engineering*, pages 35–44, 2010.
- [29] W. Harrison. Rpd3: A framework for integrating tool fragments. *IEEE Software*, 4(6):46, 1987.
- [30] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *International Conference on Software Engineering*, pages 837–847, 2012.
- [31] R. Hoffmann, J. Fogarty, and D. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the Symposium on User Interface Software and Technology*, pages 13–22, 2007.
- [32] R. Holmes and G. Murphy. Using structural context to recommend source code examples. In *Proceedings of the International Conference on Software Engineering*, pages 117–125, 2005.

BIBLIOGRAPHY

- [33] E. Hovy. Text summarization. In R. Mitkov, editor, *The Oxford Handbook of Computational Linguistics*, chapter 32, pages 583–598. Oxford University Press Oxford, 2003.
- [34] E. Hüllermeier, J. Fürnkranz, W. Cheng, and K. Brinker. Label ranking by learning pairwise preferences. *Artificial Intelligence*, 172(16):1897–1916, 2008.
- [35] H. Jing, R. Barzilay, K. McKeown, and M. Elhadad. Summarization evaluation methods: Experiments and analysis. In *AAAI Symposium on Intelligent Summarization*, pages 51–59, 1998.
- [36] H. Jing and K. R. McKeown. The decomposition of human-written summary sentences. In *Proceedings of the Annual International SIGIR Conference on Research and Development in Information Retrieval*, pages 129–136, 1999.
- [37] J. Kim, S. Lee, S.-W. Hwang, and S. Kim. Towards an intelligent code search engine. In *AAAI Conference on Artificial Intelligence*, pages 1358–1363, 2010.
- [38] J. Kim, S. Lee, S.-W. Hwang, and S. Kim. Enriching documents with examples: A corpus mining approach. *Transactions on Information Systems*, 31(1):1–27, 2013.
- [39] S. Kim, E. J. Whitehead Jr, and Y. Zhang. Classifying software changes: Clean or buggy? *Transactions on Software Engineering*, 34(2):181–196, 2008.
- [40] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [41] D. E. Knuth. *Literate programming*. Stanford, California: Center for the Study of Language and Information, 1992.

BIBLIOGRAPHY

- [42] A. J. Ko, B. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *Transactions on Software Engineering*, 32(12):971–987, 2006.
- [43] J. Kupiec, J. Pedersen, and F. Chen. A trainable document summarizer. In *Proceedings of the Annual International Conference on Research and Development in Information Retrieval*, pages 68–73, 1995.
- [44] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, pages 159–174, 1977.
- [45] D. Lawrie and D. Binkley. Expanding identifiers to normalize source code vocabulary. In *Proceedings of the International Conference on Software Maintenance*, pages 113–122, 2011.
- [46] U. Lee, Z. Liu, and J. Cho. Automatic identification of user goals in web search. In *Proceedings of the International Conference on World Wide Web*, pages 391–400, 2005.
- [47] C. Lewis and J. Rieman. *Task-Centered User Interface Design: A Practical Introduction*, chapter 5: Testing The Design With Users. Self-published, 1993. http://grouplab.cpsc.ucalgary.ca/saul/hci_topics/tcsd-book/contents.html.
- [48] H. Li, Z. Xing, X. Peng, and W. Zhao. What help do developers seek, when and how? In *Proceedings of the Working Conference on Reverse Engineering*, pages 142–151, 2013.
- [49] C.-Y. Lin and E. Hovy. Manual and automatic evaluation of summaries. In *Proceedings of the Workshop on Automatic Summarization*, pages 45–51, 2002.

BIBLIOGRAPHY

- [50] R. Lotufo, Z. Malik, and K. Czarnecki. Modelling the "hurried" bug report reading process to summarize bug reports. In *International Conference on Software Maintenance*, pages 430–439, 2012.
- [51] H. P. Luhn. The automatic creation of literature abstracts. *IBM Journal of Research and Development*, 2(2):159–165, 1958.
- [52] D. Ma, D. Schuler, T. Zimmermann, and J. Sillito. Expert recommendation with usage expertise. In *Proceedings of the International Conference on Software Maintenance*, 2009.
- [53] I. Mani. *Automatic summarization*. John Benjamins Publishing, 2001.
- [54] S. Mani, R. Catherine, V. Sinha, and A. Dubey. Ausum: approach for unsupervised bug report summarization. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, page 11, 2012.
- [55] C. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.
- [56] D. Marcu, E. Amorrortu, and M. Romera. Experiments in constructing a corpus of discourse trees. In *Proceedings of the Workshop on Standards and Tools for Discourse Tagging*, pages 48–57, 1999.
- [57] R. C. Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.
- [58] S. McLellan, A. Roesler, J. Tempest, and C. Spinuzzi. Building more usable APIs. *IEEE Software*, 15(3):78–86, 1998.
- [59] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering*, 2011.

BIBLIOGRAPHY

- [60] A. Mockus and J. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the International Conference on Software Engineering*, pages 503–512, 2002.
- [61] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of the Working Conference on Reverse Engineering*, pages 13–22, 2001.
- [62] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic Generation of Natural Language Summaries for Java Classes. In *Proceedings of the International Conference on Program Comprehension*, pages 23–32, 2013.
- [63] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus. How can I use this method? In *Proceedings of International Conference on Software Engineering*, 2015.
- [64] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, and G. Canfora. Automatic generation of release notes. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 484–495, 2014.
- [65] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns. What makes a good code example?: A study of programming Q&A in StackOverflow. In *Proceedings of the International Conference on Software Maintenance*, pages 25–34, 2012.
- [66] A. Nenkova and K. McKeown. Automatic summarization. *Foundations and Trends in Information Retrieval*, 5(2–3):122–233, 2011.
- [67] A. Nenkova and R. Passonneau. Evaluating content selection in summarization: The pyramid method. In *Proceedings of Human Language Technology Conference and the North American Chapter of the Association for Computational Linguistics Annual Meeting*, 2004.

BIBLIOGRAPHY

- [68] A. Nenkova, R. Passonneau, and K. McKeown. The pyramid method: Incorporating human content selection variation in summarization evaluation. *ACM Transactions on Speech and Language Processing (TSLP)*, 4(2):4, 2007.
- [69] T. Paek, S. Dumais, and R. Logan. Wavelens: A new view onto internet search results. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 727–734, 2004.
- [70] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the Annual Meeting on Association for Computational Linguistics*, pages 311–318, 2002.
- [71] C. Parnin, C. Treude, L. Grammel, and M.-A. Storey. Crowd documentation: Exploring the coverage and the dynamics of API discussions on stack overflow. Technical Report GIT-CS-12-05, Georgia Institute of Technology, 2012.
- [72] D. Radev, E. Hovy, and K. McKeown. Introduction to the special issue on summarization. *Computational Linguistics*, 28(4):399–408, 2002.
- [73] S. Rastkar, G. C. Murphy, and A. W. Bradley. Generating natural language summaries for crosscutting source code concerns. In *Proceedings of the International Conference on Software Maintenance*, pages 103–112, 2011.
- [74] S. Rastkar, G. C. Murphy, and G. Murray. Summarizing software artifacts: a case study of bug reports. In *Proceedings of the International Conference on Software Engineering*, pages 505–514, 2010.
- [75] M. Reape and C. Mellish. Just what is aggregation anyway. In *Proceedings of the European Workshop on Natural Language Generation*, pages 20–29. Citeseer, 1999.

BIBLIOGRAPHY

- [76] E. Reiter and R. Dale. *Building natural language generation systems*, volume 152. MIT Press, 2000.
- [77] M. Robillard. What makes APIs hard to learn? Answers from developers. *IEEE Software*, 26(6):27–34, 2009.
- [78] M. Robillard and R. DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.
- [79] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D’Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the International Conference on Software Engineering*, pages 390–401, 2014.
- [80] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5):513–523, 1988.
- [81] C. B. Seaman. Qualitative methods in empirical studies of software engineering. *Transactions on Software Engineering*, 25(4):557–572, July/August 1999.
- [82] J. Sillito, G. C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *Transactions on Software Engineering*, 34(4):434–451, 2008.
- [83] S. Sim, R. Gallardo-Valencia, K. Philip, M. Umarji, M. Agarwala, C. Lopes, and S. Ratanotayanon. Software reuse through methodical component reuse and amethodical snippet remixing. In *Proceedings of the Conference on Computer Supported Cooperative Work*, pages 1361–1370, 2012.
- [84] S. Sim, M. Umarji, S. Ratanotayanon, and C. Lopes. How well do search engines support code retrieval on the web? *Transactions on Software Engineering and Methodology*, 21(1):4, 2011.

BIBLIOGRAPHY

- [85] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *Proceedings of the International Conference on Automated Software Engineering*, pages 43–52, 2010.
- [86] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the International Conference on Software Engineering*, pages 101–110, 2011.
- [87] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *Proceedings of the International Conference on Program Comprehension*, pages 71–80, 2011.
- [88] J. Stylos and B. Myers. Mica: A web-search tool for finding API components and examples. In *Symposium on Visual Languages and Human-Centric Computing*, pages 195–202, 2006.
- [89] S. Subramanian and R. Holmes. Making sense of online code snippets. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 85–88, 2013.
- [90] J. Teevan, E. Cutrell, D. Fisher, S. Drucker, G. Ramos, P. André, and C. Hu. Visual snippets: summarizing web pages for search and revisitation. In *Proceedings of the International Conference on Human Factors in Computing Systems*, pages 2023–2032, 2009.
- [91] S. Teufel and M. Moens. Sentence extraction as a classification task. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pages 58–65, 1997.
- [92] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.
- [93] E. R. Tufte. *Beautiful evidence*, volume 23. Graphics Press Cheshire, CT, 2006.

BIBLIOGRAPHY

- [94] A. J. Viera, J. M. Garrett, et al. Understanding interobserver agreement: the kappa statistic. *Family Medicine*, 37(5):360–363, 2005.
- [95] W. Wahlster, E. André, W. Finkler, H.-J. Profitlich, and T. Rist. Plan-based integration of natural language and graphics generation. *Artificial Intelligence*, 63(1):387–427, 1993.
- [96] R. White, J. Jose, and I. Ruthven. A task-oriented study on the influencing effects of query-biased summarisation in web searching. *Information Processing & Management*, 39(5):707–733, 2003.
- [97] R. White, I. Ruthven, and J. Jose. Finding relevant documents using top ranking sentences: an evaluation of two alternative schemes. In *Proceedings of the International Conference on Research and Development in Information Retrieval*, pages 57–64, 2002.
- [98] E. Wong, J. Yang, and L. Tan. AutoComment: Mining question and answer sites for automatic comment generation. In *Proceedings of the International Conference on Automated Software Engineering, New Ideas Track*, pages 562–567, 2013.
- [99] A. Woodruff, A. Faulring, R. Rosenholtz, J. Morrisson, and P. Pirolli. Using thumbnails to search the web. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 198–205, 2001.
- [100] A. T. T. Ying and M. P. Robillard. Code fragment summarization. In *Proceedings of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 655–658, 2013.
- [101] A. T. T. Ying and M. P. Robillard. Developer profiles for recommendation systems. In *Recommendation Systems in Software Engineering*, pages 199–222. Springer, 2014.

BIBLIOGRAPHY

- [102] A. T. T. Ying and M. P. Robillard. Selection and presentation practices for code example summarization. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 460–471, 2014.
- [103] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 318–343, 2009.
- [104] L. Zou and M. Godfrey. Understanding interaction differences between newcomer and expert programmers. In *International Workshop on Recommendation Systems for Software Engineering*, pages 26–29, 2008.

BIBLIOGRAPHY

Appendix A

Changes to the Java Grammar to Parse Code Fragments

We made two types of changes to the Java grammar to handle code fragments. First, we augmented the entry point for Java code fragments from *CompilationUnit* to *MethodDeclaration*, *BlockStatements*, *ClassBodyDeclaration*, etc. The following is the initial production rule we added to the Java grammar:

```
javaFragment           // - The entry point for Java code fragments
:   compilationUnit   // - The usual Java entry point
|   methodDeclaration
|   blockStatements
|   classBodyDeclaration*
|   classBodyDeclaration* blockStatements
|   switchBlockStatementGroup*;
```

We also added ellipses of different lengths (from length of two to four, i.e., “..”, “...”, “....”) as valid tokens in three places, as a *blockStatement*, as a *classBodyDeclaration*, and as a body in an *if* statement (Figure A.1).

```

blockStatement
:   localVariableDeclarationStatement
  |   statement
  |   typeDeclaration
  |   ellipsisToken      // Added to handle Java code fragments
;

classBodyDeclaration
:   ';'
  |   'static'? '{' blockStatements '}'
  |   memberDeclaration
  |   ellipsisToken      // Added to handle Java code fragments
;

ifStatement
:   'if' parExpression (ifBody)? ('else' (elseBody))? // Made ifBody ←
    and elseBody optional for Java code fragments
;

ellipsisToken
:   '...'
  |   '...'
  |   '...'
;

```

Figure A.1: Three production rules that allow ellipses

Appendix B

Definition of Selection Units

We presented a definition of a selection unit as one of the following Java constructs in Section 6.2.2:

- a statement; when a statement contains a *body*, for example, a block or a single statement in an `if` statement, the unit is the statement excluding the content of the *body*;¹
- method signature, type signature, interface signature;
- a field declaration, a package declaration, or an import declaration;
- a comment; or
- an anonymous class creation, a method invocation or a constructor invocation *sub-divided* from a statement that spans multiple lines.

In this appendix, we first describe in Section B.1 the parent reference of a selection unit. We then expand on the definition for constructs with a body (Section B.2), and provide a description for the sub-division algorithm (Section B.3).

¹We used a place-holder, ..., to represent the body as we exclude it.

B.1 Parent Reference

For each selection unit, we maintain a reference to the immediate syntactically enclosing unit. For example, the unit `“String action = intent.getAction();”` (line 3 in Figure B.1) has a reference to the enclosing unit `“public void onReceive(Context context, Intent intent) { ... }”` (lines 2 and 11) which in turn has a reference to `“BroadcastReceiver mUsbReceiver = new BroadcastReceiver() { ... };”` (lines 1 and 12).

B.2 Constructs with a Body

For statements with a body (control flow statements, *synchronized* statements, and *try* statements) or signatures, the selection unit is defined as the *header* part of the construct, excluding the body. When such a construct is in one line, we keep it the same unit (e.g., `try { Thread.sleep(1000); } catch (Exception e) { }`). We use a place-holder, “...” to represent the body that is excluded. The following nine constructs always contain a body enclosed in brackets:

- *TypeSignature*: *typeSignatureHeader* ‘{’ ... ’}’
- *InterfaceSignature*: *interfaceSignatureHeader* ‘{’ ... ’}’
- *MethodSignature*: *methodSignatureHeader* ‘{’ ... ’}’ (e.g., Figure B.2, selection units #2) | *methodSignatureHeader* ‘;’ (for interface method declaration)
- *ConstructorSignature*: *constructorSignatureHeader* ‘{’ ... ’}’
- *SwitchWrapper*: *switchHeader* ‘{’ ... ’}’
- *TryWrapper*: *tryHeader* ‘{’ ... ’}’
- *CatchWrapper*: *catchHeader* ‘{’ ... ’}’
- *FinallyWrapper*: *finallyHeader* ‘{’ ... ’}’
- *SynchronizedWrapper*: *synchronizedHeader* ‘{’ ... ’}’

B.2 Constructs with a Body

```
1 BroadcastReceiver mUsbReceiver = new BroadcastReceiver() {
2     public void onReceive(Context context, Intent intent) {
3         String action = intent.getAction();
4
5         if (UsbManager.ACTION_USB_ACCESSORY_DETACHED.equals(action)) {
6             UsbAccessory accessory = (UsbAccessory)intent.↵
                getParcelableExtra(UsbManager.EXTRA_ACCESSORY);
7             if (accessory != null) {
8                 // call your method that cleans up and closes ↵
                communication with the accessory
9             }
10        }
11    }
12};
```

Figure B.1: A code fragment from the Android Official Guide demonstrating “Terminating communication with an accessory”

```
1 BroadcastReceiver mUsbReceiver = new BroadcastReceiver() {...};
2 public void onReceive(Context context, Intent intent) {...}
3 String action = intent.getAction();
4 if (UsbManager.ACTION_USB_ACCESSORY_DETACHED.equals(action)) {...}
5 UsbAccessory accessory = (UsbAccessory)intent.getParcelableExtra(↵
    UsbManager.EXTRA_ACCESSORY);
6 if (accessory != null) {...}
7 // call your method that cleans up and closes communication with the ↵
    accessory
```

Figure B.2: Selection units (each unit residing on a separate line) from Figure B.1

The following five constructs have a body that can be a block or a single `SimpleStatement`:

- `IfWrapper`: *ifHeader* (`'{ ... }'` | ...) (e.g., Figure B.2, selection units #4 and #6)
- `ElseWrapper`: *elseHeader* (`'{ ... }'` | ...)
- `ForWrapper`: *forHeader* (`'{ ... }'` | ...)
- `WhileWrapper`: *whileHeader* (`'{ ... }'` | ...)
- `DoWhileWrapper`: *doWhileHeader* (`'{ ... }'` | ...) *doWhileFooter* `';`

B.3 Sub-division Algorithm

```
1 IEditorDescriptor desc = PlatformUI.getWorkbench().  
2   getEditorRegistry().getDefaultEditor(f.getName());
```

Figure B.3: A statement extracted from a code fragment demonstrating “How do I open an editor programmatically”

```
1 IEditorDescriptor desc = ... .getEditorRegistry().getDefaultEditor(...);  
2 PlatformUI.getWorkbench()  
3 f.getName()
```

Figure B.4: Selection units extracted from Figure B.3

B.3 Sub-division Algorithm

We use the following algorithm to break down a selection unit s into smaller selection units in the following conditions.

Sub-dividing a statement with an anonymous class creation: When s contains an anonymous class creation a that starts on the same line as s , the algorithm considers s and a as one unit (e.g., Figure 6.4, line 9). On the other hand, when s contains a that is on a subsequent line, the algorithm considers s as a separate unit from a . For example, for the anonymous class creation that spans lines 10-37 in Figure 6.4, we create a separate selection unit `new Runnable() { ... }`; whose parent is the `SimpleStatement` spanning lines 9-39 in Figure 6.4.

Sub-dividing a statement with a method or constructor call: We create a separate selection unit (`CallUnit` for a method call and `ConstructorUnit` for a constructor call) from s if:

- the call spans more than one line;
- the enclosing statement (if it exists) contains more than two calls; or
- there is an enclosing call.

For the statement in Figure B.3, all four calls in the statement are candidates. We did not create four selection units however, if a candidate begins and ends on the same line (such as the call to `getDefaultEditor` and `getEditorRegistry`). Figure B.4 shows the three selection units we generated from Figure B.3.

Appendix C

Predetermined Scores When There are No Salient Candidates

The default function takes as input a selection unit, determines whether a selection unit contains each of the following constructs, and sums up scores all the applicable constructs. If the sum of is negative, the final score is 0.

```
CatchWrapper -10
ImportDeclaration -10
PackageDeclaration -10
TryWrapper -10
ModifierPrivate -5
ElseWrapper -3
NullLiteral -2
FinallyWrapper 0
Comment 0
Assignment 0
Return 0
BooleanLiteral 0
FloatLiteral 0
IntegerLiteral 0
ExceptionDeclarationContext 0
FieldDeclaration 0
SimpleStatement 0
SynchronizedWrapper 0
ForWrapper 1
IfWrapper 1
ConstructorArray 1
CharacterLiteral 1
ModifierAnnotationOverride 1
```

ModifierProtected 1
ModifierStatic 1
WhileWrapper 1
StringLiteral 2
ModifierFinal 2
LocalVariableDeclarationStatement 2
SwitchLabel 2
SwitchWrapper 2
ModifierPublic 3
ConstructorSignature 5
Extends 5
Implements 5
Constructor 9
IdentifierLiteral 10
CallCount 10
IdentifierMatchedQueryTerm 10
AnonymousClassCreation 10
MethodSignature 10
TypeSignature 10

Appendix D

Changes to the Eclipse Formatter Profile

Eclipse Code Style API [22] allows users to change the code formatter preferences. We control the width of the summary by changing the following entry in the XML file that stores the preferences:

```
<setting id="org.eclipse.jdt.core.formatter.lineSplit" value="50"/>
```

In addition, to create a formatter profile we referred to in Section 6.4.1, we changed the following entries from the default profile:

```
<setting id="org.eclipse.jdt.core.formatter.↵
    insert_new_line_in_empty_annotation_declaration" value="do not insert"↵
/>
<setting id="org.eclipse.jdt.core.formatter.↵
    insert_new_line_after_annotation_on_local_variable" value="do not ↵
    insert"/>
<setting id="org.eclipse.jdt.core.formatter.blank_lines_after_package" ↵
    value="0"/>
<setting id="org.eclipse.jdt.core.formatter.blank_lines_before_new_chunk" ↵
    value="0"/>
<setting id="org.eclipse.jdt.core.formatter.blank_lines_before_member_type↵
    " value="0"/>
<setting id="org.eclipse.jdt.core.formatter.comment.↵
    clear_blank_lines_in_block_comment" value="true"/>
<setting id="org.eclipse.jdt.core.formatter.↵
    insert_new_line_in_empty_type_declaration" value="do not insert"/>
<setting id="org.eclipse.jdt.core.formatter.indentation.size" value="2"/>
<setting id="org.eclipse.jdt.core.formatter.alignment_for_assignment" ↵
    value="16"/>
<setting id="org.eclipse.jdt.core.formatter.tabulation.char" value="space"↵
/>
```

```
<setting id="org.eclipse.jdt.core.formatter.blank_lines_before_method" ↵
    value="0"/>
<setting id="org.eclipse.jdt.core.formatter.↵
    insert_new_line_in_empty_method_body" value="do not insert"/>
<setting id="org.eclipse.jdt.core.formatter.↵
    alignment_for_method_declaration" value="16"/>
<setting id="org.eclipse.jdt.core.formatter.↵
    indent_switchstatements_compare_to_switch" value="true"/>
<setting id="org.eclipse.jdt.core.formatter.↵
    insert_new_line_after_annotation_on_field" value="do not insert"/>
<setting id="org.eclipse.jdt.core.formatter.tabulation.size" value="2"/>
<setting id="org.eclipse.jdt.core.formatter.↵
    insert_new_line_in_empty_enum_constant" value="do not insert"/>
<setting id="org.eclipse.jdt.core.formatter.comment.↵
    clear_blank_lines_in_javadoc_comment" value="true"/>
<setting id="org.eclipse.jdt.core.formatter.↵
    insert_space_after_closing_paren_in_cast" value="do not insert"/>
<setting id="org.eclipse.jdt.core.formatter.↵
    number_of_empty_lines_to_preserve" value="0"/>
<setting id="org.eclipse.jdt.core.formatter.↵
    insert_new_line_after_annotation_on_type" value="do not insert"/>
<setting id="org.eclipse.jdt.core.formatter.↵
    put_empty_statement_on_new_line" value="false"/>
<setting id="org.eclipse.jdt.core.formatter.comment.line_length" value="50↵
    "/>
<setting id="org.eclipse.jdt.core.formatter.↵
    blank_lines_between_import_groups" value="0"/>
<setting id="org.eclipse.jdt.core.formatter.↵
    blank_lines_between_type_declarations" value="0"/>
<setting id="org.eclipse.jdt.core.formatter.blank_lines_before_imports" ↵
    value="0"/>
<setting id="org.eclipse.jdt.core.formatter.↵
    insert_new_line_in_empty_anonymous_type_declaration" value="do not ↵
    insert"/>
<setting id="org.eclipse.jdt.core.formatter.↵
    use_tabs_only_for_leading_indentations" value="true"/>
<setting id="org.eclipse.jdt.core.formatter.↵
    alignment_for_arguments_in_annotation" value="16"/>
<setting id="org.eclipse.jdt.core.formatter.alignment_for_enum_constants" ↵
    value="16"/>
<setting id="org.eclipse.jdt.core.formatter.insert_new_line_in_empty_block↵
    " value="do not insert"/>
<setting id="org.eclipse.jdt.core.formatter.blank_lines_after_imports" ↵
    value="0"/>
```