UNIVERSITY OF CALIFORNIA,
IRVINE

Understanding the Impact of Support for Iteration on Code Search

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Software Engineering

by

Lee Thomas Martie

Dissertation Committee:
Professor André van der Hoek, Chair
Professor Cristina V. Lopes
Associate Professor James A. Jones

2017

ProQuest Number: 10621338

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 10621338

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

© 2017 Lee Thomas Martie

DEDICATION

To my wife,
Hsiao-Hsuan Yu,
who supported me more than I imagined anyone could and anyone deserved.

To my cats,
Fred and Maymay,
who added just the right amount of chaos and companionship to my life at home.

To my parents,
Richard and Lisa Martie,
who let me play and have fun with BASIC when I was a kid and supported my education in programming.

To my brothers,
Phil and Ben Martie,
who laughed at the absurdity of life with me.

To my sister-in-laws,
Jude and Meghan Martie,
for being the sisters I did not have growing up.

To Edward Shafranske,
who taught me courage and helped me grow into a more capable person.

"The art is to always be willing to make the guess, be willing to be completely wrong, be humble to the fact, and be completely haughty to man… Otherwise, you get laughed out of a right idea." — Warren McCulloch

"Always look on the bright side of life." — Monty Python

"Never give up." — Grandma Martie

TABLE OF CONTENTS

LIST OF FIGURES v
LIST OF TABLES viii
LIST OF EQUATIONS x
ACKNOWLEDGMENTS xi
CURRICULUM VITAE xii
ABSTRACT OF THE DISSERTATION xiv
Introduction 1
1.1 Dissertation Structure 18
Background 20
2.1 Empirical Studies on Code Search 20
2.1.1 Surveys 21
2.1.2 User Studies 23
2.1.4 Empirical Study Discussion 29
2.2 Tool Support for Code Search 32
2.2.1 More Expressive Queries 32
2.2.5.3 Improving Ranking with Weighted Matching 40
2.2.14 Tool Support Discussion 54
2.3 Iterative Search in Information Retrieval 58
Research Question 61
CodeExchange 66
4.2 CodeExchange Interface 78
4.3 Query Refinement Recommendations 81
4.4 Critiques 82
4.5 Language Constructs 84
4.6 Query Parts 85
CodeLikeThis 118
5.1 Architecture 127
5.2 Interface 131
5.3 The Ranking Algorithms of CodeLikeThis 132
5.3.1 Similarity Function 133
5.3.2.1 Size of R 140
5.3.2.2 Size of S 146
5.3.2.3 ST Design 147
5.3.2.4 MWL-ST Hybrid Design 147
5.3.2.4.1 Complexity^{MC} Density 150
..... 153
5.2.2.4.2 Using Complexity^{MC} Density in MWL-ST Hybrid 154

Evaluation 184
6.1 Experiment Design 185
6.1.1 Non-Iterative Approaches 185
6.1.2 Participants 187
..... 188
6.1.3 Assignments 188
6.1.4 Search Tasks 191
6.1.5 Survey System 193
6.2 Results 195
6.2.1 Experience Scores 196
6.2.2 Task Times 206
6.2.2.1 Experience and Task Times 212
..... 223
..... 224
6.2.2.2 Number of Queries and Task Times CE versus BL 225
6.2.3 Task Completion as a Success Measure 229
Discussion 267
Conclusion 281

LIST OF FIGURES

Figure 1 Grep in use to find 'x' in C code. 4
Figure 2 Creative Computing Deep Space Game. 5
Figure 3 Creative Computing Heapsort. 6
Figure 4 Bulletin Board System Menu. 7
Figure 5 Game Code Example for BBS. 7
Figure 6 Alien Stacking Game Example. 14
Figure 7 AGORA screenshot. 33
Figure 8 S6. 34
Figure 9 XFinder. 36
Figure 10 Grapacc. 38
Figure 11. Sourcerer. 42
Figure 12. CodeBroker. 44
Figure 13. Refoqus. 45
Figure 14. CodeWeb. 47
Figure 15 Explainer. 49
Figure 16 Codelets. 50
Figure 17 Krugle. 51
Figure 18. CodeFinder. 53
Figure 19. Scatter-Gather. 59
Figure 20. Polya's model of problem solving. 63
Figure 21. CodeExchange Architecture. 72
Figure 22. Mining Architecture. 76
Figure 23. CodeExchange splash screen. 78
Figure 24. Advanced Search. 79
Figure 25. Main Page. 79
Figure 26. Query Refinement Recommendations. 80
Figure 27. Query parts after using recommendations. 82
Figure 28. Increase Complexity. 82
Figure 29. Decrease Complexity. 83
Figure 30. Query parts after using critiques. 84
Figure 31. Refine by getParameter Method Call. 85
Figure 32. Query Parts. 86
Figure 33. Deactivating Query Parts. 87
Figure 34. Top of Size to Number of Classes Graph. 89
Figure 35. Top of Complexity to Number of Classes Graph. 90
Figure 36. Top of Number of Imports to Number of Classes Graph. 91
Figure 37. Tree map of Number of Imports Used by a Class. 92
Figure 38. Visitors Over Time to CodeExchange. 95
Figure 39. Search Patterns from Field Study, Part1 98
Figure 40. Search Patterns from Field Study, Part 2 99
Figure 41. Content of Search Number 7. 100
Figure 42. GitHub's Code Search. 103
Figure 43. Search Patterns in Lab Study - Part 1. 109
Figure 44. Search Patterns in Lab Study - Part 2. 110

Figure 45. Rubik's Cube Documentation Example 119
Figure 46. CodeLikeThis Source Code Space. 123
Figure 47. Path Through Space. 124
Figure 48. CodeLikeThis Architecture. 127
Figure 49. CodeLikeThis Splash Screen. 131
Figure 50. CodeLikeThis Interface. 132
Figure 51. MMR Flow Chart. 138
Figure 52. Average Group Similarity at Increasing Ranks of Top Results. 141
Figure 53. Average Group Similarities for 21 Queries at Different Ranks. 145
Figure 54. ST Algorithm. 147
Figure 55. Examples of not Concise Matrix Multiplication Results. 149
Figure 56. Examples of More Concise Results for Matrix Multiplication. 151
Figure 57. Matrix Multiplication Example with High Complexity. 153
Figure 58. MWL-ST Hybrid Algorithm. 154
Figure 59. Kullback-Leibler Diversity Algorithm. 155
Figure 60. MoreWithLess. 160
Figure 61. Instructions Given to Participants. 161
Figure 62. Survey System Screen Shot. 163
Figure 63. Affinity Diagram on Lab Walls. 168
Figure 64. Code Sorted in Index Relative to Result A. 176
Figure 65. Results Limited by Rank N. 177
Figure 66. Result to Query and Scores. 179
Figure 67. Advanced Query to Match Code in Top N Similar. 181
Figure 68. Like-This Ranking Algorithm Architecture. 183
Figure 69. SearchIt, the Baseline Search Engine. 186
Figure 70. Survey Interface. 194
Figure 71. Experience Scores as Box Plots per Task. 196
Figure 72. Space of Possibilities. 203
Figure 73. Spread of Time till First Paste by Search Engine and Question. 208
Figure 74. Spread of Time till Fourth Paste by Search Engine and Question. 210
Figure 75. Spread of Time till First Paste by Search Engine and Task Type. 211
Figure 76. Spread of Time till Fourth Paste by Search Engine and Find 4 Task Type. 212
Figure 77. Experience and Time to First Paste Correlation by Task Type. 214
Figure 78. Experience and Time to Fourth Paste Correlation. 215
Figure 79. Experience and Time to First Paste Correlation Tasks 1 to 4. 217
Figure 80. Experience and Time to First Paste Correlation Tasks 5 to 8. 218
Figure 81. Experience and Time to Fourth Paste Correlation by Task. 219
Figure 82. Find 4 and Time to First Paste Correlation by Search Engine. 220
Figure 83. Find 4 and Time Fourth Paste Correlation by Search Engine. 221
Figure 84. No Specific Role and Time to First Paste Correlation by Search Engine. 222
Figure 85. Experience and Time to First Paste Correlation by Find 1 and Search Engine. 223
Figure 86. Experience and Time to First Paste Correlation by Algorithm/Data Structure and Search Engine. 224
Figure 87. CodeExchange Time to Query Correlation. 228
Figure 88. Experience by Incomplete Tasks. 230

Figure 89. Mapping of Iterative Feature Usage Frequency to Experience. 234
Figure 90. Uses of Iterative Features to Time Correlation Across All Tasks. 236
Figure 91. Uses of Iterative Features to Time Correlation by Task Type. 237
Figure 92. Average Number of Iterative Features Used to Median Experience for CodeExchange. 239
Figure 93. Mapping of Iterative Feature Usage to Experience. 242
Figure 94. Time to Uses of Iterative Features Correlation Across All Tasks. 244
Figure 95. Uses of Iterative Features to Time Correlation by Task Type. 245
Figure 96. Average Iterative Features Used to Median Experience for CodeLikeThis. 246
Figure 97. Average Iterative Features Used to Median Experience for Google. 253
Figure 98. Average Iterative Features Used to Median Experience for Baseline. 256
Figure 99. Reasons Why Participants Selected Code. 257
Figure 100. Box Plot Summary of End of Experiment Survey. 263
Figure 101. Illustration of Negative Correlation but Positive Experience. 273

LIST OF TABLES

Table 1. Empirical Studies. 30
Table 2. Code Search Engines and Techniques, Part 1. 56
Table 3. Code Search Engines and Techniques, Part 2. 57
Table 4. Description of how Features use Results, Part 1. 74
Table 5. Description of how Features use Results, Part 2. 75
Table 6. User Study Tasks, Part 1. 104
Table 7. User Study Tasks, Part 2. 105
Table 8. Time (minutes) to Complete Tasks by Search Engine. 106
Table 9. Average GitHub Search Behavior. 112
Table 10. Schema and Description of Code Indexed. 129
Table 11. Twenty-One Representative Queries. 143
Table 12. Latin Square Assignment Design 162
Table 13. Participant Demographics. 163
Table 14. Preference Table. 164
Table 15. Win-Loss Table for MWL and MWL-ST Hybrid. 165
Table 16. Preference by Student Status and Query Type. 166
Table 17. Ranking Algorithm to Cluster 170
Table 18. Self-Reported Demographics. 188
Table 19. Task to Search Engine Assignment. 190
Table 20. Task Matrix. 192
Table 21. Experience Medians of Iterative Approaches Compared to Base Line. 200
Table 22. Experience Medians of Iterative Approaches Compared to Google. 201
Table 23. CodeExchange and CodeLikeThis Median Comparison. 202
Table 24. CodeExchange and Google Median Comparison. 204
Table 25. CodeLikeThis and Google Median Comparison. 204
Table 26. Iterative Approaches Median Comparison. 205
Table 27. CodeExchange and BaseLine Comparison. 205
Table 28. CodeLikeThis and Baseline Comparison. 206
Table 29. Mean Seconds Until First Paste. 207
Table 30. Mean Seconds Until Fourth Paste. 209
Table 31. Mean Time till First Paste by Task Type. 210
Table 32. Mean Time till Fourth Paste. 212
Table 33. Average Number of Queries per Task. 226
Table 34. Average Number of Queries per Task Type. 226
Table 35. Task Completion by Task and Search Engine. 229
Table 36. CodeExchange Search Behavior. 232
Table 37. Chi Squared Tests of Iterative Features and Experience. 235
Table 38. CodeLikeThis Feature Usage. 241
Table 39. Iterative Feature Usage by Experience Contingency Tables. 243
Table 40. Google Keywords and Advanced Query Usage. 248
Table 41. Domains Visited by Task - Part 1. 249
Table 42. Domains Visited by Task - Part 2. 250

Table 43. Google Search Behavior. 251
Table 44. Google Search Behavior by Task Type. 251
Table 45. Baseline Search Behavior. 254
Table 46. Baseline Search Behavior by Task Type. 255
Table 47. Within-Subjects CodeExchange Analysis. 260
Table 48. ANOVA Within Results on Time. 261
Table 49. End of Experiment Survey. 262
Table 50. Search Engines by Components. 268

LIST OF EQUATIONS

Equation 1. Similarity Function. 136
Equation 2. Definition of ComplexityMC Density. 150
Equation 3. TF-IDF in Apache Solr. 157

ACKNOWLEDGMENTS

André van der Hoek gets all the appreciation and thanks I can give for his guidance in my pursuit and completion of this thesis. André taught me many lessons during my PhD, but a few stand out as lessons I will remember and carry into the future. He taught me that writing is a process and he helped me discover mine. He taught me every detail matters. Only after thinking, sometimes exhaustively, about every detail, did the "big picture" present itself in a clear way to me. He taught me to be incredibly patient with those learning. In hindsight, André must have had much patience with me, and it is that patience that gave me the time to learn what I needed. He taught me to listen to intuition. Often our "gut reactions" have important ideas behind them. I learned not to dismiss intuition, but to discover and evaluate the ideas behind it. Thank you, André. I will miss our engaging design sessions.

I thank my committee members, Crista Lopes and Jim Jones. Their feedback and support shaped the ideas and experiments in this thesis for the better. I not only thank them for their time, but also for their challenging questions. Often their questions pushed me to work and think even harder than before and forced me to reflect on my research more deeply.

I thank Thomas Kwak. I count myself lucky to have recruited Thomas to our laboratory. It was not only a pleasure to help Thomas with his own research, but Thomas played a vital and crucial role in helping execute the last laboratory experiment in this thesis. Thomas, Apollo has landed.

To my mentors at IBM Watson, Peri Tarr and Harold Ossher, thank you. You not only showed me the reality of working in a major research laboratory, but welcomed me in and treated me as another colleague.

I thank Marian Petre for her insightful and helpful feedback on my work throughout my career as a PhD.

I also want to thank every member of the Software Design and Collaboration Laboratory (SDCL), Mondego Group, and the Spider Lab that I had the privilege to meet. Almost all the members I met challenged me in my research and were always available for creative brain storming.

I also acknowledge the financial support provided by the NSF and IBM.

CURRICULUM VITAE

Lee Thomas Martie

EDUCATION

UNIVERSITY OF CALIFORNIA, IRVINE **Irvine, CA**
PhD Software Engineering, 2017

GEORGIA INSTITUTE OF TECHNOLOGY **Atlanta, GA**
MS Computer Science, 2009

GEORGIA INSTITUTE OF TECHNOLOGY **Atlanta, GA**
BS Computer Science, 2007

WORK EXPERIENCE

Tools for Software Engineers (TSE), Microsoft **Redmond, WA**
Software Development Engineer II *Sep 2017*

Software Design and Collaboration Laboratory **Irvine, CA**
Research Assistant *Aug 2011 – July 2017*

IBM T.J. Watson Research Center **Yorktown Heights, NY**
Intern *Jun 2016 – Aug 2016*

IBM T.J. Watson Research Center **Yorktown Heights, NY**
Intern *Jun 2015 – Aug 2015*

Department of Informatics, University of California, Irvine **Irvine, CA**
Teaching Assistant *Aug 2012 – Jun 2013*

Design and Intelligence Laboratory, Georgia Institute of Technology **Atlanta, GA**
Research Scientist *Sep 2009 – Jun 2011*

School of Mechanical Engineering, Georgia Institute of Technology **Atlanta, GA**
Undergraduate Programmer *Aug 2006 – Dec 2006*

School of Mathematics, Georgia Institute of Technology **Atlanta, GA**
Undergraduate Research Assistant *Jun 2006 – Aug 2006*

REFEREED PAPERS

L. Martie, T. Kwak, and A. van der Hoek, "Understanding the Impact of Support for Iteration on Code Search", Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), 2017, (to appear).

L. Martie, T. D. LaToza, and A. van der Hoek, "CodeExchange: Supporting Reformulation of Internet-Scale Code Queries in Context", Proceedings of the 30th International Conference on Automated Software Engineering (ASE), 2015, pages 24–35.

L. Martie and A. van der Hoek, "Sameness: An Experiment in Code Search", Proceedings of the 12th Working Conference on Mining Software Repositories (MSR), 2015, pages 76–87.

L. Martie and A. van der Hoek, “Context in Code Search”, Proceedings of the 1st International Workshop on Context in Software Development Workshop (CSD), 2014 (4 pages).

L. Martie and A. van der Hoek, “Toward Social-Technical Code Search”, Proceedings of the 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), 2013, pages 101–104.

S. Rugaber, A. Goel, and L. Martie, “GAIA: A CAD Environment for Model-Based Adaptation of Game-Playing Software Agents”, Proceedings of the 11th Annual Conference on Systems Engineering Research (CSER), 2013, pages 29–38.

L. Martie, V. Palepu Krishna, H. Sajnani, C. Lopes, “Trendy bugs: Topic trends in the Android bug reports”, Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR), 2012, pages 120–123.

ABSTRACT OF THE DISSERTATION

Understanding the Impact of Support for Iteration on Code Search

By

Lee Thomas Martie

Doctor of Philosophy in Software Engineering

University of California, Irvine, 2017

Professor André van der Hoek, Chair

Sometimes, when programmers use a search engine, they know more or less what they need. Other times, programmers use the search engine to look around and generate possible ideas for the programming problem on which they are working. The key insight we explore in this dissertation is that, in the latter case, the results found tend to serve as inspiration or triggers for the next queries issued.

We introduce two search engines, CodeExchange and CodeLikeThis, both of which are specifically designed to enable the user to directly leverage results from a previous query in formulating a next query. CodeExchange does this with a set of four features that enable the programmer to use characteristics of the results to find other code with or without those characteristics. For instance, by selecting characteristics of the results the programmer likes (e.g., libraries used or method calls) or dislikes (e.g., code complexity or size), the programmer can refine their query for results with or without those characteristics. CodeLikeThis explores a different mechanism of supporting iteration by letting developers simply select an entire result to find code that is analogous, to some degree, to that result.

For instance, the developer can select an algorithm implementation (e.g., quick sort) with a directive to find more similar implementations, less similar implementations (e.g., heap sort), or somewhat similar implementations.

We evaluated the impact of CodeExchange and CodeLikeThis on the experience, time, and success of the code search process. We compared our iterative approaches with two approaches not explicitly supporting iteration, a baseline and Google, in a laboratory experiment among 24 developers. We found that search engines that support using results to form the next query can improve the programmers' search experience and different approaches to iteration can provide better experiences depending on the kind of task.

The main contributions of this dissertation are six-fold. First, it contributes a new approach to code search, implemented in CodeExchange, that supports the programmer in iteratively searching by bringing characteristics of the results into their query. Second, it contributes, a new approach to code search, implemented in CodeLikeThis, that supports the programmer in iteratively searching by simply selecting a result to issue a query for other similar code. Third, it contributes an extensive laboratory experiment evaluating the impact of iterative approaches on the experience, time, and success of the code search process. Fourth, it contributes new findings about how developers search for code. Fifth, it contributes the implementation of CodeExchange and CodeLikeThis as fully functioning search engines over 10 million Java classes mined off the Internet. Sixth, it contributes an Index of 10 million Java classes indexed by different technical and social properties.

Chapter 1

Introduction

Programming "amounts to determining in advance everything the computer will do" [78]. The process of determining what a computer will do has been likened to the work of an "architect, a composer, or a writer" [32], but where the programmer translates their ideas into source code and iteratively modifies their creation to meet various criteria [32]. Some criteria could be aesthetic (e.g., look and feel of the program [121] or its source code [69]), functional (e.g., what the program does), or non-functional (e.g., how fast the program runs).

The ideas that are programmed depend on the situation. If the programmer is in the position to create software to their liking, then they have full control of what ideas are programmed (e.g., Linus Torvalds and his first release of Linux [124]). Alternatively, the programmer might work in a company that has teams of various sizes addressing a variety of projects, and the programmer's ideas are scoped by the project they work on, modified to

accommodate other team members' ideas, and narrowed by the task to which the programmer is assigned [55]. Or, the programmer might join an open source project, where the project scopes the ideas, but the programmer can freely choose what part of the project they want to enhance or fix [90], [103]. Regardless of situation, and influence of team and project, what a developer programs is never fully specified – they still must decide what to program specifically.

Deciding what to program is often a problem solving process [109]. This entails making choices, such as what the architecture, algorithms, and data structures should be [32]. Different needs can drive these decisions. For example, if the program needs to evolve, then modular approaches for architecture might be better. As another example, depending on the data structures chosen, some algorithms might work better or worse (e.g., Radix Sort can sort integers and strings faster than other algorithms, but it cannot be used for certain other types of data structures).

When programmers engage in problem solving, it has been observed that they might sketch on a white board [72], talk with colleagues [130], or just think. They might sketch an architecture or algorithm to examine it or use it as a focal point of discussion. They might talk with colleagues to get their insight or experience on solving a problem. Or, they might just think about their problem by mentally simulating different solutions, considering alternative implementations, reflecting back on previous solutions they have used, and so on.

It has also been observed that programmers *search* for previously written source code. When programmers search for source code, they look for code by matching it with the problem being faced [27]. Sometimes the code is matched by remembering that it was used to solve the problem in the past [13]. Other times, programmers match source code found by making analogies between the problem the source code solves and the current problem they face [27]. If the source code can be reasonably adapted by the programmer to solve the current problem, then the programmer will often use that source code [27]. This dissertation focuses on the code search activity, and particularly seeks to: (1) understand how developers search and (2) help them in this task.

How programmers have searched for code has evolved with computers and software. In the 1970s, programmers would regularly search using command line tools, such as grep [39], to search through their files for some particular source code [111]. Grep takes a regular expression query to find files that have lines matching the query. Figure 1 presents an example in which a hypothetical programmer searches for C files containing the character 'x', presumably looking for uses of a variable. The results, as a list of files with matching lines of code, are printed below the command.

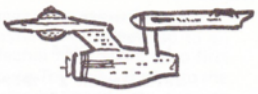
```
$ grep -w x *.c      ...search all files ending in ".c".
a.c:test (int x)
fact2.c:long factorial (x)
fact2.c:int x;
fact2.c:  if ((x == 1) ,, (x == 0))
fact2.c:    result = x * factorial (x-1);
$ grep -wl x *.c     ...list names of matching files.
a.c
fact2.c
$ _
```

Figure 1 Grep in use to find 'x' in C code.

While the use of grep was helpful in many instances, it is limited to searching local code that is authored by the programmer or their team and accessible on the local file system. Searching source code by other authors elsewhere cannot be done with grep, which is why programmers often would subscribe to magazines, such as Creative Computing [2] or Compute! [141], that would regularly publish all sorts of source code, including games, AI algorithms, or graphical programs. Figure 2 presents one page of a space game from the Creative Computing magazine. Programs like this not only illustrate how to make games, but can be modified by the programmer to create games suiting their needs. Figure 3 is another page from Creative Computing, illustrating how to implement Heapsort, where the introduction paragraph explains that the code is intended to support the programmer to both understand the algorithm and implement it. Flipping through magazines typically was a time consuming and laborious process, but programmers needed and wanted code, so they tended to read these magazines "religiously" [104].

Another new game from Creative Computing . . . .

DEEPSPACE



Author: Unknown
Modified by: Bill Cotter, Pittsfield, Mass.
Language: BASIC (Honeywell 600/6000)
Description: DEEPSPCE is another version of a space battle. You become the commander of either a scout ship, cruiser, or battleship. You then pick the weapons, and planetary system to patrol, and it's time to do battle.

The closer you get to the enemy, the better your chance of destroying him. Unfortunately, his chance of destroying you also improves. If you get too close, you can damage yourself; when a vessel's damage rating reaches or exceeds 100, it's destroyed.

Suggestion: Change the time between reports—this will shorten the game by allowing you to get closer faster. Also, for a truly random game, Honeywell users should change RND(0) to RND(—1).

PROGRAM LISTING

```
100 PRINT
110 PRINT
120 PRINT "DEEPSPACE   ";DAT$
130 PRINT
140 PRINT
150 PRINT "THIS IS DEEPSPACE, A TACTICAL SIMULATION OF SHIP-TO SHIP"
160 PRINT "COMBAT IN DEEP SPACE."
170 PRINT "DO YOU WISH INSTRUCTIONS (YES OR NO)";\INPUT I$
180 IF I$="YES" THEN 200
190 GO TO 610
200 PRINT "YOU ARE ONE OF A GROUP OF CAPTAINS ASSIGNED TO PATROL A"
210 PRINT "SECTION OF YOUR STAR EMPIRE'S BORDER AGAINST HOSTILE"
220 PRINT "ALIENS. ALL YOUR ENCOUNTERS HERE WILL BE AGAINST HOSTILE"
230 PRINT "VESSELS. YOU WILL FIRST BE REQUIRED TO SELECT A VESSEL"
240 PRINT "FROM ONE OF THREE TYPES, EACH WITH ITS OWN CHARACTERISTICS:"
250 PRINT
260 PRINT "TYPE","SPEED","CARGO SPACE","PROTECTION"
270 PRINT "1 SCOUT","10X","16","1"
280 PRINT "2 CRUISER","4X","24","2"
290 PRINT "3 BATTLESHIP","2X","30","5"
300 PRINT
310 PRINT "SPEED IS GIVEN RELATIVE TO THE OTHER SHIPS."
320 PRINT "CARGO SPACE IS IN UNITS OF SPACE ABOARD SHIP WHICH CAN BE"
330 PRINT "FILLED WITH WEAPONS."
340 PRINT" PROTECTION IS THE RELATIVE STRENGTH OF THE SHIP'S ARMOR"
350 PRINT "AND FORCE FIELDS"
360 PRINT
370 PRINT "ONCE A SHIP HAS BEEN SELECTED, YOU WILL BE INSTRUCTED TO ARM"
380 PRINT "IT WITH WEAPONRY FROM THE FOLLOWING LIST:"
390 PRINT
400 PRINT "TYPE                          CARGO SPACE    REL. STRENGTH"
410 PRINT "1 PHASER BANKS                   12              4"
420 PRINT "2 ANTI-MATTER MISSILE            4              20"
430 PRINT "3 HYPERSPACE LANCE               4              16"
440 PRINT "4 PHOTON TORPEDO                 2              10"
450 PRINT "5 HYPERON NEUTRALIZATION FIELD   20              6"
460 PRINT
470 PRINT"WEAPONS #1 & #5 CAN BE FIRED 100 TIMES EACH: ALL OTHERS CAN"
480 PRINT "BE FIRED ONCE FOR EACH ON BOARD."
490 PRINT "A TYPICAL LOAD FOR A CRUISER MIGHT CONSIST OF:"
500 PRINT "                    1-#1 PHASER BANK          =12"
510 PRINT "                    2-#3 HYPERSPACE LANCES    =8"
520 PRINT "                    2-#4 PHOTON TORPEDOES     =4"
530 PRINT "                                            ----------"
540 PRINT "                                            24 UNITS OF CARGO"
550 PRINT " A WORD OF CAUTION:FIRING HIGH YIELD WEAPONS AT CLOSE (< 100)"
560 PRINT "RANGE CAN BE DANGEROUS TO YOUR SHIP AND MINIMAL DAMAGE CAN"
570 PRINT "OCCUR AS FAR OUT AS 200 IN SOME CIRCUMSTANCES."
580 PRINT
590 PRINT "RANGE IS GIVEN IN THOUSANDS OF KILOMETERS."
```

LISTING continues on next page.

SAMPLE RUN

```
DEEPSPACE   05/29/75

THIS IS DEEPSPACE, A TACTICAL SIMULATION OF SHIP-TO SHIP
COMBAT IN DEEP SPACE.
DO YOU WISH INSTRUCTIONS (YES OR NO) ?NO
DO YOU WISH A MANEUVER CHART ?YES
                    **************
    MANEUVER CHART

 1      FIRE PHASERS
 2      FIRE ANTI-MATTER MISSILE
 3      FIRE HYPERSPACE LANCE
 4      FIRE PHOTON TORPEDO
 5      ACTIVATE HYPERON NEUTRALIZATION FIELD
 6      SELF-DESTRUCT
 7      CHANGE VELOCITY
 8      DISENGAGE
 9      PROCEED

YOU HAVE A CHOICE OF THREE SYSTEMS TO PATROL.
1 ORION
2 DENEB
3 ARCTURUS
SELECT A SYSTEM(1-3) ?3
WHICH SPACECRAFT WOULD YOU LIKE.(1-3) ?2
YOU HAVE    24 UNITS OF CARGO SPACE TO FILL WITH WEAPONRY.
CHOOSE A WEAPON AND THE AMOUNT YOU WISH. ?1,1
YOU HAVE    12 UNITS OF CARGO SPACE TO FILL WITH WEAPONRY.
CHOOSE A WEAPON AND THE AMOUNT YOU WISH. ?2,1
YOU HAVE     8 UNITS OF CARGO SPACE TO FILL WITH WEAPONRY.
CHOOSE A WEAPON AND THE AMOUNT YOU WISH. ?3,1
YOU HAVE     4 UNITS OF CARGO SPACE TO FILL WITH WEAPONRY.
CHOOSE A WEAPON AND THE AMOUNT YOU WISH. ?4,2

RANGE TO TARGET:   658.3301
RELATIVE VELOCITY:   3.154701
ACTION ?9

RANGE TO TARGET:   599.0996
RELATIVE VELOCITY:   3.154701
ACTION ?9

RANGE TO TARGET:   539.8691
RELATIVE VELOCITY:   3.154701
ACTION ?7
CHANGE TO BE EFFECTED ?2
CHANGE BEYOND MAXIMUM POSSIBLE
INCREASING TO MAXIMUM

RANGE TO TARGET:   460.1757
RELATIVE VELOCITY:    4
ACTION ?9
DAMAGE CONTROL REPORTS YOUR VESSEL DAMAGE AT:   5.329875

RANGE TO TARGET:   380.4822
RELATIVE VELOCITY:    4
ACTION ?1
SCANNERS REPORT ENEMY DAMAGE NOW:   1.260274
DAMAGE CONTROL REPORTS YOUR VESSEL DAMAGE AT:   -0.74178

RANGE TO TARGET:   300.7888
RELATIVE VELOCITY:    4
ACTION ?2
SCANNERS REPORT ENEMY DAMAGE NOW:   40.81256
DAMAGE CONTROL REPORTS YOUR VESSEL DAMAGE AT:   15.35915

RANGE TO TARGET:   221.0953
RELATIVE VELOCITY:    4
ACTION ?1
SCANNERS REPORT ENEMY DAMAGE NOW:   43.63048
DAMAGE CONTROL REPORTS YOUR VESSEL DAMAGE AT:   45.269

RANGE TO TARGET:   141.4019
RELATIVE VELOCITY:    4
ACTION ?3
SCANNERS REPORT ENEMY DAMAGE NOW:   144.1723
ENEMY VESSEL DESTROYED
DAMAGE CONTROL REPORTS YOUR VESSEL DAMAGE AT:   140.0901
YOUR VESSEL HAS BEEN DESTROYED
 ANOTHER BATTLE ?NO
TRY AGAIN LATER!
```

258

Figure 2 Creative Computing Deep Space Game.

Creative Programming Techniques. . . .

Heapsort

Most programming texts present the problem of writing one or two basic types of sort programs. Are these generally used in production? Usually not. One of the most efficient production sort algorithms is known as *Heapsort*. In the richly commented BASIC program below, Geoffrey Chase, OSB, of the Portsmouth Abbey School has written a *Heapsort* routine for both character string or numeric sorting. Look it over. Study how it works. And when you want a really efficient sort routine, use it!

```
NOTES:

(1) EVIDENTLY THIS CAN BE SPLIT INTO TWO PROGRAMS; OR YOU
    CAN CUT OUT THE UNNEEDED HALF.

(2) LINE 120 CAN BE DIMENSIONED AS DESIRED.

(3) THE  !  "TAG" COMMENTS AREN'T NEEDED.  SOME BASICS
    ALLOW  !  , SOME ALLOW  '  INSTEAD, SOME NEITHER.

100 REM.  KNUTH/WILLIAMS/FLOYD  HEAPSORT ALGORITHM.
110 !   PAS '74
120 DIM N(150),C$(150)
130 PRINT
135 PRINT
140 PRINT
145 PRINT "TYPE  C  FOR CHARACTER STRING SORT,"
150 PRINT "TYPE  N  FOR NUMBER SORT. ";
155 INPUT W$
160 N=0                          ! START COUNT=N AT 0
163 PRINT
166 PRINT
170 IF W$="N" THEN 480
180 IF W$<>"C" THEN 140          ! BAD REPLY
190 !-------------------------< CHARACTER SORT: >----
200 GOSUB 720                    ! ASK FOR STOP CODE
210 INPUT S$                     ! GET STOP CODE
215 PRINT
220                              ! INPUT LOOP:
230 N=N+1
235 INPUT C$(N)
240 IF C$(N)<>S$ THEN 230
250                              ! END OF INPUT...
260 N=N-1
265 PRINT
270                              ! HEAPSORT PROPER:
280 L=INT(N/2)+1
285 N1=N                         ! PRESERVE N, USE N1
290 IF L=1 THEN 310
300 L=L-1
303 A$=C$(L)
306 GOTO 350
310 A$=C$(N1)
315 C$(N1)=C$(1)                 ! MOVE TOP OF HEAP TO END
320 N1=N1-1                      ! HEAP IS 1 SMALLER NOW
330 IF N1=1 THEN 440             ! ONLY ONE LEFT? THEN WE'RE DONE.
340                              ! NO, CONTINUE
350 J=L
360 I=J
365 J=2*J                        ! LOOK FOR "SONS" OF I
370 IF J=N1 THEN 400
380 IF J>N1 THEN 420             ! "N1" IS SIZE OF ACTIVE LIST
390 IF C$(J)>=C$(J+1) THEN 400   ! CHOOSE LARGER "SON"
395 J=J+1
400 IF A$>=C$(J) THEN 420
410 C$(I)=C$(J)
415 GOTO 360                     ! LARGER SON REPLACES PARENT
420 C$(I)=A$
425 GOTO 290
430                              ! END OF SORT...
440 C$(1)=A$
450 FOR I=1 TO N
453 PRINT C$(I)                  ! OR REVERSE ORDER: I=N TO 1 STEP -1
456 NEXT I
460 GOTO 130
470 !-------------------------< NUMERIC SORT: >-----
480 GOSUB 720
483 INPUT S
486 PRINT
490 N=N+1
493 INPUT N(N)
496 IF N(N)<>S THEN 490
500 N=N-1
505 PRINT
510     !
520 L=INT(N/2)+1
525 N1=N
530 IF L=1 THEN 550
540 L=L-1
543 A=N(L)
546 GOTO 590
550 A=N(N1)
555 N(N1)=N(1)
560 N1=N1-1
570 IF N1=1 THEN 680
580     !
590 J=L
600 I=J
605 J=2*J
610 IF J=N1 THEN 640
620 IF J>N1 THEN 660
630 IF N(J)<N(J+1) THEN J=J+1   ! FANCY "IF" SYNTAX. COMPARE
640 IF A>N(J) THEN 660                                 390-400.
650 N(I)=N(J)
655 GOTO 600
660 N(I)=A
665 GOTO 530
670     !
680 N(1)=A
690 FOR I=1 TO N
693 PRINT N(I)
696 NEXT I
700 GOTO 130
710 !-------------------------< SUBROUTINE: >-------
720 PRINT "PLEASE INDICATE A STOP CODE--SOMETHING NOT IN YOUR"
730 PRINT "LIST, WHICH WILL ACT AS AN 'END-OF-LIST' SIGNAL: ";
740 RETURN
750     !
760 END
```

```
TYPE  C  FOR CHARACTER STRING SORT,
TYPE  N  FOR NUMBER SORT. ? C

PLEASE INDICATE A STOP CODE--SOMETHING NOT IN YOUR
LIST, WHICH WILL ACT AS AN 'END-OF-LIST' SIGNAL: ? KNUTH

? DAVID AHL, ESQ.
? COSMO COMPUTERS
? ABPLANALP LTD.
? PETRODOLLARS
? DMA TRANSFER
? CREATIVE COMP.
? M.O.S. ABACUS
? ALGORITHMS
? LEONARDO P.
? CHINESE REMS.
? SORTED STRINGS
? NEG. FULLBACK
? STAR TREK, V.2
? KNUTH

ABPLANALP LTD.
ALGORITHMS
CHINESE REMS.
COSMO COMPUTERS
CREATIVE COMP.
DAVID AHL
DMA TRANSFER
LEONARDO P.
M.O.S. ABACUS
NEG. FULLBACK
PETRODOLLARS
SORTED STRINGS
STAR TREK

TYPE  C  FOR CHARACTER STRING SORT,
TYPE  N  FOR NUMBER SORT. ? N

PLEASE INDICATE A STOP CODE--SOMETHING NOT IN YOUR
LIST, WHICH WILL ACT AS AN 'END-OF-LIST' SIGNAL: ? -1E6

? 3.1416
? 22222
? 2E10
? 2E-10
? 66.666
? -1E5
? -1E6

-100000
 2.00000E-10
 3.1416
 66.666
 22222
 2.00000E+10

TYPE  C  FOR CHARACTER STRING SORT,
TYPE  N  FOR NUMBER SORT. ?
STOP AT LINE 155
READY
```

233

Figure 3 Creative Computing Heapsort.

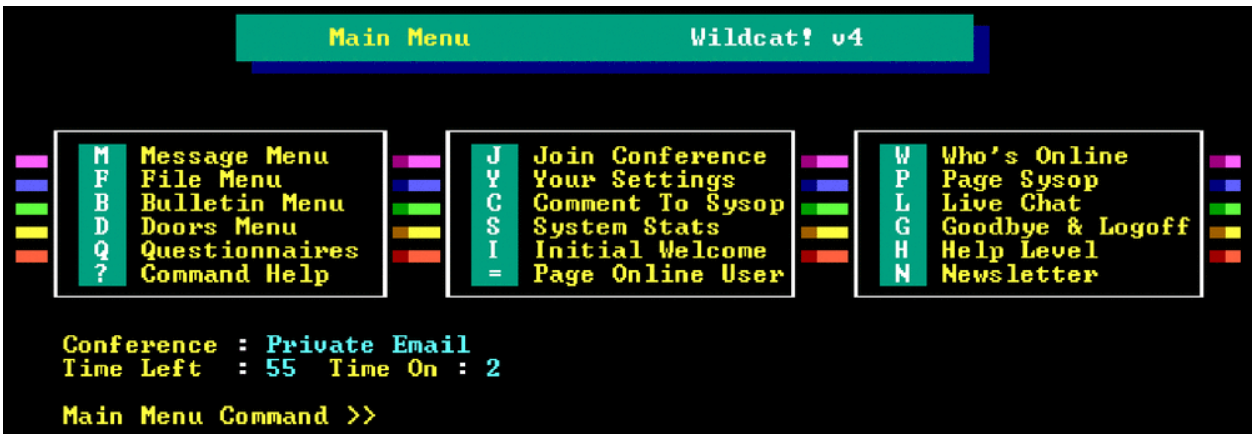


Figure 4 Bulletin Board System Menu.

With new technology, finding code became easier. In the 1980s, as modems became cheaper, programmers began to distribute code over networks. One particular method was sharing code over bulletin board systems (BBS). Programmers would use a modem to dial into a BBS (a telnet server) and be presented with a splash screen as shown in Figure 4. The programmer could then use the file menu to browse through the different source code examples made available. Interestingly, often the code found on a BBS was source code that could be used to extend the BBS itself. Figure 5 shows the top of some example code [138] in C, distributed on a BBS from 1980.

```
/* EXAMPLE GAME FORMAT FOR THE GATEWAYS PROJECT - A SIMPLE GUESSING GAME */
/* Designed and written by David M. Larson, BBS: 916-753-8788 */
/* Dynasoft, P.O. Box 915, Davis, CA 95617 */
#include "GATEWAYS.H"
#include "D0MAIN.H"

/*-----------------------------------------------------------*/
void d0read_d0questions()
{
d0quest_struct *ptr;
memset(d0questions,0,MAX_QUEST*sizeof(*d0questions));
gfd0=og_open("D0QUEST",0);
if(gfd0==-1) return; /* file didn't exist */
```

Figure 5 Game Code Example for BBS.

Companies also began developing support for searching for code internal to the company. AT&T Bell Labs, for example, developed a system called CATALOG [35], which supported the programmer in issuing keywords that it would then use to match code indexed under those words. Another example by AT&T Bell Labs was a system called the Modeling Expert System (MES) [93], that would take in a series of constraints for transmissions equipment to find source code used in equipment matching these constraints. For example, the programmer could specify dimensions and hardware components of transmissions equipment.

In the 1990s, the amount of source code to search began growing considerably because of the rapid growth of the Internet [59]. It was not long before programmers saw the advantages of setting up their own servers to support sharing source code. As early as 1991, for example, the source code for Linux was being distributed on the Internet through a simple file transfer protocol (FTP) server [79]. In 1999, SourceForge [162] released the first server aimed at supporting collaborative development of source code for anyone to read, use, or add to (referred to today as open source [103]). SourceForge freely gave away storage for code by providing an online repository with version control and issue tracking tools to use. Within a year after 1999, a few thousand software projects were on SourceForge, a number that jumped to 30,000 in 2001 and then jumped again to 150,000 in 2007 [167].

Today, the amount of source code on the Internet has grown by at least a factor of 100 since 2007, if not more, from software projects in the hundreds of thousands to the tens of millions [1] and appears to be growing exponentially [26]. GitHub, which today hosts the largest

online collection of projects and their source code, was estimated in 2013 to have over 10 million projects [1]. Other online repositories exist as well [139], [160], [168]. Further, source code now also appears in a variety of contexts outside of dedicated online repositories. On question and answer forums, such as StackOverflow [163], individual questions or answers frequently contain some lines of code to introduce a problem or illustrate how to solve the problem. Tutorial web pages can be found with source code, such as code illustrating how to use a library for domain specific applications (e.g., sample code using Eclipse libraries [143] to add features to the Eclipse integrated development environment [127]), or implementing a fundamental algorithm (e.g., the A* algorithm [96] for basic path finding for robot navigation [151]).

Finding particular source code across this vast amount of possibilities is commonly done with search engines – servers on the Internet that have indexed content elsewhere (e.g., web pages, online repositories, or other kinds of files). Two types of search engines have emerged that support searching for source code, web search engines and code search engines. Web search engines, with Alta Vista and Infoseek being among the first and Google being the most popular today [161], take keywords and then return a list of links to any web page matching the keywords. Code search engines focus the search on source code only and will commonly return a list of links to code matching the keyword query. Most code search engines focus on indexing code in online repositories, such as GitHub or SourceForge. Some exceptions exist, however, with Example Overflow [9] indexing code mined from StackOverflow posts.

Various tradeoffs exist between using web search engines versus code search engines. Web search engines have a wide range of content that they have indexed, so they can return results that include code posted on tutorial pages that have illustrative pictures, or code on question and answer forums that have descriptive comments. However, web search engines also might match off topic files having nothing to do with code or only containing a few lines of code that illustrate a point, but are otherwise incomplete and thus not useful for some situations. Code search engines have the advantage of searching over only source code, often in actual projects, and can leverage this fact to support different kinds of filters, such as filtering results by programming language or project [144]. The disadvantage of code search engines is that they may return results that have few comments or are hard to read, making the results difficult to comprehend and apply [16]. Both web search engines and code search engines share the disadvantage that they may return code of low quality; often any and all code found is indexed, but that does not guarantee the code is bug free, performant, easily maintainable, and so on.

Given the advantages and disadvantages of search engines today and the importance of search in developing software (programmers report searching for code frequently as part of their practice [97], [108]), software engineering researchers are investigating how to improve code search engines. Some, for instance, have been investigating how to support more expressive queries (e.g., searching by test case or method signatures) that afford more precise matching of code compared to keywords (e.g., [4], [15], [54], [65], [71], [82], [91], [110], [122], [142]). Others have investigated new matching and ranking algorithms (e.g., ranking code higher with method names or class names matching the keywords) so that

more results presumed to better match the topic described by the keywords are returned and appear towards the top of the list (e.g., [12], [28], [40], [44], [51], [66], [67], [76], [131]). Both more expressive queries and improved matching and ranking algorithms have shown that explicitly leveraging certain proprieties of source code can improve the performance of search engines compared to approaches that treat code as any other content (including web search engines like Google). For instance, the Specificity ranking algorithm [67] ranks code higher that has names (e.g., class name) matching the keywords and was reported to provide a better ranking than Google. As another example, S6 [91] takes test cases as input and transforms the structural properties of the code indexed so that the code can be run against the test case – enabling a type of search very different from what is possible today.

While many different approaches to improving code search exist, these approaches are generally similar in one very visible design decision: they are non-iterative approaches. They expect a query and optimize on returning the best matching results for the query, occasionally offering filters to help scope the results (e.g., programming language or file type filters) [144], [158]. This focus on a non-iterative design for search engines is mirrored in how search engines are evaluated [152]. Typically, a group of experts score the performance of search engines by the results returned for some representative set of queries, with the score reflecting how on topic the results are.

If we are to imagine the most ideal search engine, in terms of effort to find code, a non-iterative approach is appealing, because it offloads all work to the search engine, other than the need for a programmer to actually issue a query (presumably the programmer's mind

cannot be read). In this ideal world, indeed, the search engine matching algorithm returns exactly what the programmer needs after the first query, every time. A software tool doing all the work is what many imagine as the goal for software tools in general, and this view is increasingly being adopted as machine learning, a non-iterative approach using AI, becomes ever more popular [58]. Recently, for instance, Google has moved away from its original PageRank [85] algorithm to a machine learning algorithm, RankBrain [147], to rank search results. As another example, Sonia Haiduc *et al.* are working to incorporate machine learning in code search engines [44].

While a search engine that returns the results the programmer is looking for after the first query is ideal, the reality today is that search is *iterative*. Often, programmers do not, and cannot, search for code with a single query. Instead, they issue multiple queries [7], [13], [50], [106], [115], where, after receiving results, the programmer modifies their query by removing keywords, adding keywords, or some combination of both, and repeats this process multiple times [7], [50], [106]. That is, search looks like the programmer submitting a query, getting results, submitting a modified query, getting new results, and so on, until the programmer stops searching because they find what they were looking for or give up.

The mismatch between the framing of an ideal "one query for success" search taken by non-iterative approaches and the iterative behavior observed during empirical studies has two possible explanations. First, one could argue (and many do) that the non-iterative approaches to code search need more work, because the programmer is having to submit multiple queries to get the code they are looking for. Second, one could argue that, perhaps,

the idealistic non-iterative approach to search is wrong, and that search for code is inherently an iterative process between the programmer and the search engine.

Cognitive processes in which programmers engage explain why the second answer – code search is iterative – is preferred. Particularly, unlike when people search to be aware of the latest news, a distinctly informational activity [94], programmers search for source code when they are *problem solving*, where what they are looking for, a solution, is not immediately understood [24] and it is not clear how to find what they are looking for [24]. As the programmer begins to understand their current problem, they begin to consider solutions, but these solutions can, in turn, make the programmer rethink the problem by presenting constraints or different perspectives that the programmer did not previously consider [27]. Rethinking the problem will push the programmer to consider alternative representations of the problem or to further define the problem, which changes the solutions they next consider[24], [70], [81], [109]. That is, possible solutions often change the understanding of the problem and in turn change the next solutions considered. The implication, then, is that when programmers receive code results from a search engine, these results will sometimes cause them to rethink the problem they are trying to solve and they will issue a modified query driven by their new understanding of the problem – making the search iterative. For example, let us consider the following hypothetical scenario. Suppose a programmer needs to implement a general purpose algorithm to stack N aliens for a new game (one instance is illustrated in Figure 6). The goal of the game is to stack all the aliens onto the rightmost column, where an alien can only be stacked on top of an alien with more eyes. The programmer searches for code they could possibly reuse to solve this problem.

Issuing the keywords *stacking aliens game* to her favorite search engine yields no results. Upon reflection, she generalizes the search, as often happens [106], to find a general solution to any stacking game that she can adapt and use for her alien game. With a new goal in mind, she next issues the keywords *stacking game*. After looking through the results, she is unable to find any general purpose stacking game algorithm, but she does find a Towers of Hanoi [30] implementation. She recalls from her undergraduate education that Towers of Hanoi is a stacking problem, where the goal is to stack disks onto the right most pole and that disks can only be stacked on top of larger ones. She makes the analogy, as programmers often do for reuse [27], [120], between the number of alien eyes and the size of the disk and immediately understands the problem she is working on as the Towers of Hanoi problem; one she understands well. The programmer's next keywords are *Towers of Hanoi*, and she then begins searching and further refining her query to find an implementation that would be easy to adapt to her stacking alien problem.

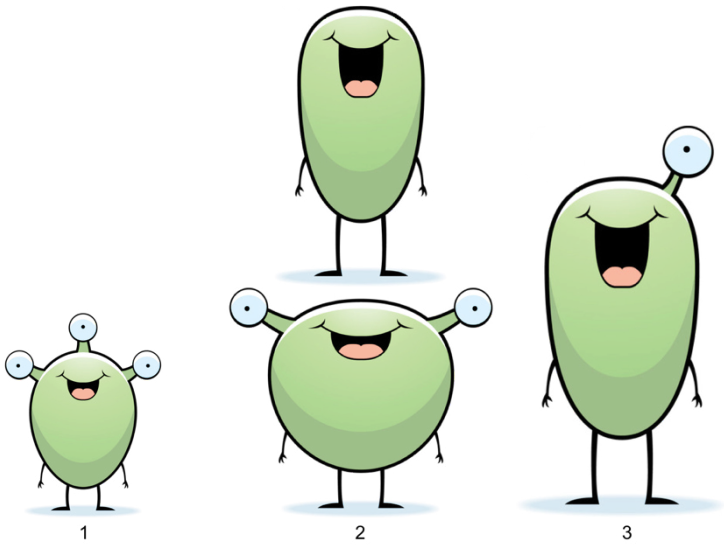


Figure 6 Alien Stacking Game Example.

This dissertation investigates what happens when programmers are explicitly supported in searching iteratively for code. It particularly answers the following research question:

> *What is the impact of explicitly supporting software developers in searching iteratively on the experience, time, and success of the code search process?*

To answer this question, I designed and developed two new approaches to code search: (1) CodeExchange and (2) CodeLikeThis. The insight behind both approaches is that the next query is not typically created at random, but formulated in response to aspects of the current results. For instance, as discussed above, it has been observed that certain results can give the programmer different ideas or constraints on the solution that they did not previously consider [27], which can drive the programmer to issue a modified query trying to find more code with those constraints [24], [70], [81], [109] (e.g., trying to use code from a result as keywords in the next query [50]). Alternatively, the programmer may discover something unfavorable about the solutions given in a set of results (e.g., discovering the results all use an incompatible library) and attempt a different query [7], [106] to find solutions without those unfavorable characteristics. Our work generalizes these individual situations and makes iteration primary in the design of both CodeExchange and CodeLikeThis.

The designs of CodeExchange and CodeLikeThis offer two alternative approaches to explicitly support iteration. CodeExchange gives the programmer manual control to incrementally modify the query for code that is (dis)similar to the current results in two ways: (1) it supports specializing the query by selecting aspects of the results that are similar/dissimilar to what the programmer has in mind, and (2) it decomposes the query

into parts that can be individually toggled to generalize or specialize the query. In this way, by selecting aspects of the results the programmer likes (e.g., libraries used or method calls) or dislikes (e.g., code complexity or size) the programmer can refine their query for results with or without those aspects. Alternatively, by toggling parts of the query on/off the programmer can quickly try different combinations of various aspects of their query in response to the results.

CodeLikeThis, taking somewhat of an opposite approach, offers automatic support for iteration in two ways: (1) the result set from the first keyword query is always diverse, to provide the developer with a variety of possible starting points, and (2) in subsequent queries, the developer searches by selecting one of the code results as a directive that guides whether the next code results are more similar, somewhat similar, or less similar to that result. In this way, the programmer can search by recognizing if a code result is (dis)similar to what they have in mind without needing to specify how. For example, after issuing the query *quick sort*, some results might only call quick sort methods, others might offer code that tests quick sort methods, and yet others might be quick sort implementations. If the programmer was looking for implementations, then she can select one of the implementations to find more quick sort implementations, but she does not need to construct or think of a query to only get implementations or code more like a specific implementation.

We evaluated CodeExchange and CodeLikeThis in a user study among 24 developers with an average of approximately 4 years of professional development experience, where we

compared CodeExchange, CodeLikeThis, Google, and a baseline search engine (CodeExchange minus iterative features). In a counterbalanced design, each participant was assigned two of the four code search engines and used them to complete eight different code search tasks, alternating the search engine used with each task. The search tasks covered a range from those that are more open ended to those that are more well defined, in order to represent a variety of real-world internet-scale code search scenarios [7], [48], [50], [76], [107], [116]. After each search task, each participant was asked to rate their experience in using the assigned search engine for that task. At the end, each participant first filled out a questionnaire about the two search engines they used and finally was interviewed.

To understand the impact of explicitly supporting iteration, we analyzed the collected data for the experience ratings, task times, and task completion rates. From this analysis, three primary results emerged:

- *Iterative search improves code search for some tasks.* Compared to the baseline search engine, we found that there existed an iterative approach providing better experiences in 6/8 tasks and equal experiences in 2/8 tasks. Compared to Google, we found that there existed an iterative approach providing better experiences in 3/8 tasks, equal experiences in 2/8 tasks, and worse experiences in 3/8 tasks.
- *Some kinds of search tasks are better suited for CodeExchange and others for CodeLikeThis.* We found the incremental approach to iteration supported by CodeExchange gave a better experience for searching when the search task was more open ended. However, we also found the automatic approach to iteration supported

by CodeLikeThis gave a better experience for searching when the search task was more defined.

- *Regardless of search engine, participants tried to iteratively search for code.* Across all search engines, users submitted multiple queries per search task and specialized and generalized the current query to create the next query.

1.1 Dissertation Structure

This dissertation is organized into 8 chapters. The remaining chapters are structured as follows:

Chapter 2 – Background

This chapter presents an overview of empirical studies in the literature studying code search, the state of the art in code search engines, and relevant background in information retrieval.

Chapter 3 – Research Question

This chapter first motivates and then introduces the research question investigated in this dissertation.

Chapter 4 – CodeExchange

This chapter presents our first new approach designed explicitly to support iterative code search, as embedded in our prototype search tool CodeExchange. This chapter also presents results of a field study and user study conducted as a preliminary study evaluating

CodeExchange and GitHub.

Chapter 5 – CodeLikeThis

This chapter discusses CodeLikeThis, our prototype search tool that embodies our second new approach designed explicitly to support iterative code search. This chapter also presents the results of a user study to preliminary evaluate CodeLikeThis' diversity ranking algorithm for keywords.

Chapter 6 – Experiment Design and Results

This chapter presents key decisions in the experiment design though which we assessed both CodeExchange and CodeLikeThis more comprehensively. This chapter presents the experiment setup, the rationale of the design, and dependent variables we explore. The experiment design is then followed by the results of the experiment and our analysis of the collected data to understand the impact of explicitly supporting software developers in searching iteratively on the experience, time, and success of the code search process.

Chapter 7 – Discussion

This chapter takes a step back and presents the lessons learned from our study. It particularly discusses the high-level takeaways, key individual results, and overall implications for the design of iterative code search engines.

Chapter 8 – Conclusion

In this section, we summarize the dissertation and discuss our future work.

Chapter 2

Background

Previous research in code search can be divided into empirical studies that capture various aspects of the process by which the programmers search and into research design and development that seeks to provide new ways of supporting code search. In this section, we present a comprehensive summary of both groups of research and at the end of each major subsection present tables providing a high-level overview of this work. We conclude with a brief summary of relevant work in information retrieval.

2.1 Empirical Studies on Code Search

Several types of studies (e.g., surveys, search log analysis, field studies, and lab studies) have been conducted to understand *why* and *how* developers search for code. In the below subsections, we summarize relevant results from these studies, as organized by type of study.

2.1.1 Surveys

Surveys asking programmers why they search for code on the Internet have been conducted by Sim *et al*. [107], Sadowski *et al*. [97], Stolee *et al.* [116], and Hucka and Graham [53]. In Sim *et al*.'s survey, all 69 participants reported that they searched on the Internet either for code to use as a reference or for code that they can use as-is. With respect to code as a reference, programmers reported that they needed reference examples to check if their implementations were correct, learn how to use an application programming interface (API), learn how to use different programming language constructs, or learn a programming concept (e.g., threading). With respect to searching for code as-is, programmers searched for reusable code they could copy and paste or use as a library in an existing project or to use as a starting point for a data structure or algorithm.

Stolee *et al.* surveyed 99 people with programming experience about why they search on the Internet for code. Their participants most often reported that they search to get ideas for programming (71% of all responses included this as a reason), but also frequently reported that they search to find code to adapt to their current project. A few programmers reported using code as-is, without any modification. These results confirm the findings of Sim *et al.* that developers search for code as a reference and to use as-is, but offers different nuances as to why. In particular, the study finds that programmers most often search to get ideas or to rewrite/adapt into their projects rather than to use as-is.

Sadowski *et al.* asked programmers at Google why they searched Google's internal code repository. The survey's questions cover local code search, as opposed to Internet-scale code search discussed in this dissertation, but, since this is a large, company wide repository, it also includes code that programmers have no affiliation with or knowledge of, making the setting at least somewhat similar to Internet-scale code search. In this survey, programmers reported in 60% of responses that they searched to learn how to accomplish a programming task and to explore code to read for a better understanding. Other responses were geared toward local search reasons, such as finding possible places where a planned or ongoing change may have impact, locating where certain code is implemented, identifying who made certain changes, or figuring out when some changes were made. In relation to Stolee et al.'s and Sim et al.'s survey, programmers also reported searching to learn, but contrasting with these studies, other than the explicit local search motivations, was a more specific motivation to search for a better understanding by exploring code. For example, participants reported to search to read code without a specific goal, understand how a function is implemented, or check how a coding task is usually completed.

Lastly, Hucka and Graham [53] surveyed scientists across a variety of fields who also do programming for their research. As in previous surveys, it was found that the scientists looked for example code for implementations (e.g., algorithms or data structures), to remember how to program some function, learn new concepts, or to discover new algorithms. However, contrasting with Sim *et al.*'s [107] survey, Hucka and Graham found that the participants often reported looking for code to reuse as-is. Also, different than other

surveys, the participants reported to search to find more efficient code than what they had written.

Overall, these survey studies provide an important understanding of the motivations behind searching for code, documenting the motivations for code search that guide the design of code search engines. These motivations include getting ideas/inspiration, learning, remembering, clarifying knowledge, and finding code to reuse as-is. Researchers of future code search engines can use these motivations to design and evaluate their tools. For instance, researchers could test how much a programmer learned or the number of ideas a programmer can generate using different kinds of code search engines.

2.1.2 User Studies

Three user studies have been conducted that look at how programmers search for code. Scott Henninger [48], in the first user study of code search, conducted a user study among nine students with programming experience to evaluate code search tasks using three different small scale code search engines (each indexing 1800 Lisp functions): CodeFinder, Helgon+, and Document Examiner. CodeFinder and Helgon+ both provided explicit tool support for the programmer to refine their query, while Document Examiner served as a baseline search engine offering no such refinement support. This study revealed that, on average, the users submitted 7.25 queries and specialized (also known as refined) their query by adding keywords.

In an experiment by Sim *et al.* [106], 24 programmers were given programming tasks to find reference examples as functions/classes or to find subsystems of code to reuse as-is (e.g., libraries). Each task was performed with one of five search engines: Google [145], Krugle [149], Koders (no longer available), Google Code Search (no longer available), and Sourceforge [162]. The authors found that looking for references usually took more queries, more query terms, more modifications, and more time than searching for subsystems. In general, depending on the search engine used, the average number of terms in a query ranged from 3.8 to 4.7 and the average number of queries was 2.38. Frequently, to create new queries, current queries were modified by generalizing (e.g., removing keywords or a filter), specializing (e.g., adding keywords or a filter), or a combination of generalizing and specializing (e.g., removing and adding keywords and filters but still preserving some terms from the previous query). Less common were modifications that changed all keywords in the query or corrected spelling mistakes. These results build on and confirm Henninger's study showing that the next query is often created by specialization, but also showing that it can be created by generalization or a combination of generalization and specialization. Further, this study shows that searching for code as a reference takes more time and iterations that searching for code to reuse as-is.

Joel Brandt *et al.* [13] conducted a user study among 20 students with programming experience. The participants needed to complete programming tasks inside an incomplete preexisting project, using the Internet as an aid. In general, they found that the participants often used the Internet to find code to learn how to use an API, clarify some existing

knowledge to accomplish a task, or as an external memory aid where they looked up code to remember how to accomplish a task. Further, they found that, when programmers needed to learn how to accomplish a programming task, they would learn not just by reading, but also by pasting the code into their editors, modifying it, and running it. Similar to the findings from the survey studies, these participants reported to search for code as a reference from which to learn, clarify knowledge, and remember to successfully complete their programming tasks.

Overall, the studies by Henninger, Sim *et al.*, and Brandt *et al.*, provide insight into the fundamental behavior of code search. Their findings suggest that searching for code is highly iterative, spanning a sequence of queries, where each new query is often a modification of the previous. Researchers can use this insight to design and evaluate search engines explicitly supporting iteration (as done in this dissertation).

2.1.2 Log Analysis

Three studies examined patterns in search engine logs to understand how programmers search for code, the kinds of queries they issue, and the kinds of code that they look for. Bajracharya *et al.* [7] analyzed the logs from the commercial code search engine Koders (unfortunately, no longer available) over a year time span. The analysis of the keywords in the logs revealed that programmers most often searched for domain or application specific code (e.g., mobile games or calendar scheduling), algorithm and data structure code (e.g., traversing graphs or B tree data structures), and framework code (e.g., examples of code using Java or Eclipse libraries). Further, the analysis showed that, overall, programmers looked for code with keywords comprised of just one term, with an average of just 1.31

terms. Further, the analysis revealed that programmers would often submit more than one query during their search (2.16 queries on average), where queries were modified by totally changing the keywords, specializing them, generalizing them, or using advanced query options. However, contrasting with Sim *et. al.*'s user study [106], which showed that programmers most often specialize and generalize their queries, Bajracharya *et al.* found programmers most often totally changed their keywords. There currently is no understanding as to why there is a difference, but it could be an artifact of the methodologies of either study. Using logs to study code search necessarily means encountering more noise and having less control as to why users are searching, but the data is collected from a real world setting. The user study performed by Sim *et al.* has the benefit of controlling the noise and only looking at results from code search tasks, but the data is collected from a less realistic lab setting.

Brandt *et al.* [13] analyzed query logs during July 2008 from the Adobe Developer Network website explicitly about the Adobe Flex Web framework. They sampled 300 from about 70,000 search sessions (defined as periods of queries and clicks no more than 6 minutes apart) and hand coded them into categories. The researchers found that most queries were about reminding oneself of how to use the framework (78%) and the rest about learning new concepts (22%). Further, it was found that learning sessions started with natural language queries in about 50% of the cases, but reminding sessions usually started with code specific terms (75% of cases). They found that, across sessions, about 1.45 queries are issued, where the next query is created by modifying the current query though generalization, specialization, a combination of generalization and specialization, creating a totally new

query, or correcting a spelling mistake. Brandt *et al.* note that the number of queries is lower than other studies, which might be due to its particular focus.

Lastly, Holmes [50] analyzed the search logs from nearly 100 users of the Strathcona search engine in order to characterize code search behavior. Strathcona is not a keyword query search engine, but rather supports the programmer to select parts of code in the programmer's development environment to serve as the context in which Strathcona will automatically create a query and return matching code. Holmes found that developers often selected many different parts of their local project (e.g., classes or methods) of the code in their project for Strathcona to automatically construct a query from. Further, Holmes found that programmers, after viewing search results that were returned, would refine their selection with additional code in their project (2.5 queries were issued on average during a search session). Holmes speculated that developers might be gleaning some information from the results that inform them how to modify their selection. Similar to previous studies discussed, Holmes also finds that programmers submit multiple queries to search for code, even when not using keywords. Further, Holmes finds the queries to be heterogeneous in nature, since queries often were composed from different parts of the programmer's project.

Overall, the studies analyzing the log files of different search engines confirm some of the findings from the surveys and lab studies. Programmers submit multiple queries and some of their motivations include learning new concepts or being reminded how to accomplish a programming task. However, the studies also provide important nuanced findings, such as Holmes' results showing that queries are heterogeneous. We also see a difference in how

queries are usually modified between Bajracharya et al. and Sim et al.'s user study [106], suggesting further clarifying research needs to be done.

2.1.3 Field Studies

Rosalva Gallardo-Valencia *et al.* [37] conducted a field study onsite at a company in Peru where they observed employees search for code and had them write down their queries and their goals for their search. The researchers found that:

- 38% of searches were concerned with a need to learn how to figure out how to complete a task, use code they already had, or to configure code they had;
- 24% of searches were concerned with a need to remember how parts of some code work or the meaning of parameters, flags, or what a method does;
- 14% of searches were concerned with a need to gain a deeper understanding of the code so that they could write better documentation or make modifications to it;
- 11% of searches were concerned with a need to solve a bug;
- 8% of searches were concerned with a need for libraries translating English to Spanish; and
- 5% of searches were concerned with finding and comparing multiple candidate software components to use in a project.

The results of this study provide important results from a realistic setting that confirm the results in lab studies and surveys that show motivations to find code include learning, remembering, to clarify/gain a deeper understanding, and as method to fix a bug.

2.1.4 Empirical Study Discussion

The empirical studies on code search indicate that programmers search iteratively and search for many reasons. Emphasizing this point, we present in Table 1 a high-level summary of the findings from the empirical studies. The Behavior, Query Types, and Motivations column headers organize findings from all empirical studies discussed and each row references specific studies by author name. Cells with a value of "✓" map studies to the corresponding phenomenon found and cells with decimals map studies to corresponding averages found. For example, the first row shows Sim *et al.* found the motivations to search for code are to find code to support cognitive processes of the programmer or to use as-is. Further, these authors found that some of these cognitive processes involve learning, remembering, or getting clarification. As another example, Holmes found that the average number of queries submitted during search is 2.5.

Of importance to this dissertation are the following four observations.

- First, it is clear that programmers search for code iteratively. That is, they submit an initial query, receive results, submit a modified query, receive a new set of results, and so on.
    - This behavior suggests the opportunity to better support search by designing features that explicitly support the programmer to iteratively search for code.
- Second, creating a new query often means modifying the current query by generalizing it, specializing it, or some combination of both. Further, the terms added/removed to/from a query are either natural language terms and/or code specific terms.

- This behavior suggests that how iteration is supported is important and suggests the opportunity to better support iteration with features that specialize and generalize the query with code specific and/or natural language terms.

Table 1. Empirical Studies.

Code Search Studies																	
		Behavior						Query Types		Motivations							
		Counts		Iterating with Keywords						Local Code Search			Purpose				
													As-is	Cognitive			
Authors	Study Type	Avg. Number of Terms in Query	Avg. Number Of Queries	Specialize Queries	Generalize Queries	Combination of Specialize & Generalize	Totally Different Queries	Natural Language as Keywords	Code Terms as Keywords	Who/When	Impact	Change	Copy & Paste Code or Find Project	Learn How	Recall/Reminding	Get Ideas	Clarify/Exploring/Learn for Detailed Understanding
Sim *et al.* [107]	Survey												✓	✓	✓		✓
Hucka *et al.* [53]	Survey												✓	✓	✓		✓
K. Stolee [116]	Survey												✓			✓	
R. Gallardo-Valencia *et al.* [37]	Field Study													✓	✓		✓
Sim *et al.* [106]	User Study	3.8-4.7	2.38	✓	✓	✓	✓										
S. Henninger [48]	User Study		7.25	✓													
Joel Brandt *et al.* [13]	User Study and Log Analysis		1.45	✓	✓	✓	✓	✓	✓					✓	✓		✓
Sadowski *et al.* [97]	Survey and Log Analysis	1.85	2.69	✓						✓	✓	✓		✓			✓
Holmes [50]	Log Analysis		2.5	✓													
Bajracharya *et al.* [7]	Log Analysis	1.31	2.16	✓	✓	✓	✓	✓	✓								

Sadowski *et al.* studied Google repository search

✓ = phenomenon found

empty = Not studied

- Third, the goal of code search is often not predetermined. Code search frequently serves to support cognitive processes, such as learning, remembering, clarifying, and generating ideas, as they pertain to programming – a problem solving driven task [109]. Problem solving and cognitive processes often do not have clear goals initially, but become clearer over time, which implies the goal driving the search can be initially unclear and changing [24], [27], [120].
    - Unclear and changing goals driving search has implications on how code search engines are evaluated. In particular, it implies that classical approaches to measuring performance in information retrieval that use predetermined results that a ranking algorithm should return or a user should find may not measure success or performance, because programmers often are changing their goals for their search. This implication suggests a more programmer centric study might be more appropriate, where the study lets the programmer discover or decide what code they are looking for and collects and analyzes user reported measures about the search process.
- Lastly, because Sadowski *et al*. studied search in a company repository, some motivations to find code are to understand the impact of making a change, what code has been changed, or who changed code and when. These motivations are found in local code search [97], where code is found to then change for debugging, refactoring, or other maintenance tasks [34], [61], [112]. In contrast, searching on the Internet, code is found to support different cognitive processes (sometimes referred to as discovering requirements [36]) or sometimes used as-is. These difference between local search and Internet search also lead to very different behaviors in empirical

studies of local search (e.g., navigation in file hierarchies and editor windows or navigation for bug fixes and line changes) [92] as compared to the behavior of Internet-scale code search discussed above.

- The difference between the goals and behavior between local code search and Internet-scale code search has implications on experiment design. In particular, it suggests that the design of studies of searching for code on the Internet should mimic and remain as true as possible to the setting and goals of Internet-scale code search.

2.2 Tool Support for Code Search

Research has explored a wide range of tools and techniques to support code search. These tools can be organized into six categories of support: more expressive queries, better matching algorithms, query creation support, result usability, result navigation, and iteration support. In the below, we discuss relevant work in each of these categories.

2.2.1 More Expressive Queries

Recognizing that keywords are somewhat limiting, and do not allow developers to easily target their search to the content of code, several search tools have been developed to match structural queries, usually submitted with an advanced query form. These approaches support search by the code's method signature [105], packages [49], framework [126], and language constructs (e.g., if statements) and relationships (e.g., one method calls another)

Figure 7 AGORA screenshot.

[67]. The insight behind these approaches is that queries on structural parts of the code can be used to find functionality, because of developer naming conventions. For example, a query *quicksort* on class names can find classes that implement the algorithm quick sort. Figure 7 presents one of the first search engines, AGORA, supporting search over structural properties of code. The user can search for Java classes (specifically Java Beans) by specifying, in an advanced syntax, what method names the class should have. Once submitted, links to results are presented below the form. Such search engines have been shown to increase the solution quality of search results [49] and increase the recall of results [126].

2.2.2 Functional Semantic Queries

Rather than searching for functionality based on the structure of the code, several approaches have investigated how to support developers in more meaningfully searching

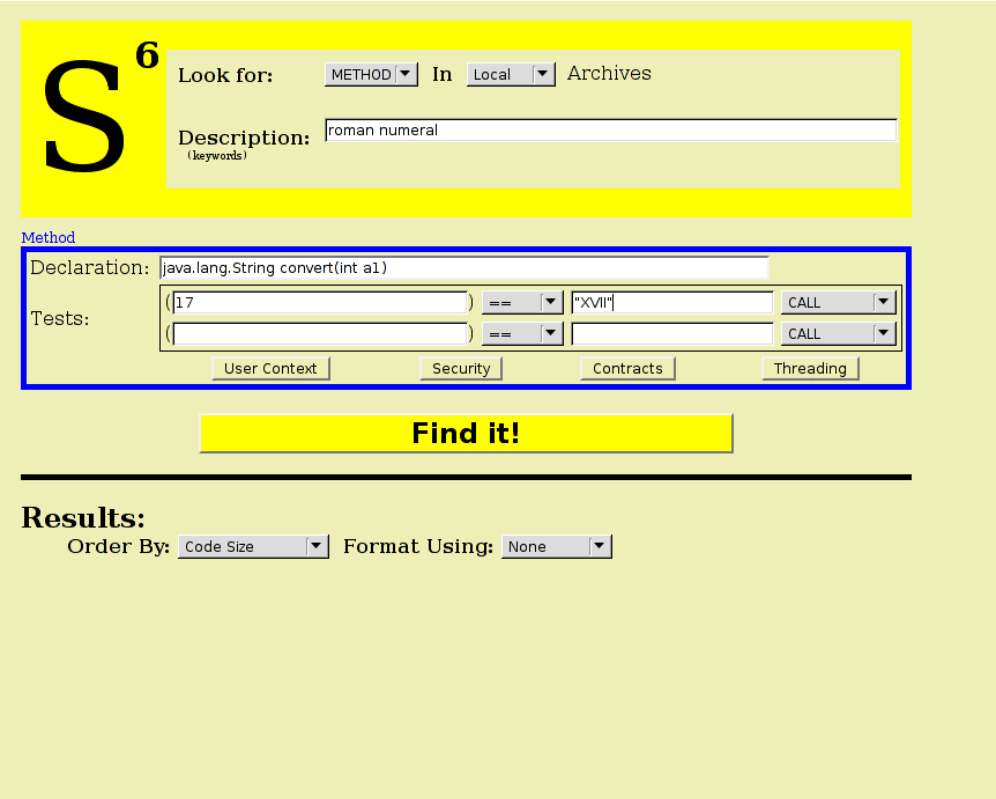


Figure 8 S6.

for specific functionality by supplying semantic queries. A common approach is to use test cases as queries to find code passing them. One approach transforms test cases into structural queries to match code that could pass the test cases with some modification [65]. Other approaches take test cases and transform the code indexed to run on these test cases, where passing code is returned [54], [91]. For example, Figure 8 presents S6, a search engine that accepts a method signature (in this example a method called *convert*) and a series of test cases, as assertions, that the implementation of *convert* should satisfy (in this case, given input 17, *convert* should output "XVII"). Once the user selects the "Find it!" button, the search engine will transform the signature of methods indexed to match *convert*'s method signature and then test if the transformed code passes the test case. If so, S6 returns links to the transformed code below the submission form. A study on 13 test cases submitted to S6 showed that it is possible to find results [91]. At the same time, due to the experimental

nature of the tool, it could take several minutes to respond to a query. A study on a similar kind of search engine, Code Conjurer, showed it can take hours for results to be returned [54].

A less common approach to semantic search is to use a satisfiability modulo theory (SMT) solver that takes well-defined constraints on the input/output of the code, and the search engine then uses an SMT solver to find code satisfying those constraints [116]. There is also research focusing on locating more specific kinds of functionality, such as code that transforms a variable of one type into another. In this case, the programmer issues a type mapping (A → B) and the search engine then attempts to find a sequence of code that transforms a variable of type A into a variable of type B [71], [122], [134]. In [71], the authors show that, given the query type mapping IEditorPart → IDocumentProvider, their search engine can find code results such as

```
IEditorPart inp = ep.getEditorInput();
DocumentProviderRegistry dpreg;// free variable
IDocumentProvider dp = dpreg.getDocumentProvider(inp);
```

transforming variable inp of type IEditorPart into variable dp of type IDocumentProvider.

2.2.3 Task Description Queries

One approach explored a different use of structure in constructing queries. Specifically, it supports a form of code search in which the query is composed of a sequence of commands specifying what structures need to be implemented. This approach was designed to support building plugins in the Eclipse development environment, which entails creating a plug-in project, configuring XML files, implementing interfaces, overwriting methods, and other activities that amount to creating different program structures using Eclipse libraries. The user can issue these kinds of steps, each step naming the structure to be created, as a query to the search engine to get sets of results that have implemented these steps. Figure 8 shows the interface to XFinder [23] that implements this approach. In this example, the user has issued a query to create a text editor plugin to the Eclipse environment. The steps are displayed in the root nodes of the tree viewer (e.g., the last step is *Implement IContentOutlinePage*) and under each step are results implementing it. Examples are found

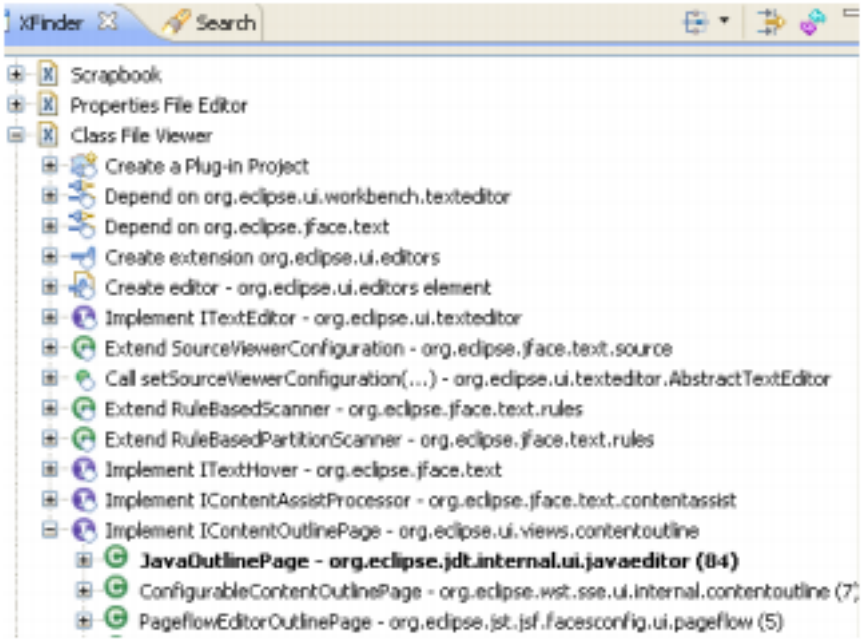


Figure 9 XFinder.

by mapping each step onto matching structural properties of source code in the repository. For example, *Implement IContentOutlinePage* can be mapped onto any class using the language construct implements followed by the class name IContentOutlinePage. The authors of XFinder ran a small case study testing the precision of their tool. Within two different code bases for different queries, they mapped the examples XFinder found to the examples the authors found, and measured the precision to range from 88.5% to 97.8%.

2.2.4 Incomplete Code Queries

Several approaches support the user to write part of the code they need (at the method or statement granularity) and submit it as a query to get results that complete it. This approach is intended to more seamlessly go from code to results, rather than requiring the programmer to translate the immediate source code completion needs into a keyword query. These approaches rely heavily on pattern matching abstractions of the code being written with abstractions of code indexed. Abstract syntax trees [82] and binary vectors that encode

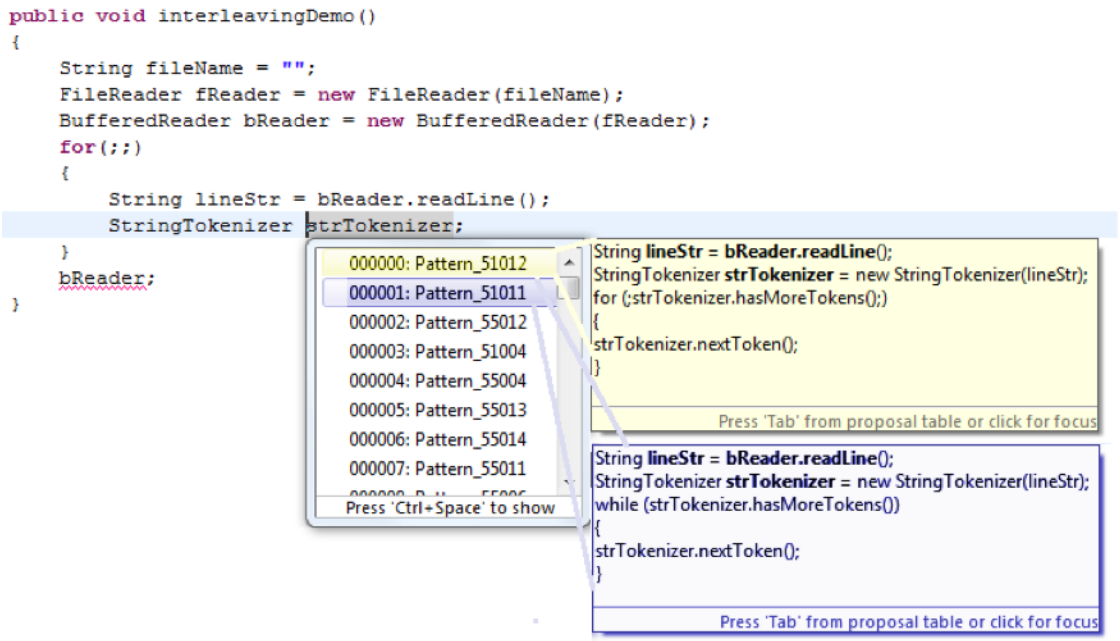


Figure 10 Grapacc.

properties of the code [14], [15] have been used to retrieve statements completing a programmer's function they are writing at the time. Figure 10 shows a screen shot of one such system, Grapacc [82]. It shows the user declaring a StringTokenizer variable and then hitting the keys CTRL+Space to issue it as a query to the Grapacc search engine. Returned are results, displayed in a dialog box, of possible ways of completing the code the user is writing. To evaluate Grapacc, methods from 24 different open source projects were selected and each had their bottom half of code removed. Grapacc was then run on these methods to autocomplete them using a result in the top five. The resulting autocompleted methods were mapped to the actual methods and it was found Grapacc scored a precision of 71% and recall of 77%.

More niche approaches have focused on supporting the developer in completing their instantiation of an object (this can be complicated in large frameworks) using context patterns (e.g., the parent class of the current code and the type being instantiated) [80], [98].

2.2.5 Improving Ranking Algorithms

While some research has focused on approaches for matching code with different kinds of queries, other research has investigated how to improve the ranking of the code that is returned. Next, we discuss the variety of approaches that have been explored.

2.2.5.1 Improving Ranking Algorithms with Automatic Query Modification

To return more on topic results, one approach is to automatically add synonyms and other related words to the current keywords to match topically related code that would be missed by matching only against the programmer's keywords. The terms added can come from a variety of thesauruses [66], rule systems mapping keywords to related terms [28], related Java documentation [41], or from the code the developer is currently writing [12]. For example, Lemos *et al*. [66] found that, when queries were automatically expanded with synonyms from the WordNet [135] thesaurus, it increased recall of CodeGenie [65] by 30% (i.e., query expansion allowed CodeGenie to return more on topic results that otherwise would not have been returned).

2.2.5.2 Improving Ranking Algorithms with an Enriched Index

An alternative approach to matching more topically related code is to index code in the search engine not only with terms occurring in it, but also with descriptive terms elsewhere (e.g., descriptive terms found in the code's documentation). This method supports matching of topically related code that would be missed by matching only against the terms inside the code. Related terms are often taken from documentation [18], [49], descriptive tags from the community [126], [133], and, if the code is on a web page, surrounding natural language on the web page [133]. For example, in the search engine Maracatu, they were able to increase the precision of the search engine by about 23% by using descriptive tags to expand keyword queries. However they also found that it had an inverse effect of lowering the recall by about 37%.

2.2.5.3 Improving Ranking with Weighted Matching

Traditionally, ranking algorithms optimize on matches between keywords and all terms (save stop words) in documents and other document relationship heuristics [152]. However, Linstead et al., found that it is possible to improve ranking performance (measured by the area under the ROC curve metric [45]) for code compared to Google by matching keywords against different parts of fully qualify names in code (e.g., class name, method names) in an algorithm called Specificity [67]. The general procedure is to give equal weight to matches with terms in the least qualified part of the fully qualified names (FQN), but give

higher weight to matches in the most qualified name in the FQN (often referred to as the simple name). This algorithm was implemented in the Sourcerer search engine, as shown in Figure 11. In the figure, the query *http server* (logically AND'ed) was issued, for which Sourcerer returned many classes named HttpServer in the top results because it weighted matches with the simple names of types (in this case class names) more heavily.

SOURCERER

about | usage | projects | reset search

All Components Functions Fingerprints

http AND server Go

Search in comments ?

1 2 3 4 5 6 7 8 9 10 | >>

org.mortbay.http.HttpServer

CLASS (rank: 4.437308781281751)

Relations >> Find uses

Fingerprints >> Show Details

Source >> Inline | Expanded | Browse in Project | Download

[X] Close

```
 * @see org.mortbay.jetty.Server
 * @version $Id: HttpServer.java,v 1.49.2.3 2004/07/10 06:52:41 gregwilkins Exp $
 * @author Greg Wilkins (gregw)
 */

public class HttpServer implements LifeCycle,
                                   Serializable
{
    /* ------------------------------------------------------------ */
    private static WeakHashMap __servers = new WeakHashMap();
    private static Collection __roServers =
        Collections.unmodifiableCollection(__servers.keySet());
    private static String[]  noVirtualHost=new String[1];

    /* ------------------------------------------------------------ */
    /** Get HttpServer Collection
```

Project: jetty / Version: 4.2.25-all / License: Artistic License , Other/Proprietary License / Category: ?

org.mortbay.http.HttpServer

CLASS (rank: 4.434554928896655)

Relations >> Find uses

Fingerprints >> Show Details

Source >> Inline | Expanded | Browse in Project | Download

Project: usemodj / Version: usemdoj-utf8_v1.2-jetty-4.2.22 / License: GNU General Public License GPL / Category: ?

com.programics.simpleweb.HttpServer

CLASS (rank: 1.0463815508704)

Relations >> Find uses

Fingerprints >> Show Details

Source >> Inline | Expanded | Browse in Project | Download

Project: javujavu / Version: iserverbox1.1.1_src / License: GNU General Public License GPL / Category: ?

net.noderunner.http.ServerRequest

INTERFACE (rank: 0.924083225108225)

Relations >> Find uses

Fingerprints >> Show Details

Source >> Inline | Expanded | Browse in Project | Download

Figure 11. Sourcerer.

2.2.6 Query Formation Support

Rather than seeking to support improvement in matching the user's query, some approaches attempt to automatically create or recommend queries that may lead to better matching.

2.2.6.1 Automatic Query Creation

Several approaches attempt to eliminate the effort of formulating queries, as well as when to formulate them, by automatically constructing queries on behalf of the user and continuously pushing results to the user. The insight behind these approaches is that the context of the developer (e.g., the project they are working on) can determine, in part, some initial queries. The typical approach is to use names from the user's opened project (e.g., method names and class names) [3], [40], [131] as well as certain structural properties in the project (e.g., method signatures and supertypes) [51] to automatically construct a query. Further, while some approaches rely on the programmer to select code in their project to initiate a query [51], some approaches will automatically push results to the programmer as the programmer is writing [3], [40], [131].

Using the code context to automatically create a query is different than issuing the incomplete code queries in Section 2.2.4, because those methods rely on the user to manually create the query out of code, whereas the approaches in this section automatically create a query leveraging the code context to do so. Figure 12 shows a screen shot of CodeBroker, a system that takes the code currently being written, transforms it into a query, and pushes the results back to the programmer in the editor. In the figure, CodeBroker has taken the

method getRandomNumber and surrounding context, transformed it into a query, and returned links to different random number generator methods at the bottom of the Emacs code editor.

The authors of CodeBroker ran a user study among five participants whose task was to implement a program with the option to use CodeBroker. The authors found that, among the 50 components reused by participants, 20 came from CodeBroker. For nine of the 20 components taken from CodeBroker, the participants reported not anticipating using them beforehand. Participants reported CodeBroker made it faster to reuse 11/20 components. Further, participants stated that the results themselves helped shape and guide the creation of the code by giving the participants ideas they would not have had otherwise.

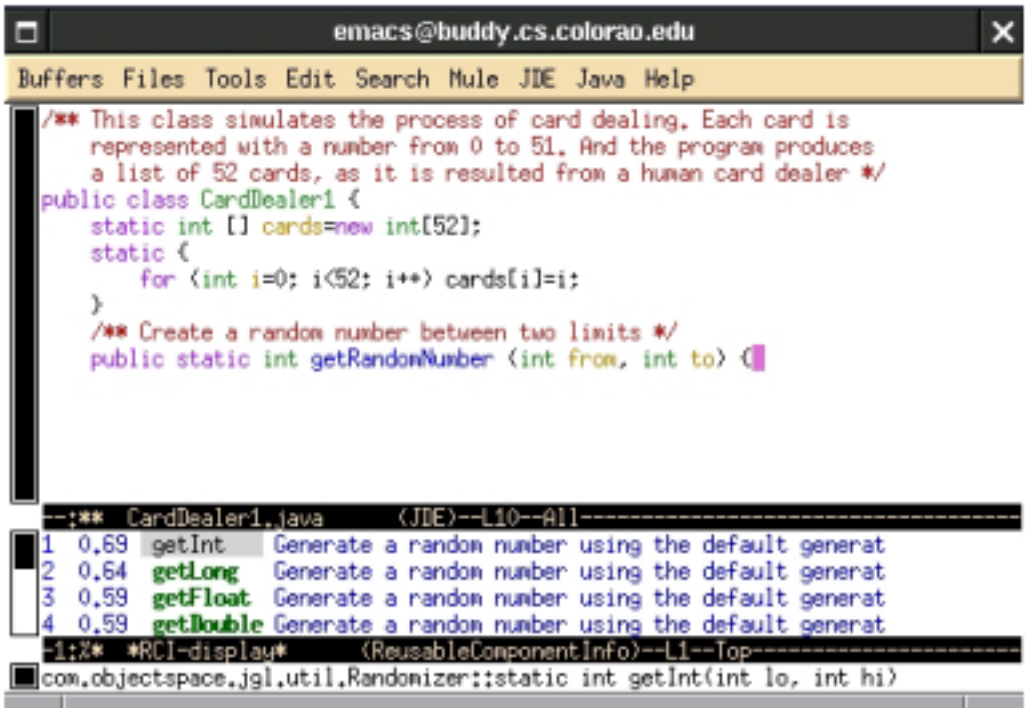


Figure 12. CodeBroker.

2.2.6.2 Query Replacement in Local Code Search

While not Internet-scale approaches to code search, some local code search approaches, for various maintenance tasks, support the programmer in replacing or completing their keyword queries. For example, rather than improving ranking algorithms, Refoqus [44], as shown in Figure 13, assumes the user's query is what needs improving and will recommend ways of replacing a user's query based on the output of a machine learning model. Before Refoqus can be used, it must be trained by suppling it with training data as a list of queries and corresponding code that should be returned from a TF-IDF ranking algorithm (as implemented in Lucene [137]). During training, all queries in the training data are issued to Refoqus and if they do not return what is expected, they are classified as "low quality". Refoqus uses the trained classifier to match future queries to low quality or high quality queries, and, if low quality is matched, will offer recommendations for replacing the query. The authors of Refoqus ran a simulation test to see if their query replacement recommendations would rank methods that needed to be changed for known bugs reported in an issue tracker, higher than other approaches found in the literature. They found that Refoqus could improve ranking over TF-IDF in 52% of queries, do no worse in 32% of

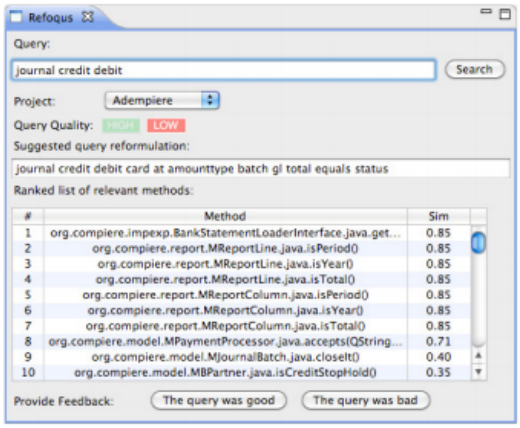


Figure 13. Refoqus.

queries, and do worse in 16%. Further, when comparing against other ranking algorithms, they found Refoqus improved on what's called the effectiveness score (a unit of measure of effort to view all methods returned).

Similar to the assumption behind Refoqus — that the user's query needs improving rather than the ranking algorithm — another approach offers ways of modifying keyword queries if they lead to zero results [38]. Lastly, some methods will recommend terms for completing a query if they occur more often in the search engine's index [88].

2.2.7 Code Result Usability

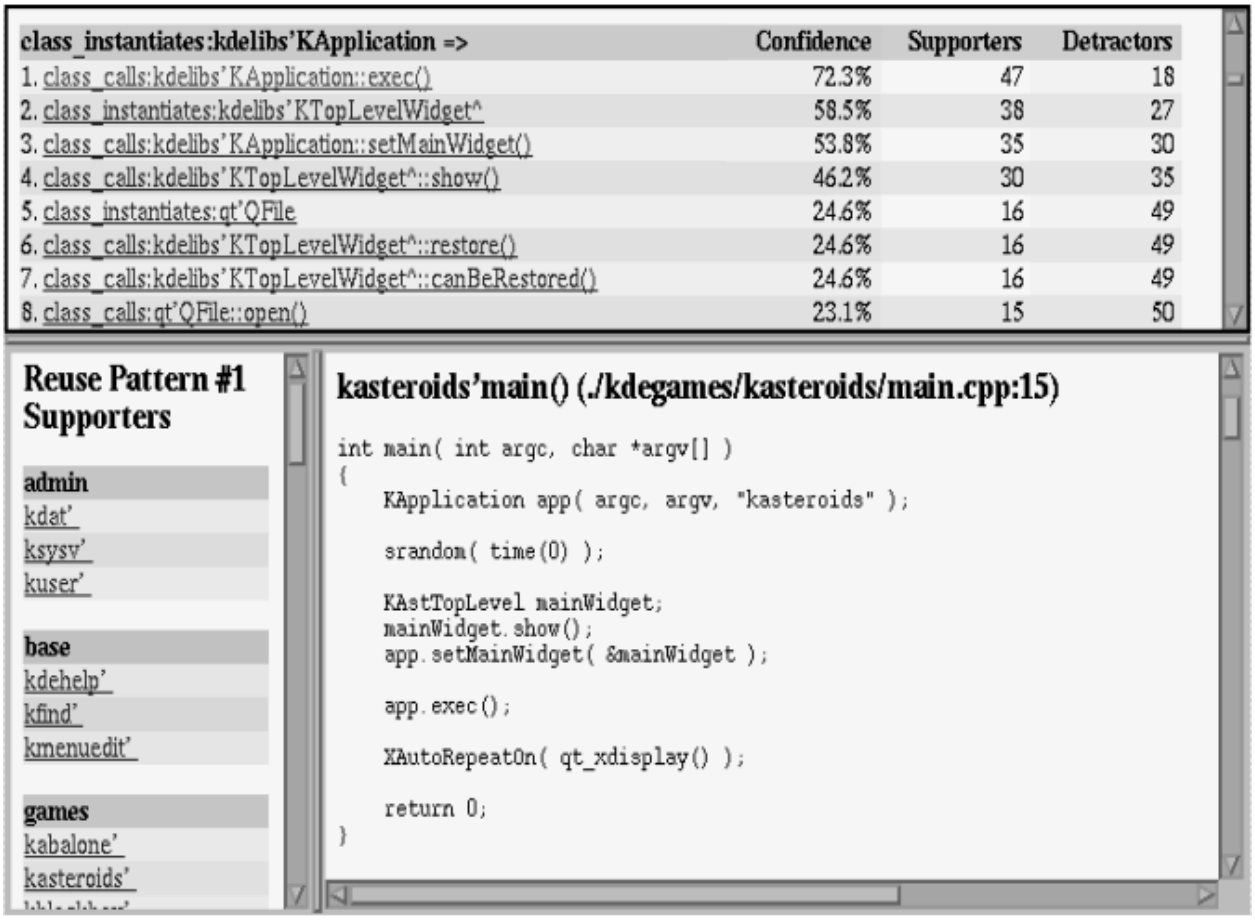
While returning topically related code to a query is crucial, other research has noted that the usability of the results in terms of their quality, understandability, and ease of integration are also important for search. Poor quality results, difficult to understand results, or results that are hard to reuse (if looking for code to reuse as-is) can all increase the programmer's effort using the code they find and impact the value of searching for code in the first place.

2.2.7.1 Code Result Quality

Quality can mean many things for source code (e.g., few bugs, well documented, or low cohesion). However, techniques in code search most commonly use an indirect measure to find code of higher quality, for instance through popularity metrics. Popularity itself has many meanings, but it has been measured by counting the number of times code is used by other code (similar to PageRank [85] once used by Google) [67], [76], [123], and it has also been measured by extracting high level patterns from the code indexed and counting how

often those patterns occur in the search engine [31], [60], [77]. Another approach proposes to use votes from other developers [43]. These popularity scores are then used in the final ranking algorithm to organize results differently than just in order of being on topic.

One of the first search engines to use patterns to rank code was CodeWeb [77], as shown in Figure 13. It ranks code by the popularity of the patterns they implement. CodeWeb extracts implementation patterns from code as implementation rules and scores how confident (using the frequency) it is in the rule. For example, a rule might be "if class instantiates object X then call method Y in next statement" with a confidence of 70% in the rule. Once these patterns are extracted, CodeWeb ranks code results by the confidence value of the rule they



class_instantiates:kdelibs'KApplication =>	Confidence	Supporters	Detractors
1. class_calls:kdelibs'KApplication::exec()	72.3%	47	18
2. class_instantiates:kdelibs'KTopLevelWidget^	58.5%	38	27
3. class_calls:kdelibs'KApplication::setMainWidget()	53.8%	35	30
4. class_calls:kdelibs'KTopLevelWidget^::show()	46.2%	30	35
5. class_instantiates:qt'QFile	24.6%	16	49
6. class_calls:kdelibs'KTopLevelWidget^::restore()	24.6%	16	49
7. class_calls:kdelibs'KTopLevelWidget^::canBeRestored()	24.6%	16	49
8. class_calls:qt'QFile::open()	23.1%	15	50

Figure 14. CodeWeb.

implement. For example, in Figure 13 the rules are listed at the top of the window, where the first rule has a confidence of 72.3% and the code implementing this rule is displayed in the bottom right window. While no empirical study or user evaluation was done on CodeWeb, the tool was among the first to demonstrate how to rank code by popularity, or one notion of popularity, showing the possibility of including quality into the search process.

2.2.7.2 Code Result Understanding

Another critical part of code search is the ability of the user to understand the code results, as otherwise it may be too difficult for the user to know when they have found code they want. For example, in [99] they report that comprehending source code can be very hard on Stack Overflow, as 75% of results have less than 19 lines of code. Most code snippets, too, do not compile and are typically part of short and poorly structured posts.

One very early approach, shown in Figure 15, supported the user to select parts of the code to issue "why" questions to retrieve manually created documentation explaining the selected part [33]. A more automatic approach collapses code results intro groups by different functionalities in the code. Collapsing code into groups supports the developer to get a higher level understanding of a code result and can be uncollapsed, when the user is ready, to obtain a more detailed understanding of the code [99], [100]. Some other methods include documentation and comments from the web with the results [87], [132] or support finding an expert that can explain code to the user [132]. Generally, users of these systems report being able gain a higher understanding of the code compared to other systems.

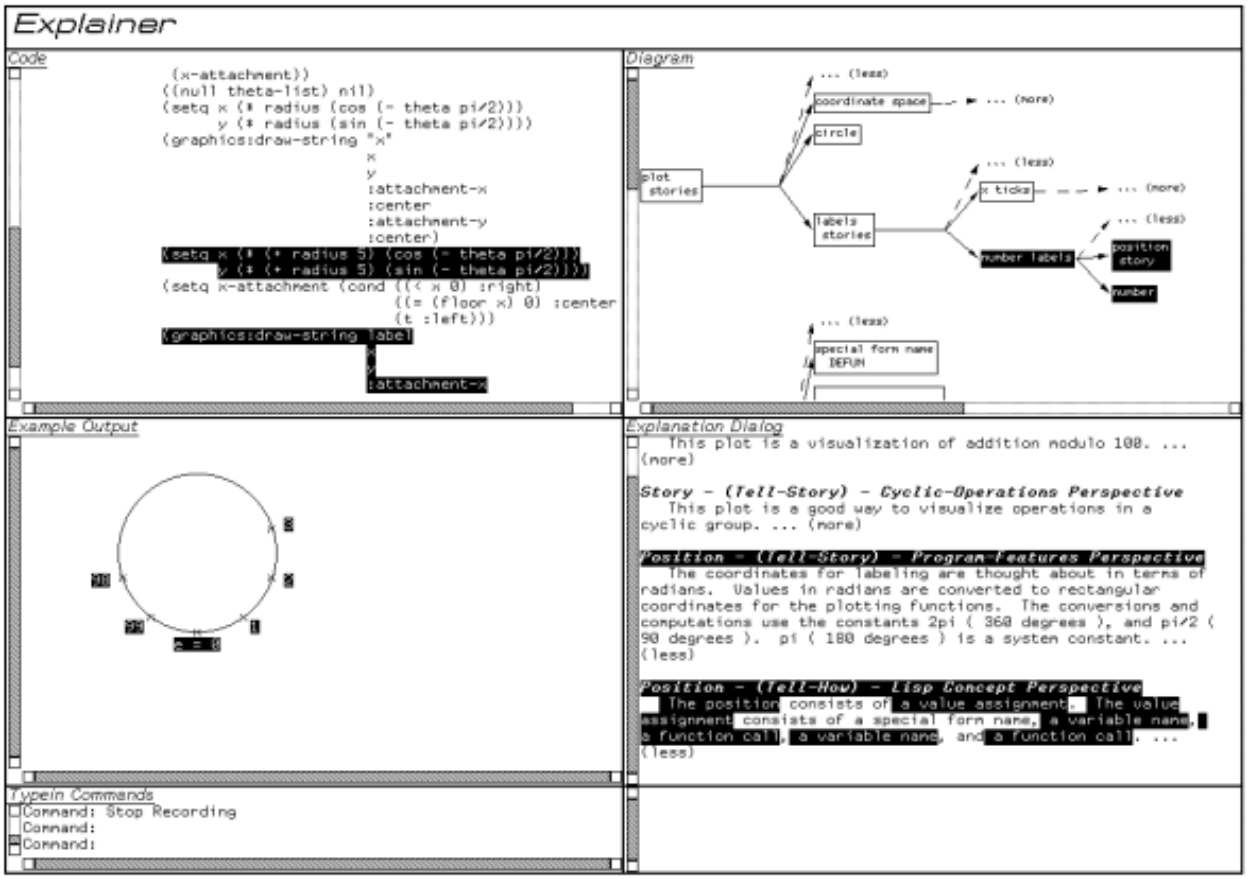


Figure 15 Explainer

2.2.7.3 Code Result Integration

If code is reused as-is, then the programmer needs to integrate it into their project. This can be difficult and time consuming, because it entails resolving all the dependencies of the code and then integrating those dependences. This can mean renaming/creating variables or locating libraries and other dependencies (e.g., configuration files) to also integrate into the project. To support integration, two kinds of approaches havalphae been researched. One approach is to support developers to write integration templates and documentation that can later be used by other developers to help integrate a code result into their project [83], [128]. For example, Figure 16 shows a screen shot of a programmer reusing some CSS code and using a template in the Codelets [83] tool to integrate it into their local code. Codelets

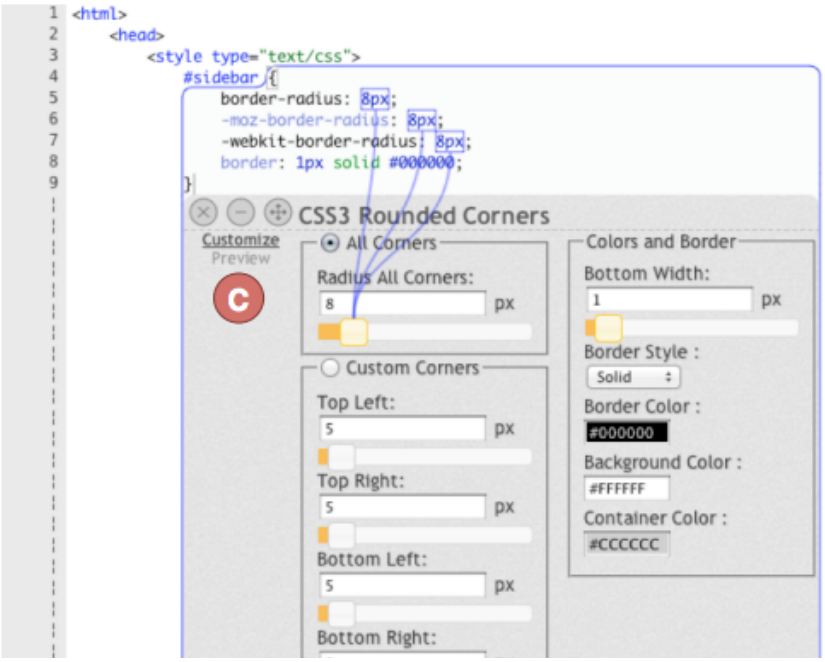


Figure 16 Codelets.

was evaluated in a user study and was found to decrease the time it takes programmers to implement web applications.

The other approach is to support the developer in locating dependencies. One method for doing so automatically tries to resolve all dependencies needed to use a code result in a search engine [84]. Another method supports the programmer in the activity of locating dependencies by giving them an interactive visualization to navigate dependences from the code they want to integrate and ignore dependences from code they do not want [52]. Lastly, there are methods that index libraries and source code and return both when matching keyword queries [49]. In general, these approaches intend and sometimes are shown to reduce the time it takes to implement code.

2.2.8 Result Navigation

Traditionally, navigation of code search engine results is done by paging through 10 ranked results at a time. However, some code search engines, many commercial, support navigating the results by scoping them with descriptive fields called filters. For example, the commercial search engine Krugle supports scoping results, as shown in Figure 17, by known projects, file types, and authors of the code indexed. Not much evaluation has been done on filters, since most commercial code search engines do not publish research on themselves. However, in Sim *et al.*'s user study [106], they evaluated code search engines that have filters (Krugle, Koders, and Source Forge) and found that 42.2% of all queries issued included the use of a filter, suggesting they have much utility in finding code.

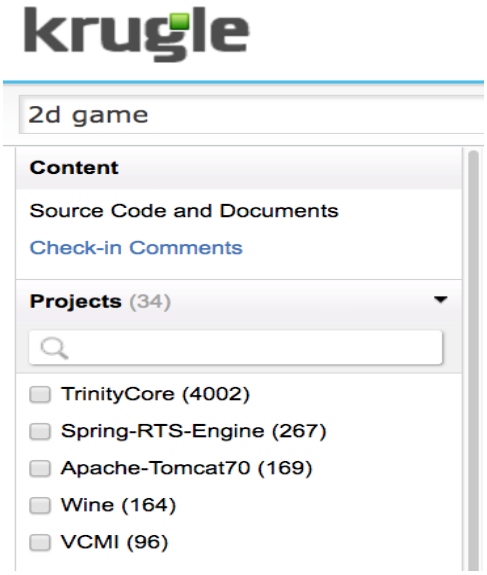


Figure 17 Krugle.

2.2.12 Iteration Support

While there has been positive and impactful research in tools supporting more expressive queries, better matching and ranking of results, and result usability, each of these areas of research focus on queries and results in isolation of the next query and results. However, as discussed in Section 2.1, we find search consisting of multiple queries, where the next query is often a modification of the previous – making each query a step in a connected process rather than as isolated events.

An early small-scale approach to code search (searching 1800 Lisp function) recognized the need to assume a next query and designed some support for it [47]. This early approach supported the programmer with iterative refinement, where, after submitting a query, the search engine would present related keywords that could be selected to refine the query. Figure 18 is a screen shot of the implementation of this approach in a tool called CodeFinder. In Figure 18 (1), points to related keywords that can be added to the query, (2) points to where a summary of a single result is returned, and (3) points to the programmer's current query. The author of this approach showed through user evaluations among nine students, each with programming experience, that explicit support for iteration helped find code for ill-defined code search tasks compared to other non-iterative approaches.

Much later, support for refinement recommendations in web search engines was investigated by extracting words from Java documentation in results from Google and presenting these words to the user as refinement recommendations [119]. The authors of

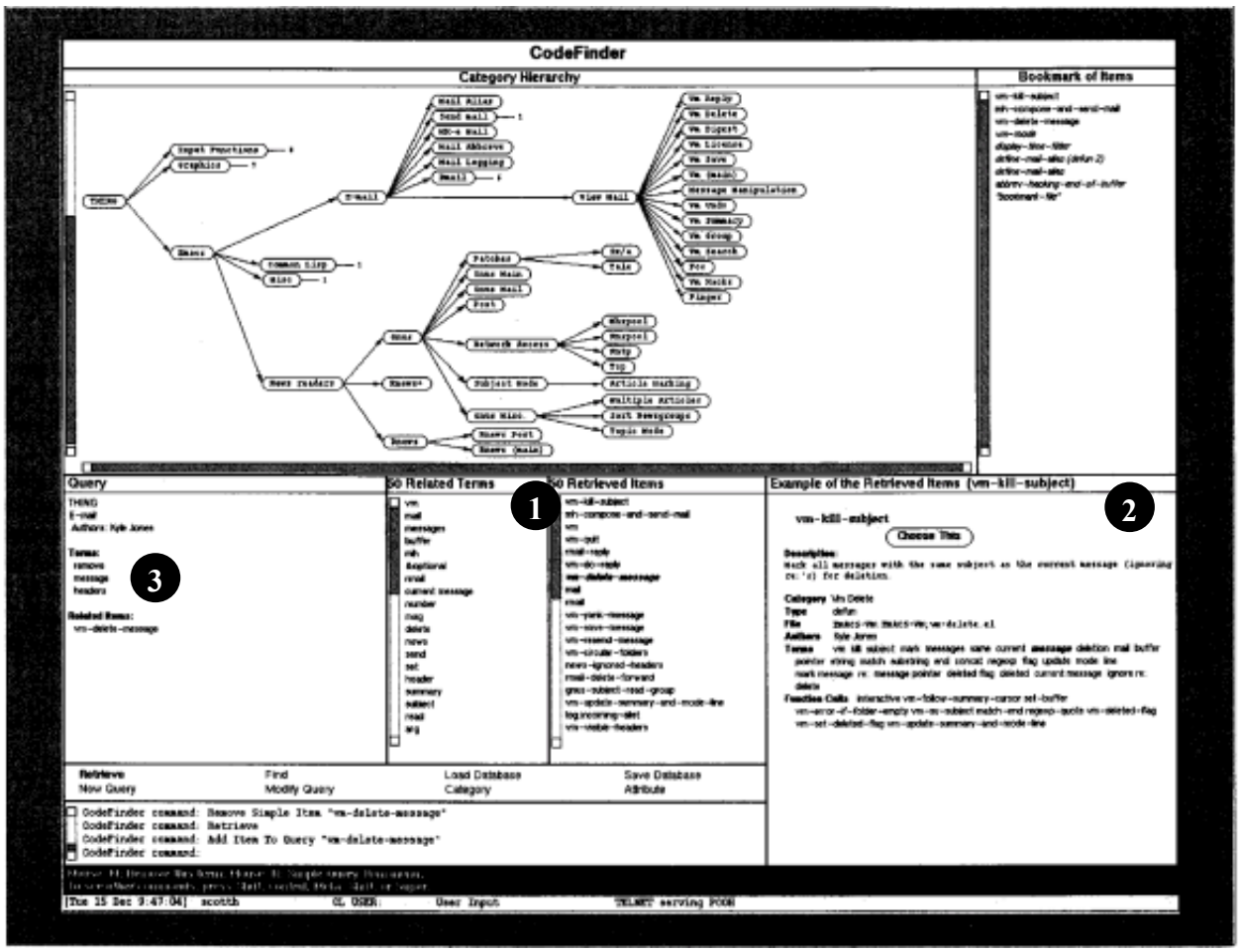


Figure 18. CodeFinder.

this approach conducted a field study and found about half of all searched involved taking a keyword refinement recommendation.

Lastly, while iteration has not been the primary aim of many approaches and even when supported often not evaluated, some tool research has acknowledged the importance of features supporting iteration, albeit tangential to their main design. This includes future work discussions illustrating the possibility of offering refinement recommendations by frequent method calls in the results [5], [8] or by the package in which a function occurs [132].

2.2.14 Tool Support Discussion

Clearly, tool support research for code search has investigated many different approaches to improve queries and results. Emphasizing this point, we present in Table 2 and Table 3 a high-level summary of all 50 tools and methods discussed in this section. The headers of the table organize the kinds of tool support for code search discussed and each row references specific tools or methods by name. Cells with a value of "✓" map tools and methods to their corresponding support. For example, the first row shows that the Agora search engine supports structural queries and paging of results. As another example, the table shows that CodeFinder has an iterative refinement feature that was also evaluated.

Of importance to this dissertation are the following two observations:

- There is much more research to be done in the design and evaluation of iterative support for code search engines. Among the 50 tools and methods, only two have developed a feature to support iteration that was also evaluated. However, in 100% of the empirical studies on code search that look at the number of queries issued, search is shown to be iterative.
- There are many kinds of features supporting iteration left to investigate. The two iterative features designed and evaluated (in CodeFinder and Mica) are both refinement recommendation features that only support iteratively refining a query with keywords. While clearly important, other support such as refining the query by characteristics of a result deserve equal exploration.

Based on the two observations immediately above, this dissertation designs and evaluates diverse approaches to iteratively search for code.

Table 2. Code Search Engines and Techniques, Part 1.

Code Search Engines / Techniques																			
	More Expressive Queries than Keywords						Better Matching		Query Creation Support			Result Usability			Result Nav		Iteration Support		
Name	Regular Expression Query	Dependency Query	Structure Query	Task Description Query	Code as Query	Functional Semantics Query	Enriches Query	Enriched Index	Automatically Creates Query	Keyword Autocomplete	Query Quality Recommendations	Example Quality	Code Result Understanding	Code Result Integration	Paging	Filter	Iterative Refinement Feature	Iteration Support Evaluated	Type of Support
Agora [105]			✓												✓				Code Search Tool
Assieme [49]		✓	✓					✓							✓	✓			
B.A.R.T [28]							✓												
BluePrint [12]							✓												
Calcite [80]					✓														
Code Conjurer [54]						✓													
CodeBroker [131]									✓										
CodeGenie [65]						✓													
CodeRecommenders [142]					✓														
Codetrail [40]									✓										
CodeFinder [47]																	✓	✓	
CodeWeb [77]												✓							
Codelets [83]														✓					
CodeX [31]												✓							
Explainer [33]													✓						
Exemplar [41]							✓												
Example Overflow[133]								✓							✓				
FrUiT [15]					✓														
Gilligan [52]		✓																	
GraPacc [82]					✓														
Jbender [43]												✓							
JIRISS [88]											✓								
MAPO [134]			✓																
Maracatu [126]			✓					✓											
Mica [119]															✓		✓	✓	

Table 3. Code Search Engines and Techniques, Part 2.

Code Search Engines / Techniques																			
	More Expressive Queries than Keywords						Better Matching		Query Creation Support			Result Usability			Result Nav		Iteration Support		
Name	Regular Expression Query	Dependency Query	Structure Query	Task Description Query	Code as Query	Functional Semantics Query	Enriches Query	Enriched Index	Automatically Creates Query	Keyword Autocomplete	Query Quality	Example Quality	Code Result Understanding	Code Result Integration	Paging	Filter	Iterative Refinement Feature	Iteration Support Evaluated	Type of Support
PARSEWeb [122]			✓																Code Search Tool
Portfolio [76]								✓											
Prospector [71]			✓																
S6 [91]						✓													
SeaHawk [87]									✓				✓						
SNIFF [18]													✓						
SnipMatch [128]														✓					
Sourcerer [67]		✓	✓					✓							✓				
Sourcerer API Search [8]															✓	✓	✓		
Spotting Working Code Examples [60]												✓							
SpotWeb [123]												✓							
STEP_IN_JAVA [132]							✓	✓									✓		
Sythesizing API usage Examples [16]																			
Strathecona [51]			✓						✓										
Vesperin [99]													✓						
Xfinder [23]				✓															
Xsnippet [98]					✓														
Coronado [38]										✓	✓								Local Code Search Tool
Refoqus [44]											✓								
AQE [66]							✓												Technique
MTU [100]													✓						
SMT [117]						✓													
Koders / Ohloh / OpenHub [140]															✓	✓			Commercial Tool
Google Code Search [146]	✓														✓				
GitHub Code Search [144]															✓	✓			

2.3 Iterative Search in Information Retrieval

While people search for natural language documents for different reasons than code, and while code and documents are very different mediums, some of the work in the field of information retrieval can inspire new directions of research in code search. Relevant to this dissertation are the approaches investigated that support iteratively searching for unstructured natural language documents.

John Tukey *et al.*'s Scatter-Gather [22] method was among the first to support iterative search for news articles. Their search engine presents a high-level clustering of all the articles indexed with summaries for each cluster. The user selects several clusters that appear on topic and the search engine will take the subpopulation of articles in the selected clusters and return finer grained clusters on the subpopulation. This gradual iterative refinement continues until there are few enough documents in the selected clusters to stop clustering and to just return the documents for the user's inspection. Figure 19 shows the implementation of this method as a command line tool. The user is presented with eight clusters (0-7), each with terms to describe the clustered documents.

A more recent version of the Scatter-Gather approach accepts a keyword query first and then build clusters dynamically on the returned results [46]. This approach supports the user to start their search with clusters topically related to a description of what they want, rather

```
> (time (setq first (outline (all-docs tdb))))
cluster 4970 items
global cluster 199 items...sizes: 18 24 53 5 25 47 13 14
move to nearest...sizes: 517 1293 835 86 677 1020 273 269
move to nearest...sizes: 287 1731 749 275 481 844 310 293
0 (287) CRITICS URGE NEW METHODS; PROGRAMS FOR PARENTS THE; TEACHING SUBJECTS T
 school, year, student, child, university, state, program, percent, study, educ
1 (1731) FEDERAL WORK PROGRAMS HE; RESORT TAKES STEPS TO PR; AMERICANS CUT BACK
 year, state, york, city, million, day, service, company, week, official, house
2 (749) PENTAGON SAYS 60,000 IRA; BUSH ''DRAWS A LINE'' IN; BUSH SAYS FOREIGNER
 iraq, iraqi, kuwait, american, state, unite, saudi, official, military, presid
3 (275) Trillin's Many Hats; New Musical from the cre; After Nasty Teen-Agers i
 film, year, music, play, company, movie, art, angeles, york, american, directo
4 (481) TWISTS AND TURNS MAY MEA; SAX LOOKING FOR RELIEF I; PAINTING THE DODGER
 game, year, play, team, season, win, player, day, league, hit, right, coach, l
5 (844) CRISIS PUSHES OIL PRICES; WHY MAJOR PANIC OVER A M; OIL PRICES RISE AS
 price, oil, percent, market, company, year, million, stock, day, rate, week, s
6 (310) LEADERS OF TWO GERMANYS ; REPRESENTATIVES OF TWO G; SECURITY COUNCIL RE
 government, year, state, party, political, country, official, leader, presiden
7 (293) U.S. APPEALS ORDER FREEI; DID JUDGE MOVE TOO HASTI; MAYOR BARRY CONVICT
 case, court, charge, year, judge, lawyer, attorney, trial, jury, federal, dist
real time  131258 msec
```

Figure 19. Scatter-Gather.

than starting with very generic clusters that may have no obvious meaning to the user or their search, making selecting clusters hard. The user study of the new Scatter-Gather approach showed that people could find more on topic articles with the updated Scatter-Gather approach compared to search engines returning results ranking articles with TF-IDF.

Rather than iteratively selecting clusters to search for documents, Smucker and Allan's work [114] presents a way to iteratively search for medical journal articles on the PubMed web site [89] by selecting articles of interest to get more articles on a similar topic. After selecting an article and getting results, the user can again select an article to find articles on a similar topic, and so on. This approach is implemented by transforming the selected article into a keyword query to find other journal articles containing similar terms. A similar, but not iterative, approach supports search for similar documents by calculating the Hamming

distance (number of edits required to go from one document to another) between an uploaded document and other documents indexed [129]. However, documents must be uploaded from the local drive to search with, so this approach does not support iteratively searching with the results.

Lastly, another approach, called Rocchio [152], is similar to Smucker and Allan's, but adds terms (logically OR'ing them together) to the keyword query from all user selected results that appear on topic. In this way, the query expands with each selection and will contain terms from all results ever selected during the search. The idea behind this approach is that eventually the expanded query will grow with all terms describing a topic so that all documents on the topic of interest are returned. A possible issue with this approach is that the query can grow so large that the precision of the results begins to decrease with each selection of additional results.

Like Sucker and Allan's work, this dissertation too investigates iteratively searching by result similarity, but where similarity is defined for source code results and defined at various degrees to get different and similar code through the search.

Chapter 3

Research Question

In the previous chapter, we discussed various empirical studies on code search that found code search to be an iterative process, where the programmer submits a query, examines and sometimes learns from the results, submits a new query, examines the next results, and so on. In contrast, most tool support for code search has taken a non-iterative approach, which optimizes on simply returning the best results for whatever the query is. In a sense, this is reasonable, because success in doing so would mean giving the developer exactly what they queried for. Yet, only returning the top ranked results for a query is *insufficient*, because it offers no support for the developer in creating a next query, which the studies in the previous chapter show programmers do and often when they are not exactly sure what they are looking for. In such cases, search is more of an exploratory *process* where multiple

queries are issued by the programmer so that they can discover results to learn and gain ideas from or to learn more about what code they might want in the search engine.

Our work distinguishes itself by focusing on leveraging the results from the first or a subsequent query as the primary vehicle for assisting the developer in formulating a next query. Our motivation to do so it is based on a two-fold observation:

(1) When programmers are working on formulating a next query, they frequently are in the process of also formulating what they are searching for, which makes the next query less about finding the right words for a specific and known need but more about figuring out what to search for. Some of the empirical work described in Section 2 suggests that search is more exploratory when programmers do not quite know what they are looking for, which often involves learning and getting ideas about what to search for as they look at the results. This dissertation focuses on this process, and particularly explores novel approaches to support developers in searching for code in an iterative manner when they may not initially know exactly what they are looking for.

(2) Programmers tend to create a next query relative to the results they receive. For example, they might decide there is some aspect of one or more of the results that they want to see more of in the next set of results, and attempt to encode that aspect as keywords to get to those examples. In contrast, a programmer might not like a certain aspect of the results and attempt different keywords to prevent those aspects from occurring in the next set of results. This behavior is not unlike Polya's problem solving model (shown in Figure 20) [159][81], which observes that, in the iterative process involved in solving some problems, iterating tends to either generalize

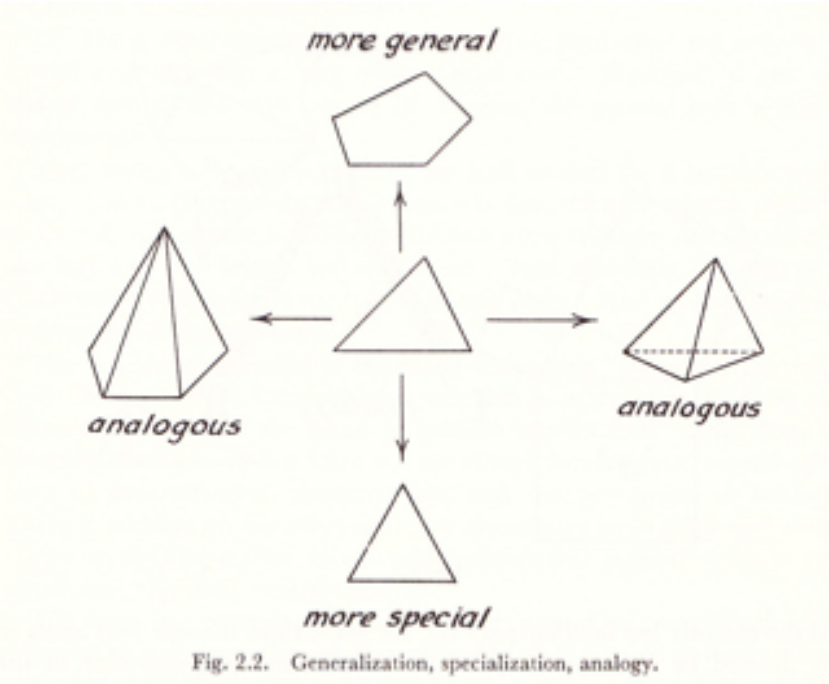


Figure 20. Polya's model of problem solving.

(triangle to polygon Figure 20), specialize (triangle to equilateral triangle in Figure 20), or move to an analogous solution (triangle to pyramid in Figure 20). In traditional search for source code, this would translate into "fewer keywords", "additional keywords", and changing some keywords while keeping others the same. Regardless, it still requires the developer to come up with the new choice of keywords given the results. This dissertation seeks to aid this process, in general, through new code search approaches that alleviate the programmer how to do so.

In this dissertation, we specifcally explore two different approaches that help the programmer in using the results of a previous query to construct a next query. The first approach is inspired by the cognitive science literature showing that, often, people can recognize something as partially similar (or not) to something else [113] and that people try to get closer to other solutions by partially adjusting a solution they have to get to a solution

more similar to what they are looking for [81]. Implemented in CodeExchange, our first experimental code search engine, we support this practice with four features that were designed to let the programmer use an aspect of a result (e.g., its complexity, a method call, or a library used) to adjust a query incrementally.

The second approach, implemented in CodeLikeThis, our second experimental code search engine, is inspired by the concept of whole similarity in the cognitive science literature [113], which is about when people can recognize something holistically (rather than any specific part) as similar (or not) to something else [27], [113]. As such, the second approach presents the programmer intentionally with a diverse set of results after the first keyword query, and then expects the programmer to choose one result together with one of three directions to "steer" the search: "more like this", "somewhat like this", or "less like this". The search engine responds correspondingly, allowing the programmer to search simply by choosing whole results, one after the other.

Both CodeExchange and CodeLikeThis are explictly designed to improve support for iterative code search. Improvement, of course, can be along many different dimensions. In this dissertation, we specially examine three:

- *success*, as measured by the programmer's ability to find code they deem appropriate for their need;
- *experience*, as measured by the programmer's ratings of their experience in finding suitable code; and
- *time*, as measured by how long it takes the programmer to find suitable code.

Overall, then, the research question underlying the remainder of this dissertation is as follows:

> *What is the impact of explicitly supporting software developers in searching iteratively on the experience, time, and success of the code search process?*

Chapter 4

CodeExchange

When a programmer is searching for code and is not quite sure what they are looking for initially, different aspects of the results returned for a first query might trigger the programmer to modify their query to attempt to incorporate one or more of those aspects. For example, suppose a programmer, Suzie, is looking to build a new adventure game, but she is not certain about how to get started. She wants to see some examples of how to implement characters and how to handle updating game state, but also wants to discover what other people are doing to get ideas for her own game. To do so, she opens her favorite search engine and issues the keywords *adventure game*. The first code result she sees implements an Adventurer character that extends a Character parent. Recognizing she will probably have many characters in her game, she thinks extending a Character parent is a good idea and wants to see some more examples of how this is done, so she refines her keywords to *adventure game extends Character*. Among the results returned, she notices that

there are some classes that extend the Character parent and they all tend to override methods with names starting with "draw". Upon inspection, she sees each class is responsible for rendering its character graphically on the screen. She thinks this is a good idea, because it makes use of polymorphism, so she could write an update loop that simply iterates over all the characters, regardless of subtype, and call their draw method. To get a better idea of how to render graphics on the screen, she tries a next query *adventure game extends Character draw graphics 2D* to get to more examples of rendering graphics for adventure games.

After searching deeply into code related to characters, Suzie still wants to look around and get some other ideas on how to build adventure games, so she hits the back button several times to return to the original results from the query *adventure game*. As she scrolls through these results, she discovers code for a text adventure game. She immediately stops and has to think about this, because she had assumed before searching that her adventure game had to be graphical, but upon seeing a text adventure game result, she discovers a new possible direction for the design of her game. To learn more about text adventure games, Suzie refines her keywords to *text adventure game*, and once again begins to learn, modify her query, learn some more, and so on, until she is done.

The example of Suzie is not unusual and represents a case where, as the programmer examines the results, certain aspects of the results trigger the programmer to reflect and issue modified queries that attempt to encode those aspects as keywords. For instance, as with Suzie, a programmer must add some keywords to focus on code containing desired

structures, methods, or on a library they may want to explore further. They may also, however, choose to remove some keywords to backtrack or modify some keyword to somewhat shift the focus of the search. Regardless, programmers frequently create next queries in response to the results of a previous query.

CodeExchange, our first experimental code search engine, was specifically designed to aid the developer when, initially, the programmer is uncertain of exactly what they are searching for and is engaged in a more exploratory search involving the submission of multiple queries through which to explore what examples may be available. In such a search scenario, the insight behind CodeExchange is that the next query tends to be relative to the results, and often to specific aspects of the results of the previous query. As such, CodeExchange supports the developer in forming the next query by letting them construct it out of aspects of a result (e.g., method calls, parent class, or complexity), rather than trying to encode it as keywords. In this way, CodeExchange changes how a query is constructed by the programmer, from entering just keywords, to one that is created incrementally out of aspects of the results from each query.

CodeExchange supports the user with four specific features for creating a next query. Two of the features support refining the query by aspects of one of any of the results returned. Specifically, **language constructs** support the developer in selecting structural characteristics of a result (e.g., method calls, interfaces implemented, or code imported) to bring those characteristics into the query. Using a language construct yields a query that is a mix between keywords, if they were a part of the query before selecting a language construct,

and characteristics of results. Unlike keywords, that may or may not retrieve code matching a topic described by the keywords, a language construct constrains the query to retrieve code exactly matching the characteristics specified by that language construct. For example, if the programmer notices a method call of which they want to see more examples (e.g., different ways of parsing parameters from an HTTP request using a method) or wants to see more examples of using a particular library, then they can click on that method call or an import statement to add it to their query. All code returned will have the selected method call or import statement.

The second feature, **critiques,** supports the developer in selecting the value of different technical qualities (complexity, size, number of imports) of a result as a lower or upper bound to bring that bound into the query to constrain the next set of results. In this way, if the developer feels a code result is lacking (e.g., too long or not complex enough), they can bound the next set of code results to attempt to avoid that quality.

In contrast to modifying the query relative to a specific result, **query refinement recommendations** (the third feature of CodeExchange supporting iteration) presents the user with common aspects (imports, parent classes, or interfaces) or domain related terms across *all the results* to add to the query. The recommendations help make visible to the programmer common aspects of results that are difficult to infer just from the top results that are actually visible. For instance, if the third result uses a particular library and is the only result among the top ten to do so, yet hundreds of other results not visible also use that library, then a query refinement recommendation is likely to present that library to the

programmer, making its common appearance visible. After adding a recommendation to the query and getting the results, the recommendations are updated again using the newly returned results. In this way, the programmer can iteratively search by continually selecting recommendations. For example, if the developer issues the keyword query *chess*, they may receive refinement recommendations for keywords *pieces* and *move* and parent class *ChessPiece*. When the recommendation for parent class *ChessPiece* is chosen by the programmer to add to the query, further recommendations are returned, one being the keyword *piececolor*.

The final feature, **query parts**, modularizes the programmer's query each time a programmer adds to a query. Whether by providing one or more new keywords or using one of the new features of CodeExchange, the addition is separately identified by CodeExchange in its interface. Each query, then, consists of a set of separate parts that, together (logically AND'ed), form the actual query issued, but to the programmer remain individual components. Query parts leverage this by enabling a programmer to turn off / turn back on each of these parts separately. In this way, after a programmer gets new results, they can respond by trying different combinations of their query parts to search in different "directions". For example, if the programmer issued a query comprised of three query parts, "adventure game" (initial keywords), "import *java.awt.Graphics2D"* (a characteristic added with a recommendation), and "method call *playSound()"* (a characteristic added with a language construct), but has decided they want sound in their adventure game, but 2D graphics are not required, then they can toggle off the query part "import *java.awt.Graphics2D"* from their query to search for code that is about adventure games and

plays sound. At any moment thereafter they can return to previous results by toggling the query part "import *java.awt.Graphics2D"* back on. Alternately, if they decide to want to learn more about graphics and sound, they could toggle off the query part "*adventure game"* to find more code about rendering graphics and playing sound that may or may not be related to adventure games.

In the remainder of this chapter we detail how the iterative features work in the architecture of CodeExchange and how the user interacts with them through the interface of CodeExchange. We then describe our preliminary evaluations of CodeExchange's features and present our analysis of the results and what we learned.

4.1 Architecture

The architecture of CodeExchange is presented in Figure 21. as a data flow diagram. The features the programmer interacts with and other important functional components of the system are represented as rounded rectangles. Further, the features are grouped together in the dotted box named Features to show they are the main components the programmer interacts with. When the programmer modifies a query with a language construct, critique, recommendation, query part, or a keyword, then a Query Modification (representing either a refinement or a generalization on the current query) is sent to the Query Manager, which forms a new query that the Search Engine Server can parse (our implementation uses the Apache Solr query syntax and server). Once the server parses the query, it will match and rank code indexed and return those results to the Result Processor. The results are not

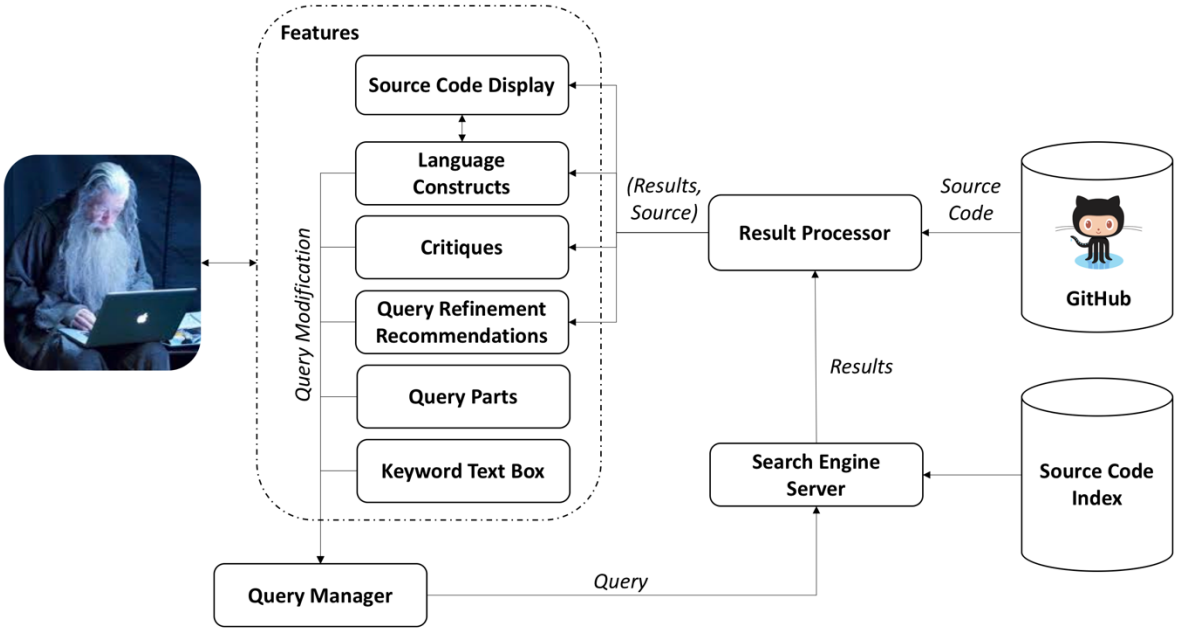


Figure 21. CodeExchange Architecture.

source code, but rather a summary (described by the blue columns of Table 4 and Table 5) of all the information needed by the CodeExchange features, as well as a URL to the source code on GitHub. For each of the results, the Result Processor fetches the source code from GitHub and returns the results and their source code to be further processed by CodeExchange.

The features of CodeExchange know the schema of the results and use the results in different ways that are detailed in the green columns of Table 4 and Table 5. At a high level:

- The source code display presents the results' source code
- Language constructs use the locations in the source code of each result's package, parent class, interfaces, method calls, and method declarations to highlight them in the source code presented to the user and use them to, when one is selected, create a query requiring the corresponding characteristic in the next results

- Critiques use each result's complexity, size, and number of imports to present these values above each result's source code. For each of the values, critiques provide operations that, when selected, create a query that uses the value as an upper/lower bound for the complexity, size, or number of imports of the next set of results.
- Query refinement recommendations use all the results' parent classes, interfaces, and imports to count some top occurring characteristics and provide them to the programmer. Further, terms are extracted from all variables occurring in the results to recommend domain related keywords. Using variable names to obtain domain related words relies on three observations regarding code search and the structure of code. First, search results have co-occurring words that are often related to a programmer's keyword query [95]; this is an essential assumption in the entire field of information retrieval, and applies here as well. Second, variable names usually have domain words in them, and consist often of concatenations of such words [64]. Finally, variable names are the most abundant source of names in the code, as they can make up even 70% of its content [25]
- Query parts can toggle on/off any query composed of any values from previous results.

Table 4. Description of how Features use Results, Part 1.

Schema of Code Indexed	Description	Features that Reference Field	How Features Reference it
Cyclomatic Complexity	The cyclomatic complexity of class as an integer.	Critique Query Parts	A **critique** displays a result's complexity and operations to create a query using the complexity as an upper/lower bound on the complexity of all next results. A **query part** highlights any complexity bound query and, if selected, will deactivate/activate it in the current query.
Size	A count of number of characters in class.	Critique Query Parts	A **critique** displays a result's size and operations to create a query using the size as an upper/lower bound on the size of all next results. A **query part** highlights any size bound query and, if selected, will deactivate/activate it in the current query.
Number of Imports	A count of number of imports in class.	Critique Query Parts	A **critique** displays a result's number of imports and operations to create a query using this number as an upper/lower bound on the number of imports of all next results. A **query part** highlights any number of imports bound query and, if selected, will deactivate/activate it in the current query.
Class start	The location of the start of the class declaration	Editor	The editor displaying result scrolls to start of class.
Import List	A list of all the import statements in the class and their location in the class	Language Constructs Query Refinement Recommendations Query Parts	**Language constructs** highlight each import statement in a result and, if one is selected, creates a query requiring the the import statement in all next results. A **query refinement recommendation** counts the top occurring imports in all the current results and display them to the user. If an import is selected, this feature creates a query requiring the import statement of all next results. A **query part** highlights any import query and, if selected, will deactivate/activate it in the current query.
Parent Class	Name of parent class extend.	Language Constructs Query Refinement Recommendations Query Parts	**A language construct** highlights the extended parent in a result and, if selected, creates a query requiring the extension of the parent class in all next results. A **query refinement recommendation** counts the top occurring parents in all the current results and display them to the user. If a parent is selected, this feature creates a query requiring the extension of the parent in all next results. A **query part** highlights any parent extension query and, if selected, will deactivate/activate it in the current query.

Table 5. Description of how Features use Results, Part 2.

Schema of Code Indexed	Description	Features that Reference Field	How Features Reference it
Interfaces Implemented	List of all interfaces implemented by class.	Language Constructs Query Refinement Recommendations Query Parts	**Language constructs** highlight each interface implemented in a result and, if one is selected, creates a query requiring the implementation of the interface in all next results. A **query refinement recommendation** counts the top occurring interfaces implemented in all the current results and display them to the user. If an interface is selected, this feature creates a query requiring the implementation of the interface in all next results. A **query part** highlights any interface implementation query and, if selected, will deactivate/activate it in the current query.
Method Call	The method call's name, parameter types, and starting location of call in code.	Language Constructs Query Parts	**Language constructs** highlight each method call in a result and, if one is selected, creates a query requiring the method call in all next results. A **query part** highlights any method call query and, if selected, will deactivate/activate it in the current query.
Method Declaration	The method declaration's name, parameter types, and starting location of declaration in code.	Language Constructs Query Parts	**Language constructs** highlight each method declaration in a result and, if one is selected, creates a query requiring the method declaration in all next results. A **query part** highlights any method declaration query and, if selected, will deactivate/activate it in the current query.
Package	Name of package class is in.	Package Refinement Query Parts	**Language constructs** highlights the package the result is in and, if one is selected, creates a query requiring all next results to be in the package. A **query part** highlights any package query and, if selected, will deactivate/activate it in the current query.
Terms in class file	All terms occurring in file.	Keywords Ranking Algorithm Query Parts	Keyword ranking algorithms use this. TF-IDF is default, but we have experimented with others that we describe in later chapters. A **query part** highlights any keyword query and, if selected, will deactivate/activate it in the current query.
Terms in Variables	All terms, split by camel case, in variables in class	Query Refinement Recommendations Query Parts	A **query refinement recommendation** counts the top occurring terms in variables in the current results and display them to the user. If term is selected, this feature creates a keyword query using the term. A **query part** highlights any keyword query and, if selected, will deactivate/activate it in the current query.

All code indexed on CodeExchange is done by cloning classes on GitHub on a local server and extracting the needed information from the classes using an abstract syntax tree (AST) walker. The overall architecture supporting this process is presented in Figure 22. A List server maintains a complete list of URLs to all repositories on GitHub, and provides a URL to a Java project that is not yet mined to the Code Miner. The Code Miner, composed of a cluster of computers, uses the URL to clone the repository from GitHub. For each class in the cloned repository, the CodeMiner prepares a summary of the code to index by creating an instance of the schema in of Table 4 and Table 5. To do so, it provides values for each field in the schema by walking the class's AST (using the Eclipse AST walker [10]) and visiting each import statement, method declaration, method call, variable name, package name, parent class name, and interface name. When the CodeMiner visits each element, it gathers and records the needed information for the element (as described in of Table 4 and Table 5) and the exact position the element occurs at in the source code file. Further, as the CodeMiner is walking the AST, it counts the number of characters for size, counts the number of import

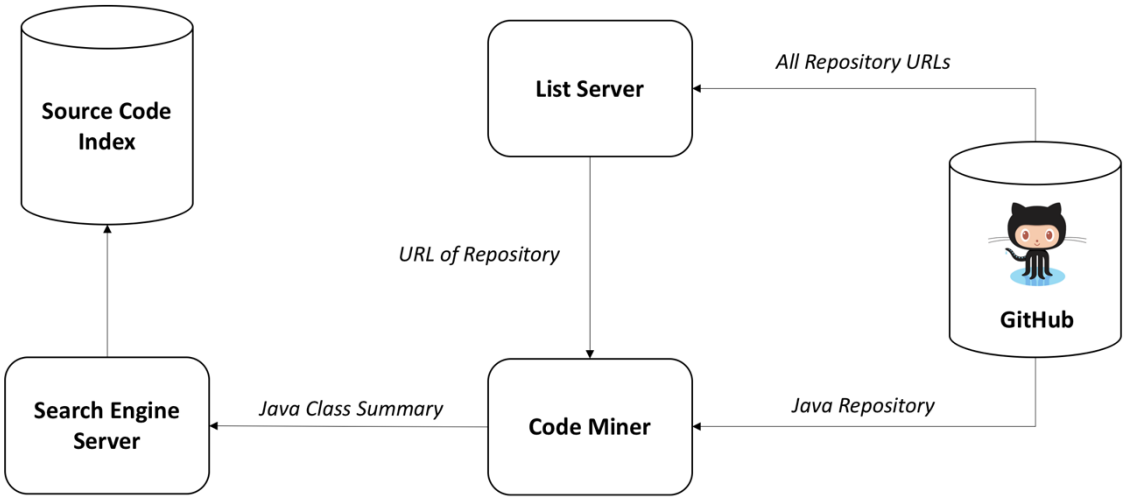


Figure 22. Mining Architecture.

statements, and calculates the cyclomatic complexity by counting all possible branch points in the code, which we count using the typical method of counting all conditional expressions [155]. Once all the values for each field have been obtained, the CodeMiner sends the class summary to the Search Engine Server, where the class will be indexed in the Source Code Index under all the terms occurring in the source code and under each of the values for the fields in the schema.

4.2 CodeExchange Interface

In this section, we demonstrate the features of CodeExchange in its interface. Specifically, we will demonstrate the iterative features, and common but important features in search engines, in the context of a scenario of a programmer looking to build an HTTP server. The beginning of the programmer's search starts at the splash screen of CodeExchange (shown in Figure 23) that resembles the splash screen of many search engines by presenting the user with a keyword text box and an optional advanced search form. The programmer has entered the keywords *http servlet*, presumably looking for source code implementing an HTTP server, and clicked the submit button to issue the query. If the programmer had clicked the "Advanced Search" button, then they would be presented with the advanced search form as shown in Figure 24. The advanced search form allows users to specify queries that involve imports of certain classes, interface properties (extends, implements), location (package or project), method calls (class, method, parameters), and method declarations. Additionally, it allows developers to specify, where appropriate, whether a class or method should be generic or have variable arguments.

Figure 23. CodeExchange splash screen.

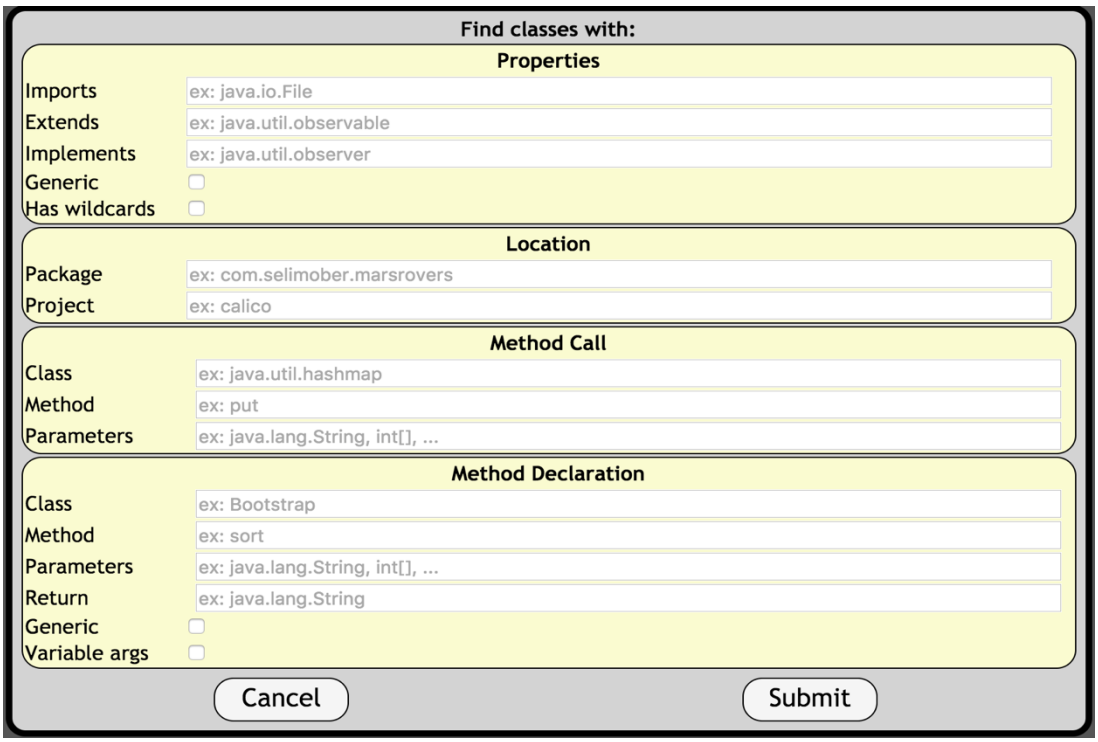


Figure 24. Advanced Search.

The programmer decides to click the submit button on the splash page and is then presented with CodeExchange's main page, shown in Figure 25, which implements the four new features supporting iteratively searching for code. Query parts appear toward the upper left

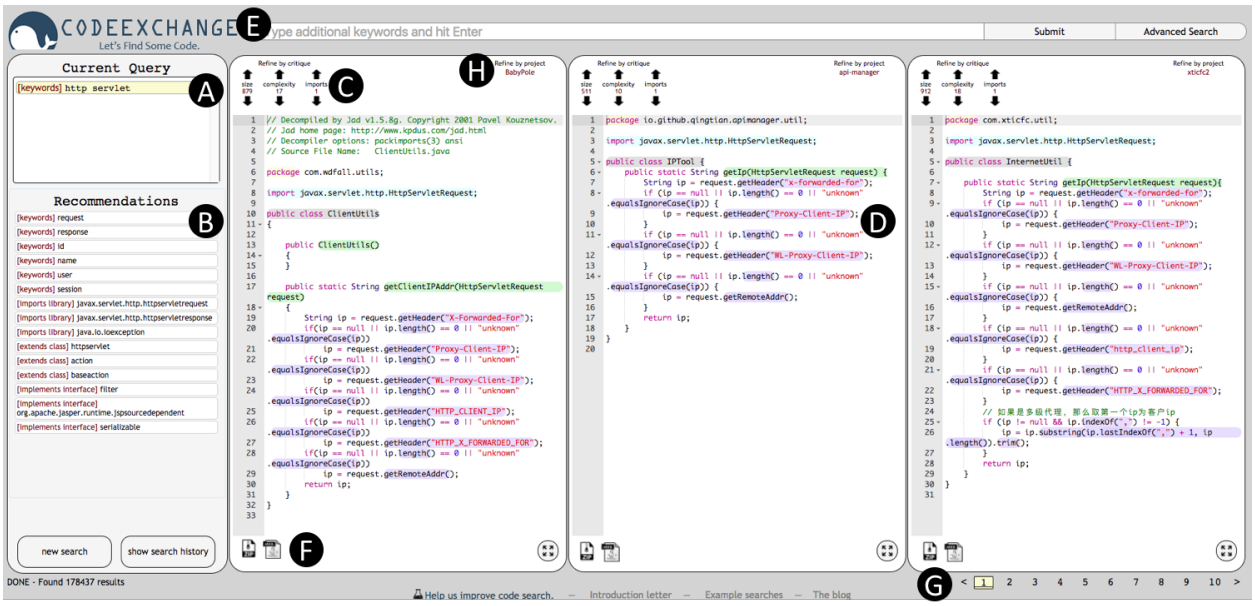


Figure 25. Main Page.

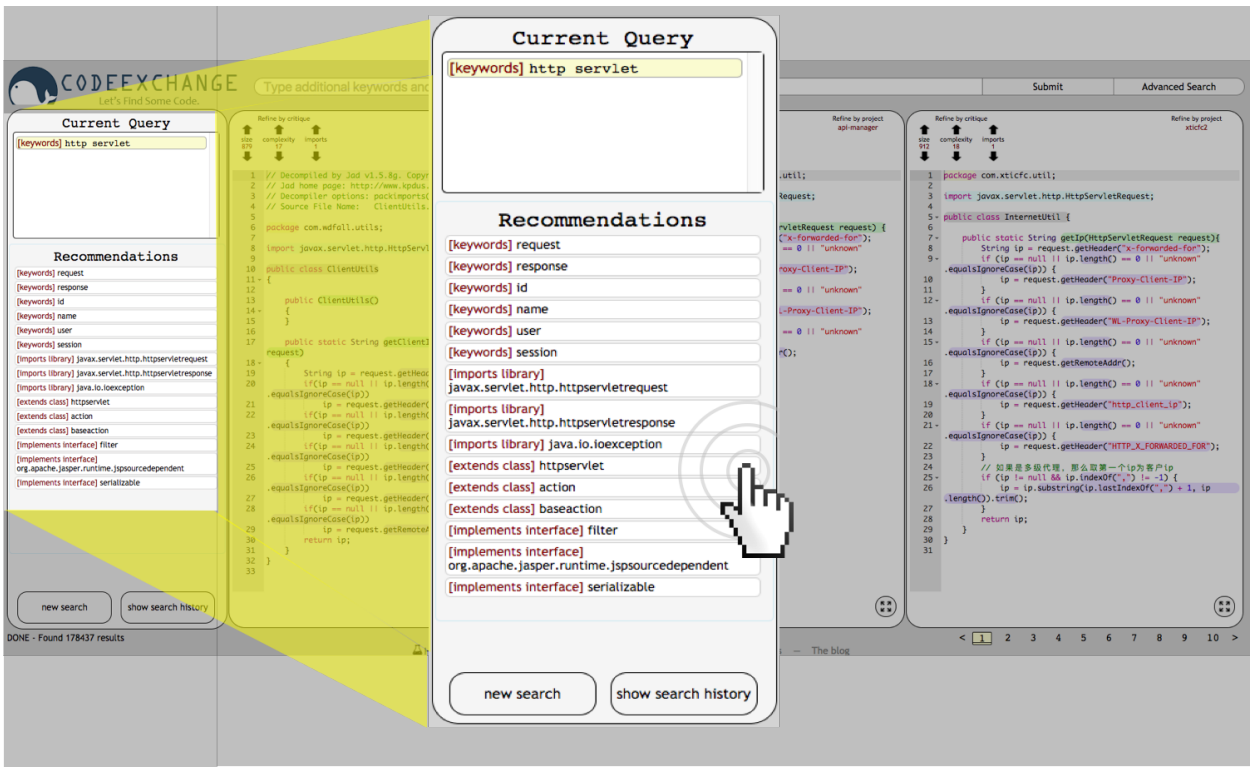


Figure 26. Query Refinement Recommendations.

(A), query refinement recommendations appear toward the middle left (B), critiques appear above each result (C marking where they appear above first result), and language constructs highlight structural properties of the results (D marking where some occur in the second result). In the following sections, we walk-through each of these four features. The other features on the main page are the keyword text box at the top (E), where new keywords can be entered to refine the current query rather than replace it. Buttons to download the result or result's project appear at the bottom left of each result (F), page navigation appears at the bottom right (G), and, finally, at the top right of each result appears its project (H), which, when selected, will refine the query by that project.

4.3 Query Refinement Recommendations

As shown in Figure 26, after the programmer issues the keyword query *http servlet*, CodeExchange recommends the keywords *request*, *response*, *id*, *name*, *user*, *session*. Most of these recommendations have obvious meanings in the domain of HTTP servers, where URL *requests* are accepted, *responses* are returned, and often requests and responses are sent between a client and server during a *session*. Further, the words *name* and *user* are common URL parameters in a HTTP request regarding user accounts.

The other three types of refinement recommendations are the most frequently occurring code imported, parent classes extended, and interfaces implemented among all the results. In this case, the recommendations show that many results import the classes HttpServletRequest and HttpServletResponse in the JavaX library. By clicking either, the programmer could issue a query for code using HTTP requests or responses, which the programmer's server would mostly likely need to do. However, the programmer also sees the recommendation "[extends class] httpservlet" that, when added, would refine the query for code extending the parent class HttpServlet. Since the programmer needs to implement an HTTP server and using the recommendation would return results implementing HTTP servers, of different sorts, the programmer decides to click on that recommendation. Doing so updates the query parts (shown in Figure 27) and issues the next query.

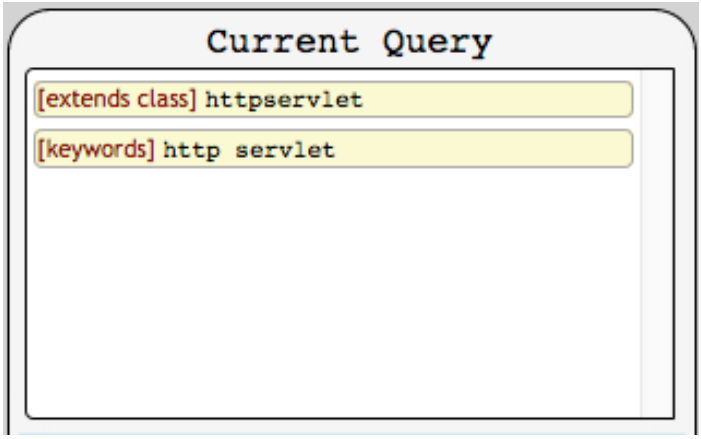


Figure 27. Query parts after using recommendations.

4.4 Critiques

One of the top results that is returned is presented in Figure 28. The critiques present the size (491), complexity (3), and number of imports (1) of the result and clicking the up/down arrow above/below one of the values will dial the value up/down in the next set of results.

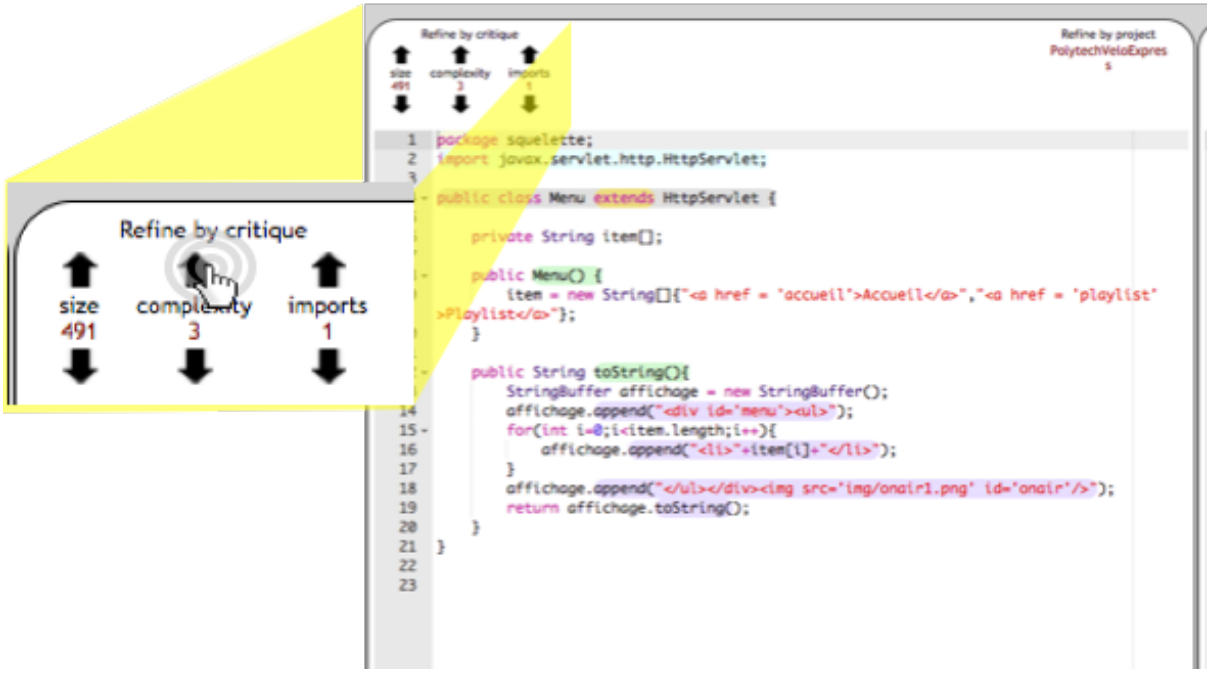


Figure 28. Increase Complexity.

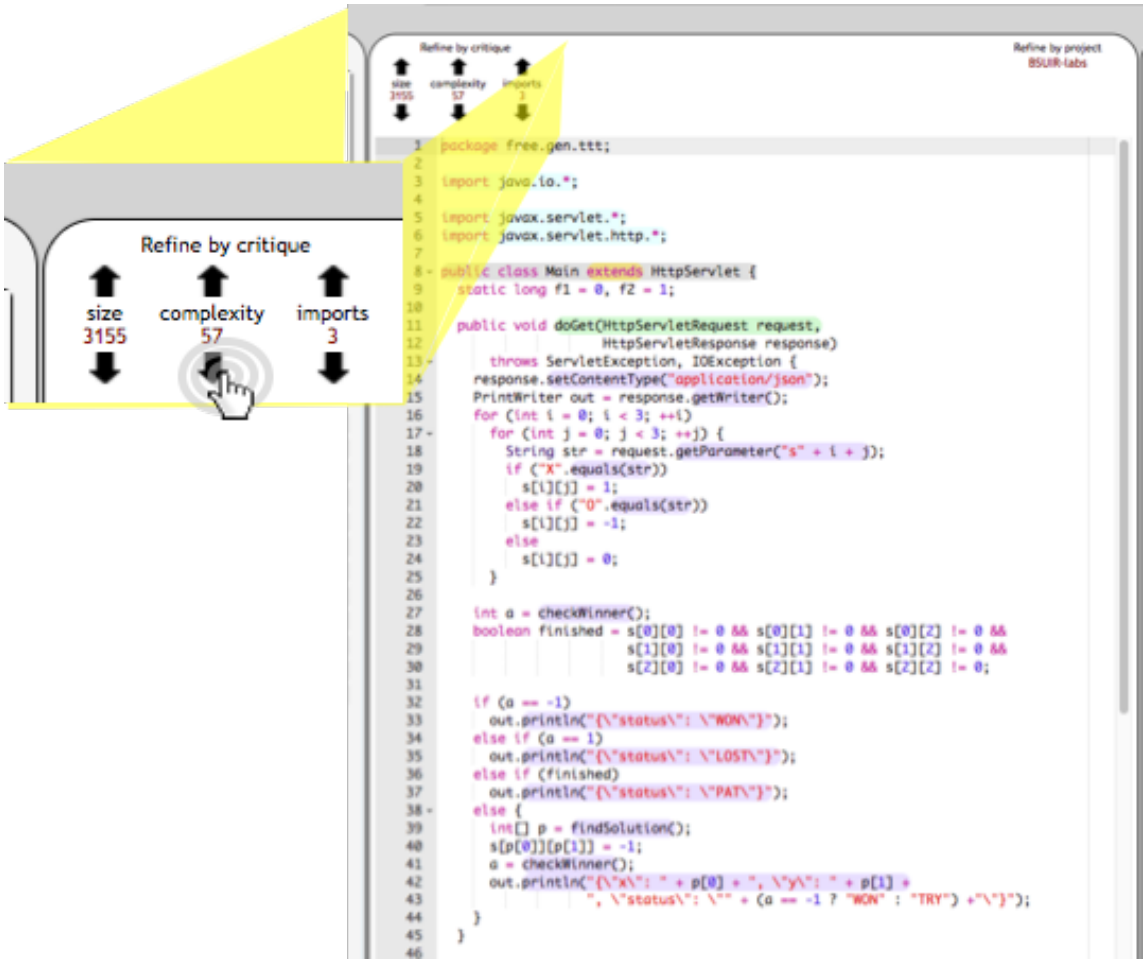


Figure 29. Decrease Complexity.

While the programmer sees that the first result is an implementation of an HttpServlet, they find the code lacking, because the only function it performs is to return a String containing some HTML code. In an attempt to retrieve code doing more, the programmer decides that the complexity should be greater than the unsatisfactory result's complexity (i.e., greater than a complexity of 3) and clicks the up arrow in the complexity critique to do so. After getting the next set of results, the programmer finds the results to be too complicated, with one particular result shown in Figure 29. The programmer issues another query to dial down the complexity to be less than 57 and finds the next results to be more appealing. The current query now is displayed in Figure 30.

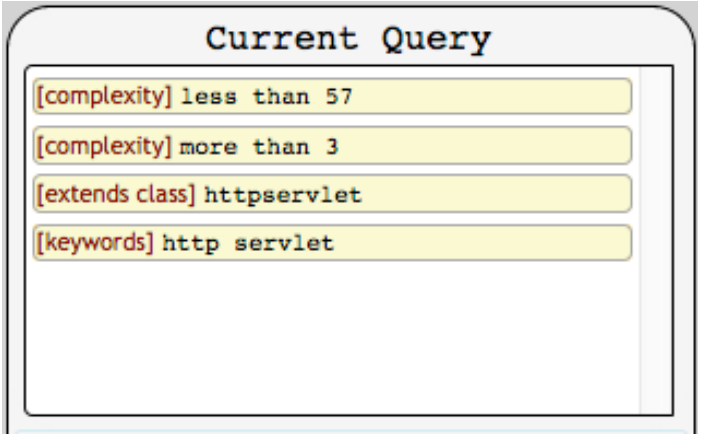


Figure 30. Query parts after using critiques.

4.5 Language Constructs

In the current set of results, the programmer finds a code result, shown in Figure 31, that calls the method getParameter on an HttpServletRequest object, req, that appears to get the value of the name parameter included in a URL request to the server. On reflection, the programmer also wants to create an HTTP server that handles user accounts, so they decide to find more code examples getting parameters from the URL by clicking on the language construct highlighting the method call getParameter. As shown in Figure 32, the current query now specifies to find all code that is topically related to the keywords *http servlet*, extends the parent class HttpServlet, has a complexity greater than 3 and less than 57, and calls the method call getParameter taking a String parameter. The next results now all call the method getParameter passing it different String values.

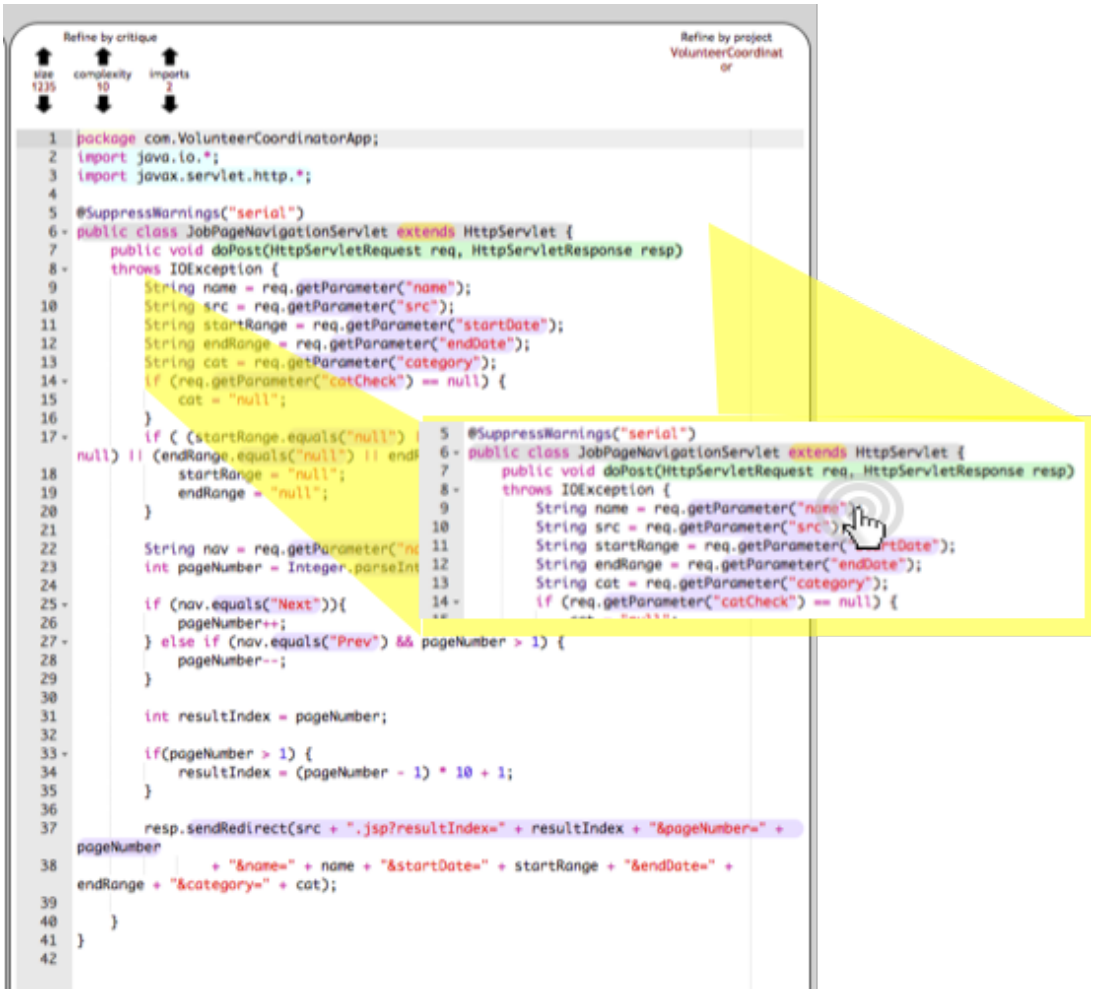


Figure 31. Refine by getParameter Method Call.

4.6 Query Parts

The query parts feature takes the current query, as illustrated in Figure 32, and visually separates the parts by rounded rectangles that can be clicked to toggle off a part to generalize the query or can be clicked to toggle on a part to refine the query. Further, by clicking on multiple parts the programmer can issue different combinations of the parts to quickly try different queries. Visually, a query part is toggled on if it appears yellow and toggled off if it appears white. In Figure 32, all parts of the query are active.

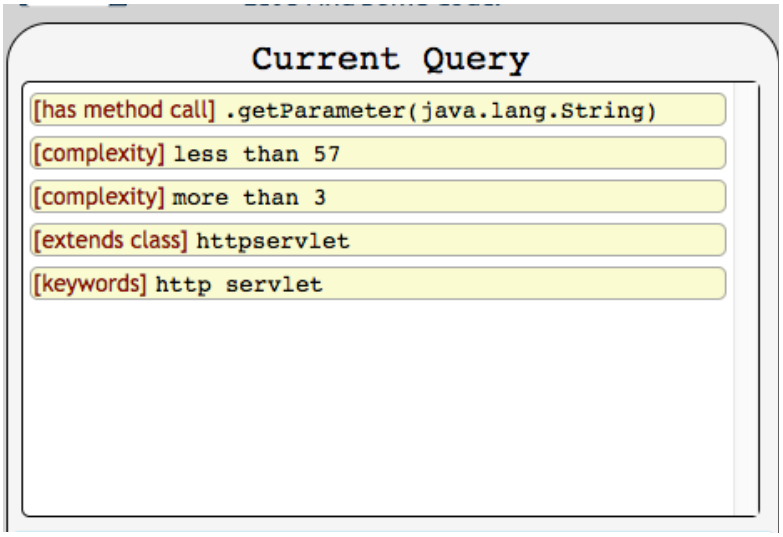


Figure 32. Query Parts.

Continuing the walk-through, the programmer thinks they have a better idea of what they are looking for and decides that they are looking for code that extends the HttpServlet class rather than code being about HTTP servers in general (as specified by the keywords). So, they toggle off the keywords query part by clicking on it as shown in Figure 33 (A). Further, after looking at examples of servers, the programmer decides they are ready to learn from more complicated examples and deactivates the query part specifying all code should have a complexity less than 57, as shown in Figure 33 (B). Now the programmer's query is to find all code that extends the parent class HttpServlet, has a complexity greater than 3, and calls the method call getParameter taking a String parameter.

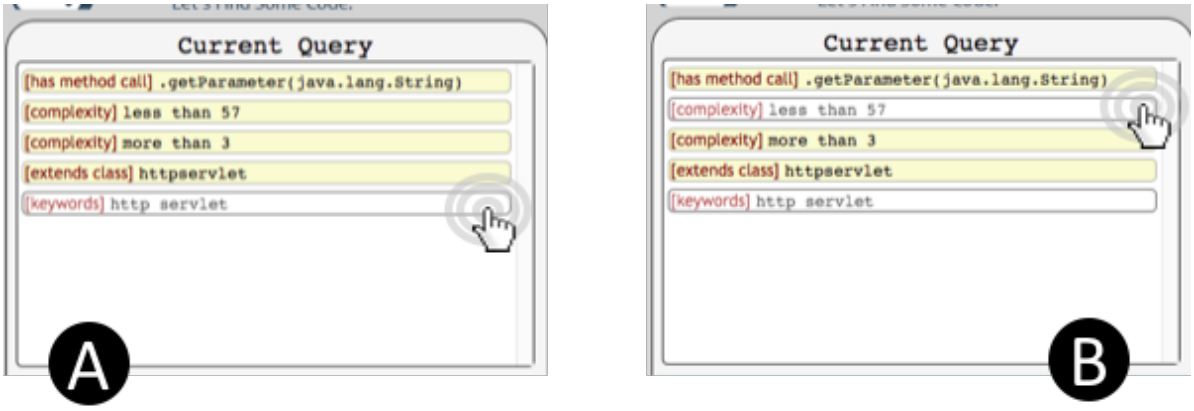


Figure 33. Deactivating Query Parts.

4.7 Walkthrough Discussion

The query created by our hypothetical programmer is a much more heterogeneous and specific query than keywords and was created by iteratively selecting aspects of the results. While it is possible to issue such a query upfront with an advanced query form, it is very difficult to do so because programmers often do not know, as discussed in previous sections, exactly what code they want at the start of a search, which is further evident by how scarcely advanced search is used in general [7]. In our example, it was not until the programmer saw the aspects, such as complexity of 57 in the context of an HTTP servlet example, did they know they wanted results with less complexity. Further, it was not until the programmer saw the method call to retrieve and parse parameters for user data that the programmer knew they wanted results with the getParameter method call. The features of CodeExchange are explicitly designed to support such moments, that is, when the programmer learns more about what they want from aspects of the results and can explicitly bring some of those

aspects into a new query with CodeExchange's features, without trying to encode them as keywords.

4.7 Preliminary Evaluation of CodeExchange

As a preliminary evaluation of the effects of supporting iteration, we wanted to see what, if any, effect CodeExchange may have on the search process. We took a two-pronged approach to do so. First, to evaluate if programmers would use the iterative features in a real-world setting at all, we conducted a field study by releasing CodeExchange to the public and recording all activity over approximately a 20 day time span. Second, given that field studies can generate much noise, due to a lack of control on why people are searching, we conducted a lab study among 6 developers evaluating the performance of CodeExchange versus GitHub's code search engine (which does not explicitly support iteration). In this way, the control of the lab study compensates for the noise of the field study, and the field study compensates for the less realistic setting in the lab study. If both the lab study and field study tell a similar story, then we can be more confident in that story (a common technique used in social sciences referred to as triangulating data from more than once source [166]).

4.6.1 Instantiating the Index for Experiment

To populate CodeExchange's index for our preliminary experiments, we identified the URLs of 602,244 repositories that included only Java code, between February 4, 2014 and February 12, 2014 using GitHub's API. From the 602,244 repositories, we mined

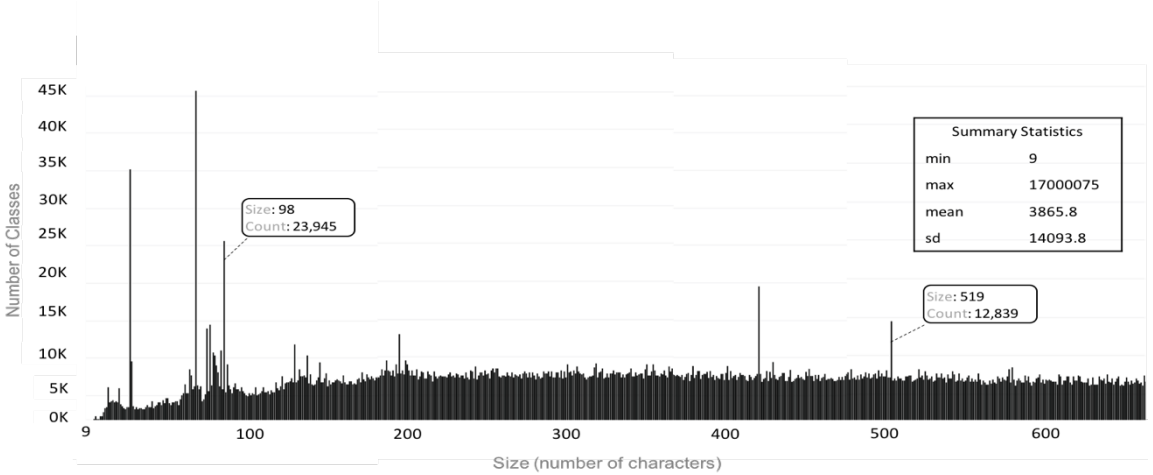


Figure 34. Top of Size to Number of Classes Graph.

approximately the first 300,000 projects, with the resulting index consisting of about 10M classes, 150M methods, and 253M method calls. We stopped mining a little over 10M classes because of a noticeable delay in processing queries on our servers (unacceptable when users expect queries to be processed in a few seconds). However, we also thought 10M classes allowed us to closely approximate an Internet scale code search engine when compared against the sizes of the Internet code search engines Koders (600K), Krugle (3.5M), and Google Code Search (2.5M) [108].

While 10M classes gives us a reasonable *quantity* of code for applying our code search techniques at Internet-scale, we wanted to get an indication of how the 10M classes are spread out over kinds of examples. The "shape" of the code indexed will give us a better understanding of the range and variety of code on the internet, but also serves to frame our findings so that others may reproduce the results on similar looking repositories. To get an indication of the distribution of the kinds of code we indexed, we graphed the code indexed across size, complexity, number of imports, and kinds of imports.

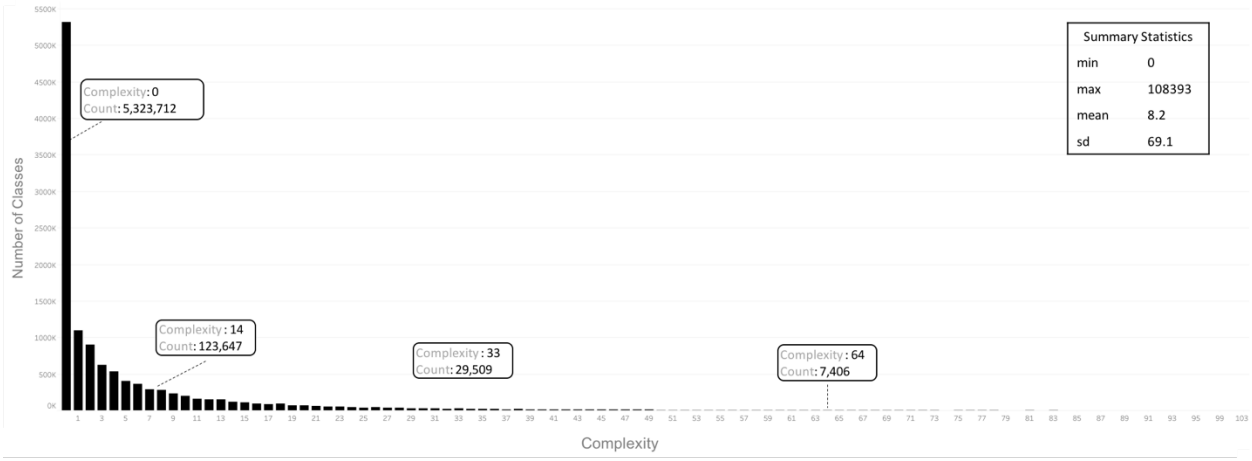


Figure 35. Top of Complexity to Number of Classes Graph.

We found that that the classes we indexed cover a wide range of sizes (min of 9 characters, max of 17,000,075 characters, mean of 3865.8 characters, and a standard deviation of 14,093.8 characters). Shown in Figure 34 is the top portion of a graph mapping size to the number of classes of that size. In general, we find that as size increases there is a slow linear decline in the number of classes having that size, where this slow decline extends off Figure 34 all the way out to a class with 17,000,075 characters.

We found that the classes we indexed cover a wide range of complexities (min of 0, max of 108,393, mean of 8.2, and a standard deviation of 69.1). The top of the graph mapping complexity to the number of classes with that complexity is presented in Figure 35. As complexity increases the number of classes with that complexity decreases sharply and appears to follow an inverse log curve with a long tail. However, we find that the tail decreases slowly and that there are still thousands of classes mapped to many of its points (e.g., 7,406 classes have a complexity of 64).

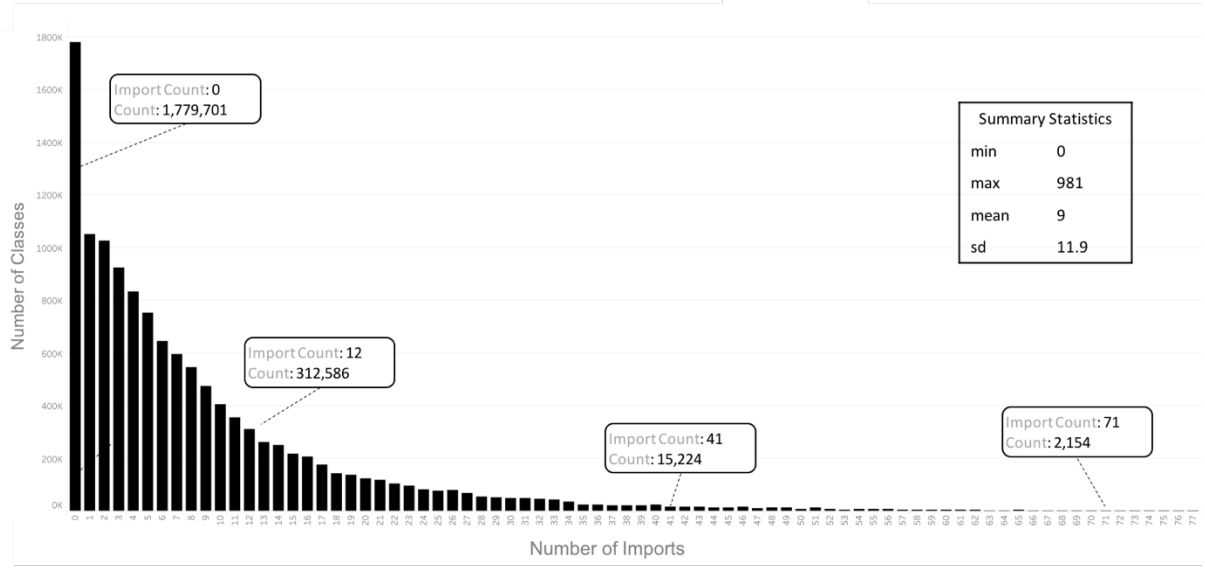


Figure 36. Top of Number of Imports to Number of Classes Graph.

We found that the classes we indexed cover a wide range of number of imports used (min of 0, max of 981, mean of 9, and a standard deviation of 11.9). The top of the graph mapping number of imports to the number of classes with that quantity of imports is presented in Figure 36. As the number of imports increases the number of classes having that quantity drops sharply and, like complexity, follows an inverse log curve with a long tail. However, the long tail decreases gradually and there are still thousands of classes on many of its points (e.g., 2,154 classes have 71 imports). The tail continues off the figure and stops at a class containing 981 imports.

Lastly, we looked at the distribution of the kinds of imports in classes, which, to some extent, would give us an idea of the variety of functionality of the code indexed (given that the functionality of the different libraries imported represent some of the functionality of the class importing those libraries). To conduct this analysis, we created a tree map (as shown

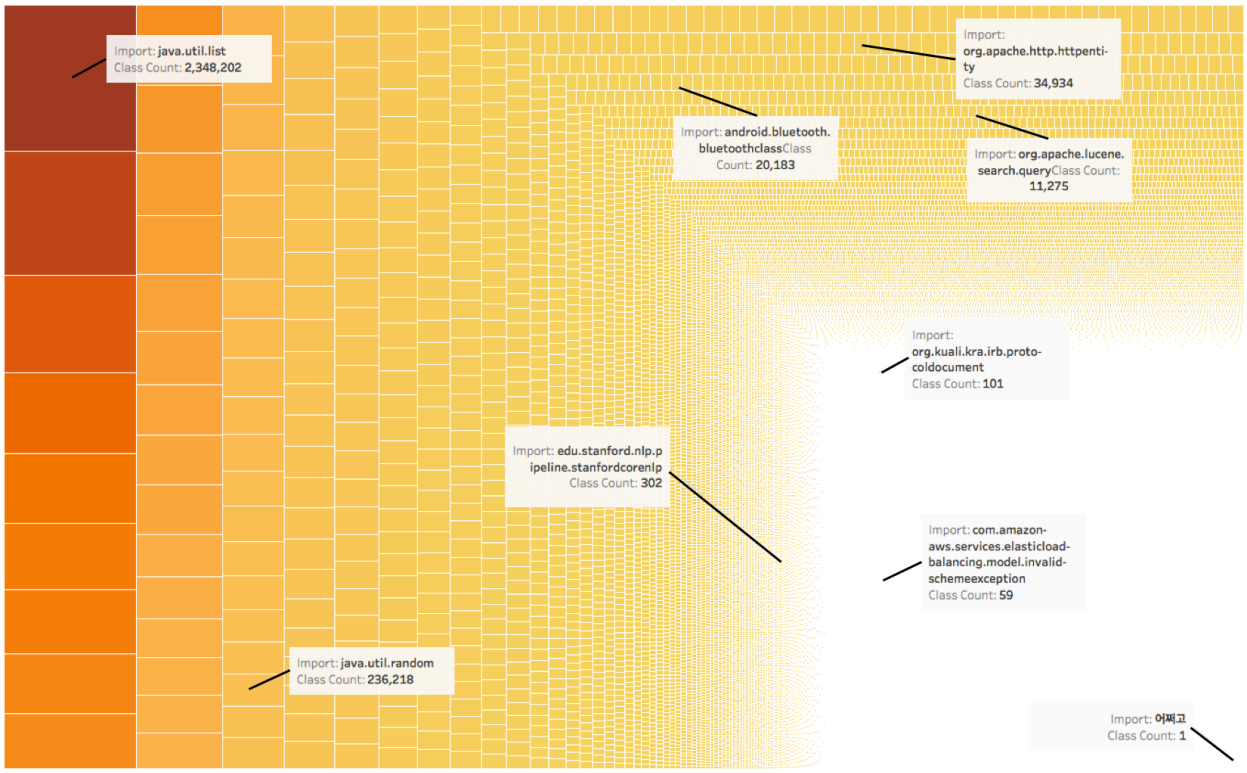


Figure 37. Tree map of Number of Imports Used by a Class.

in Figure 37), where each rectangle represents an import statement and the rectangle's size represents the number of classes that have that import statement. Altogether, there are 2,737,959 rectangles on the tree map, which means that the 10M classes indexed serve as an example, of some kind, of using one or more of the classes from the ≈3M imports. We annotate some rectangles to show the trend of how the number of classes decreases with the size of the rectangles. The tree map's rectangles steadily decrease in size from left to right and top down, which indicates some imports are more common than others (Java standard libraries being among the most common), but the imports represented by pixel sized boxes (toward the bottom right) still have hundreds of classes using them and even the imports that cannot be rendered (white area) have 10s to 100s of classes using them as well.

From the above graphs, we make two observations about the code indexed. First, the code indexed has a wide range of values for size, complexity, and number of imports as well as wide range of kinds of imports. Second, while there are more common values for size, complexity, number of imports, and kinds of imports, the relatively less common values still range in the hundreds to thousands. This suggests that when the programmer is looking for less common kinds of code, they often will still have hundreds to thousands of results that could be candidates for what they might be looking for, which in part might be due to indexing 10M classes. With additional code indexed, the overall variety might increase yet more.

4.6.2 Field Study

To obtain an early impression as to whether supporting developers in iteratively searching for code may help in a real setting, we released CodeExchange to the public and analyzed how it was used. On July 31, 2014, we posted a pre-announcement to a small group of developers that our research group interacts with on a regular basis. We made this pre-announcement to vet possible problems before we made the more public announcements. As we did not experience any serious issues, we moved on quickly and posted several brief announcements of CodeExchange's availability on forums (Java specific when possible) for programmers such as those on Reddit [154], JavaRanch [156], and Hacker News [148]. Our post on a Java forum on Reddit, for instance, read as follows:

CodeExchange – a new Java code search engine

(codeexchange.ics.uci.edu)

We received the most activity in the week we posted on Hacker News (August 4th) and Reddit (August 6th). Other forums were less effective in drawing attention. CodeExchange logged all visitor behavior by assigning each first-time visitor a unique anonymous id that is stored as a cookie in their browser. This is especially important to track their return behavior. Every time a CodeExchange feature was used, an entry was made in the logs, detailing which feature was invoked, any input the user provided, and the date and time of feature use.

From July 31 to August 19, 2014, we observed more than 4,000 visitors to CodeExchange. Approximately 2,000 of the visitors were robots, which we discarded immediately. About 1,000 visitors just typed in keywords on the splash screen, and subsequently never did anything else. We discarded these from our analysis as well, leaving roughly 1,000 unique visitors to CodeExchange. Figure 38 plots these users' visits (a visit is defined as the sequence of user events starting from the splash screen) on a graph, where the y-axis is the identifier of each visitor and the x-axis is the date and time the visit began. As the visitors came from many different time zones, we treated all their local search times as PST time (this is why there is a search on August 20, even though we stopped collecting data on August 19 PST). Black dots denote visitors who visited once. Blue squares count "return visitors" who are visitors that after at least one hour had elapsed visit CodeExchange again (blue lines connect searches by the same visitor). Red circles around a blue square or a black dot indicate that a copy or a download occurred during that visit. For example, visitor 2747 visited on August 6th, 11th, and 12th, with the latter two visits involving copies or downloads of actual code.

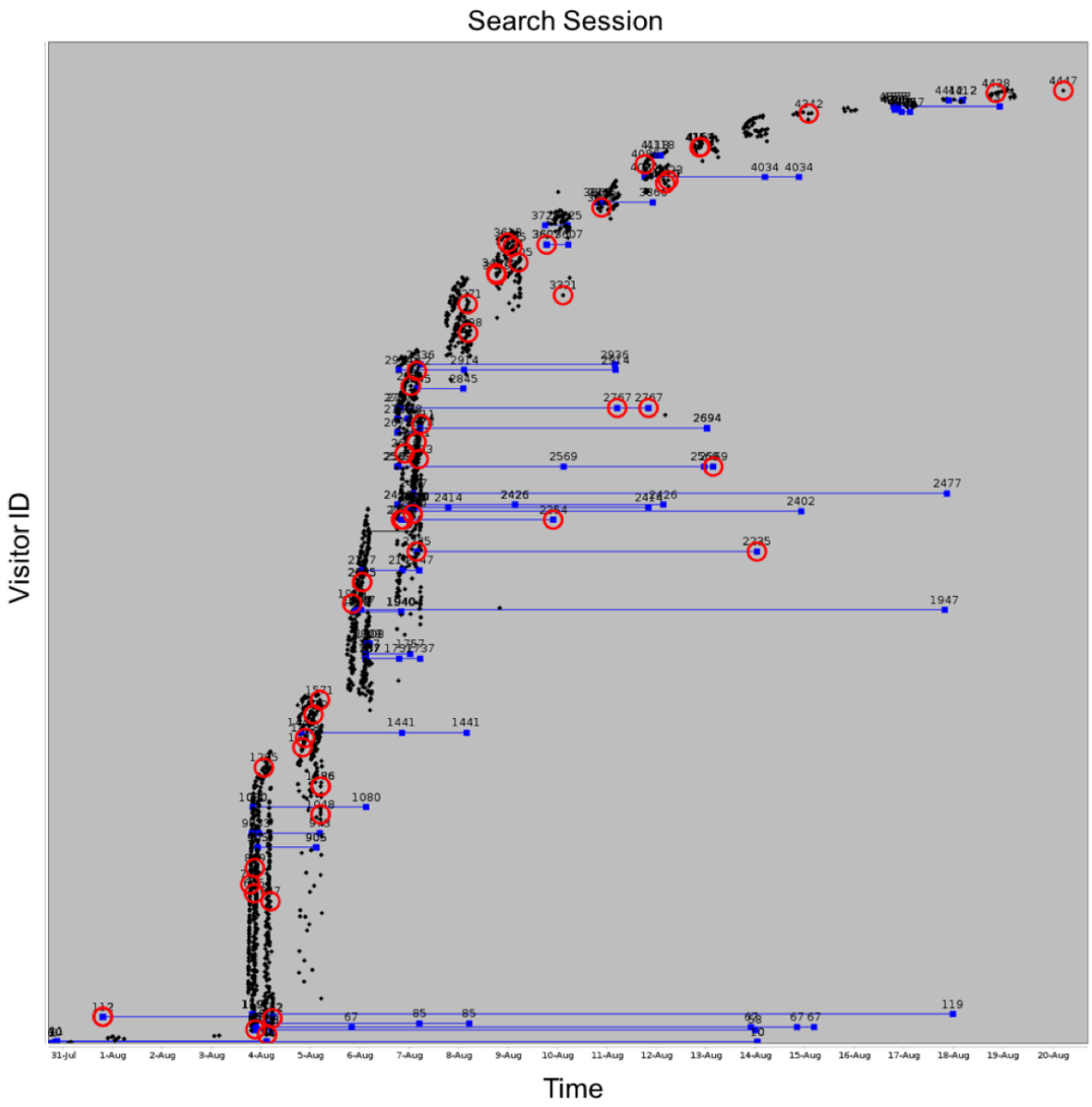


Figure 38. Visitors Over Time to CodeExchange.

Overall, there were 242 return visitors and 895 visitors used the advanced features of CodeExchange.

For purposes of understanding the search behavior of the users to find code during the field study, we attempted to focus our attention on users who were genuinely searching for code rather than simply playing with CodeExchange itself. To do so, we identified visits from

users who copied or download one or more results, since copying or downloading is one indication the user is genuinely looking for code [7]. However, given that many of the reasons programmers search are purely to learn or remember (as discussed in Section 2), which does not require coping/downloading, this choice is conservative.

Out of about 1000 visitors, 44 users copied or downloaded 63 classes/projects (40 copies, 20 project downloads, and 3 file downloads) in 45 visits. The queries across the 45 visits cover a broad range of topics (e.g., travel agency code, Twitter API usage, convex hull algorithms). To analyze the visits for search patterns, we hand coded all the queries during a visit as a sequence of query modifications to represent the user's search. Shown in Figure 39 and Figure 40 are the encoding of each user's search, where the UID column (first column) is the user's identifier, SID is the visit identifier, and the third column is an encoding of the search. The encoding of the search is a sequence of columns (each column has a black background and white text), where each block in a column represents a query refinement, each column represents a query, and all columns represent the set of queries that constitute the search. The block encoding is **K** for keywords, **R** for recommendations, **Q** for critiques, **C** for language constructs, **P** for project refinement, "*" annotations for toggling a query part on, **H** annotations when using CodeExchange's history feature to create a query (the history feature saves previous queries issued by the programmer and can be reissued later if selected), **A** for a query using the advanced query form, and empty for times when all query parts are deactivated. Further, a column that is one block smaller than the preceding column means a query part was toggled off. Lastly, we encode some important events between columns, which are **SP** for when the user is at the splash screen (counted as a start of a

search), **NQ** for when the user hits the new query button, **copy** for when the user copies part of the results one or more times, **DP** for when the user selects the download project button, and **DF** for when the user clicks the download file button.

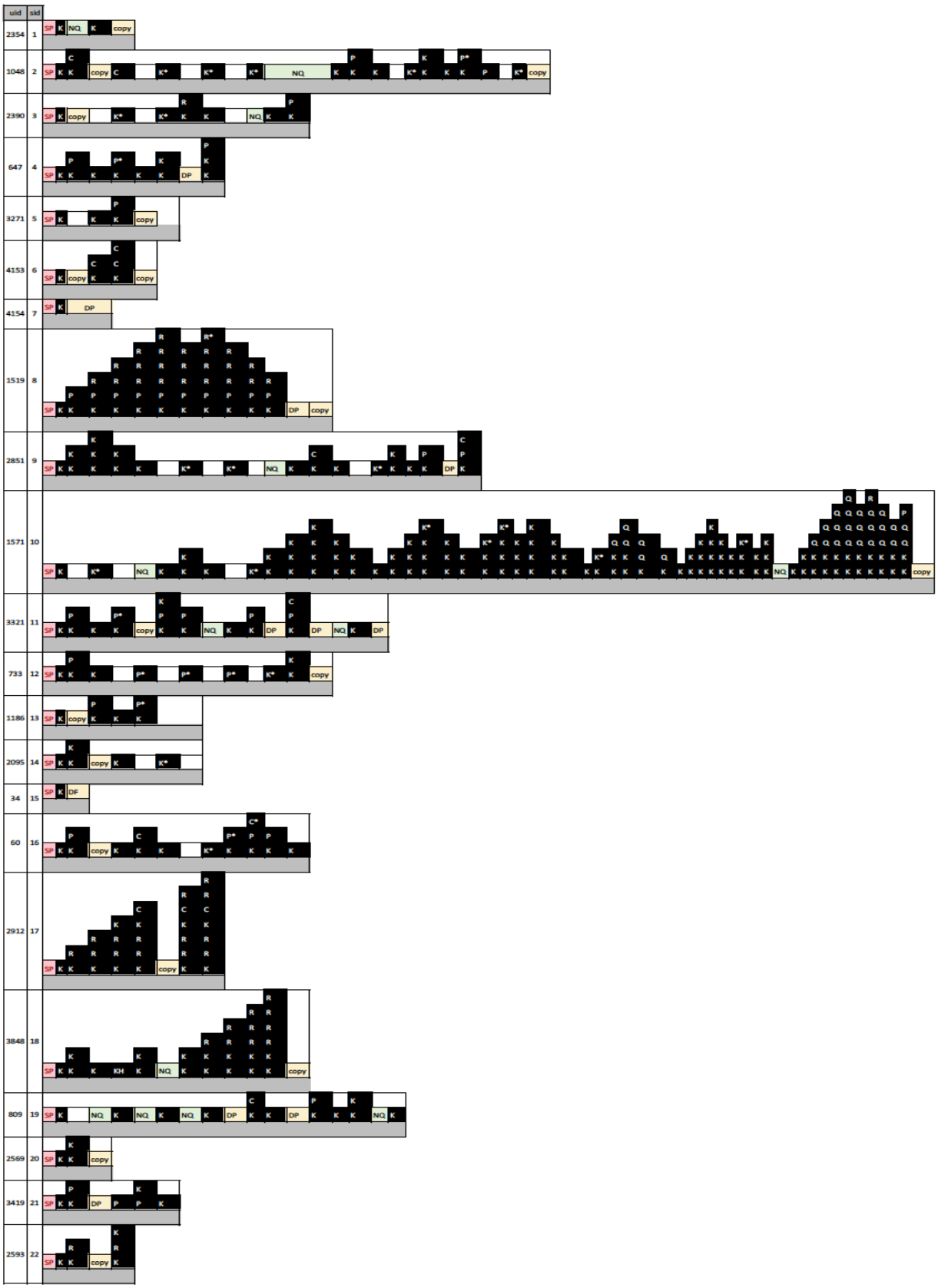


Figure 39. Search Patterns from Field Study, Part1.

Figure 40. Search Patterns from Field Study, Part 2.

					[R] exporttype		[K] source			
				[R] _log	[R] _log	[R] exporttype	[R] exporttype			
			[R] source	[R] source	[R] source	[R] _log	[R] _log	[K] source		
		[R] extends	[R] extends	[R] extends	[R] extends	[R] extends	[R] extends	[R] exporttype	[R] exporttype	
		basetransformer	basetransformer	basetransformer	basetransformer	basetransformer	basetransformer	[R] _log	[R] _log	[R] _log
	[P] core	[P] core	[P] core	[P] core	[P] core	[P] core	[P] core	[P] core	[P] core	[P] core
[K] switchyard	[K] switchyard	[K] switchyard	[K] switchyard	[K] switchyard	[K] switchyard	[K] switchyard	[K] switchyard	[K] switchyard	[K] switchyard	[K] switchyard

Figure 41. Content of Search Number 7.

We found that people searched in a variety of ways, often using the features we designed for iterative search, but also sometimes just using a keyword search to find code. For example, Figure 41 presents a more detailed view of search 10. In this search, it appears that the programmer is looking for code using the JBoss SwitchYard framework that supports the full development process of developing and releasing service-oriented software [164]. To search for code, the programmer issues keywords, refines by a project, adds a recommendation to find all code extending a parent class, adds three additional keyword recommendations one by one, deactivates a keyword recommendation but then adds the same keywords in manually, deactivates two recommendations, and then copies code which appears to use the JBoss logging library in the context of the JBoss SwitchYard framework.

We analyzed each of the 45 searches and found that 41/45 (91%) of these searches iterated (i.e., submitted more than one query) and that CodeExchange's iterative features were used in 34/45 (75%) of searches and in 34/41 (82%) of the searches that iterated. Among all the searches that used CodeExchange features, query parts were used the most (28/34 (82%)), deactivations in 28 out of 34 (82%), activations in 16 out of 34 (47%), language constructs in 14 out of 34 (41%), recommendations in 12 out of 34 (35%), and critiques in 3 out of 34 (8%). Among the 23 searches using some combination of language constructs, query refinement recommendations, and critiques, we find language constructs used in 14 out of 23 (60%), recommendations in 12 out of 23 (52%), and critiques in 3 out of 23(13%).

Further, we also looked at the frequency of use of CodeExchange's features as compared to keywords. Among all queries, we found that 388 (55%) keywords were used and 315 (44%) iterative features were used (149 query parts, 89 recommendations, 42 language constructs, and 35 critiques). The frequencies tell us that keywords were still the most frequently used feature in the field study (p=.005 with $\chi 2$ on a 2x1 contingency table).

While we hoped to see more usage over the approximate 20-day time span of CodeExchange, these early findings show that CodeExchange's iterative features were used in a majority of searches in a real-world setting and composed almost half the features used across all searches, suggesting they may indeed be helpful for the programmer in creating the next query and are worth further study.[1]

4.6.3 User Study

While the field study allowed us to collect data in a realistic setting, a field study such as ours necessarily leads to noise in the data because there is an absence of the reasons driving the programmers' searches. As such, we were left to guessing what the programmer was searching for by looking at the code copied or downloaded and left with a heuristic to define when a search is "genuine", but heuristics are approximate and necessarily will exclude and include (in)valid searches. Therefore, we additionally conducted a controlled user study. We recruited six participants with professional development experience. Two were working

[1] We note that our numbers vary from those found in [75], but we again reach the same conclusions despite the possibility of a minor error, so we are confident in our conclusion here and in [75].

as developer interns, one was working a full-time development job, and three were graduate students with two or more years of experience working as a professional developer. We had the participants complete code search tasks with either CodeExchange or GitHub's code search engine to determine what difference, if any, supporting iteration had on the programmers' search. We chose to compare CodeExchange with GitHub's code search engine because both index Java code on GitHub and GitHub supports a more traditional approach of searching (i.e., keywords and results as links to source code), as shown in Figure 42.

We constructed six independent search tasks, each inspired from code copied or downloaded from the field deployment to make them more realistic, for each participant to .complete. To create the tasks, we read the code that was copied or downloaded during the field deployment, looked up documentation to further determine what the code does, and developed a general task description for finding similarly suitable code. Table 6 and Table 7 presents the six tasks. During the experiment, each participant was instructed to find the code that they felt best satisfied the task and to paste that code into a provided Google survey form.

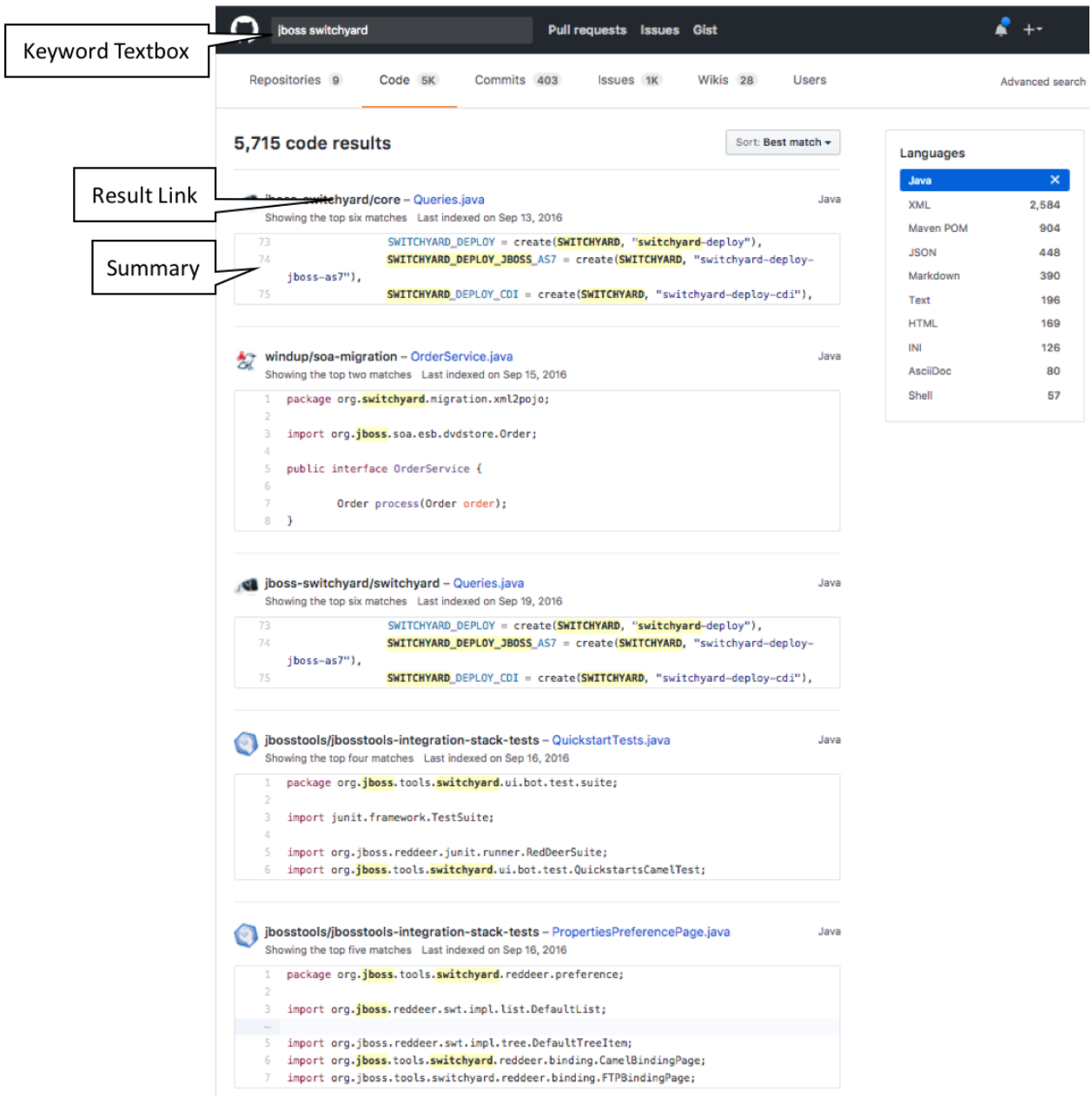


Figure 42. GitHub's Code Search.

Table 6. User Study Tasks, Part 1.

Task Name	Description
Minecraft	**Background:** Minecraft is a game where people build virtual worlds with zero or more people. These worlds are often hosted on servers. Bukkit is a new platform for creating and modifying Minecraft worlds. These worlds are configured with what are called YAML configuration files. Part of the configuration files specify the permissions of users in the Minecraft world. These users can be assigned to groups where each group has a set of permissions. **Task:** Find Java code that will return all the groups that are in the YAML file for a Minecraft world created or modified with the Bukkit platform.
JBoss	**Background:** JBoss is an open source Java application server maintained by Red Hat. There is a JBoss API for creating different kinds of references to Java objects. References point to objects in Java and based on their "strength" the Java garbage collector treats them differently. They types of strengths are: Phantom, Weak, Soft, and Strong. **Task:** Find code to create a Phantom reference using the JBoss API.
Android	**Background:** Android applications can manage phone power usage by creating what are called wake locks. There are different kinds that will wake the phone up and make the screen and keyboard brightness at their highest levels. While others will wake the phone up but dim the screen and keyboard. **Task:** Find Android code that will allow your application to obtain a wake lock so that the phone is awake and sets the screen and keyboard to full brightness.

Table 7. User Study Tasks, Part 2.

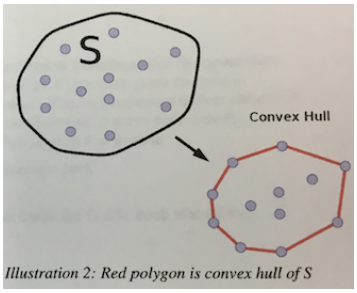
Task Name	Description
RaspberryPi	**Background**: A Raspberry Pi is a small computer that has been used for many purposes. It has a general-purpose input and output port (GPIO) that can be used to hook it up to external devices. Illustration 1 to the right shows what the GPIO looks like (the port with 26 pins standing up). *Illustration 1: GPIO* **Task:** Find code that listens for input and output changes in the Raspberry Pi GPIO.
ConvexHull	**Background:** In Illustration 2, imagine the points of S as being pegs; the convex hull of S is the shape of a rubber-band stretched around the pegs as is shown by the red polygon in Illustration 2.  *Illustration 2: Red polygon is convex hull of S* *Formal definition*: The convex hull of S is the smallest convex polygon that contains all the points of S. **Task:** Find code to return the convex hull of a given set of points.
Oracle	**Background:** A Connection Pool is a cache of database connections maintained in memory so that the connections can be reused when the database receives future requests for data. Connection pools are used to enhance the performance of executing commands on a database. Various parameters such as number of minimum connections, maximum connections and idle connections can be set to make sure the connection pool works well according to the environment it is deployed to work in. The Oracle thin JDBC driver makes it very easy to use a connection pool. **Task:** Find code that will allow you to create a connection pool with the Oracle thin JDBC driver

Table 8. Time (minutes) to Complete Tasks by Search Engine.

	Tasks						Averages	
ID	**Minecraft**	**JBoss**	**Android**	**RaspberryPi**	**ConvexHull**	**Oracle**	**CodeE.**	**GitHub**
1	7.62	10.87	4.88	9.83	7.58	7.72	6.69	9.47
2	12.23	**1.42**	2.80	**17.42**	9.87	10.27	9.70	8.30
3	11.00	8.60	**11.27**	**20.00**	2.92	9.75	8.39	12.78
4	12.55	3.33	4.38	2.10	3.08	**1.20**	2.21	6.67
5	3.03	**2.00**	1.95	6.60	3.33	**1.60**	2.77	3.40
6	**20.00**	11.02	10.60	5.25	4.07	2.85	6.37	11.56

Before the experiment began, CodeExchange and GitHub's code search engine were explained in detail to each participant. For CodeExchange, we walked through all the features (discussed in Section 4 above), and for GitHub, we showed each participant how to use keyword search, advanced search, sorting functions, navigation to a project page of a code result, as well as how to filter results by the Java language.

The experiment started with each participant given their own room with a computer setup making either CodeExchange or GitHub available, depending on the task. Each of the six tasks were completed by all participants and completed sequentially, with 20 minutes allotted for each participant to complete their task. After completion of the task, the participant was given a new task and switched to the other search engine. Half of the participants started with CodeExchange and half with GitHub to help reduce the risk of ordering effects. We recorded the screen and logged participants' actions. From this, we calculated task time from

the moment they began interacting with the search engine by typing search terms until they pasted the found code into the Google survey form. Finally, at the end of each experiment, the experimenter conducted a semi-structured interview focusing on the participant's perceptions and their use of features during the study.

Table 8 presents our main results, listing times for each task. Grey cells indicate GitHub times and white cells CodeExchange times. On average, participants completed tasks 3 1/2 minutes faster with CodeExchange than with GitHub (7.23 versus 10.44 minutes). However, there were several outliers in the data. For example, in two cases using GitHub, participants were still looking for code at the end of 20 minutes. To remove these and other outliers, we performed symmetrical truncation [62], an accepted technique for trimming outliers, and removed the two slowest and two fastest times from each search engine (denoted by a double underline in *Table 8*). After removing outliers, we found participants were about 2 1/2 minutes faster on average with CodeExchange (5.5 minutes with CodeExchange versus 8 minutes with GitHub). Performing a one tailed t-test (chosen because CodeExchange and GitHub task completion times are separate groups having equal variance) on the truncated results, we found that the time difference is significant (p = 0.0268). Further, with a power level of .99, we found the effect size to be large (d = 1.67). This means that CodeExchange had a large effect on the user's task times (Cohen's d greater than .8 is considered large [20]).

Next, we looked at the patterns of search generated by the lab participants using CodeExchange. The design of the study gave us three patterns per task, where each of the three patterns was generated by a different participant. All the patterns for each task are

presented in Figure 43 and Figure 44. Each search pattern is encoded as in the field study, however each search is preceded by the name of the task it was produced for and an identifier. Further, the blocks identifying a

Figure 43. Search Patterns in Lab Study - Part 1.

Figure 44. Search Patterns in Lab Study - Part 2.

copy also include the name of the class that was copied (sometimes a number is included to distinguish different code with the same class name).

From the patterns, we find that 17 out of 18 (94%) involve the participants iteratively searching for code (only JBoss 4 did not). Further, the iterative features of CodeExchange often helped participants in identifying the code they copied. In 15 out of 18 (83%) tasks, participants iterated with CodeExchange features. Among the tasks using CodeExchange features, query parts were used in 86%, language constructs in 80%, recommendations in 13%, and critiques in 6%. In 3/18 (16%) tasks the participants did not use any iterative features, relying on keywords only. Among all queries, we found that 263 (65%) keywords were used and 141 (35%) iterative features were used (81 query parts, 6 recommendations, 46 language constructs, and 8 critiques). The frequencies tell us that keywords were still the most frequently used feature in the user study (p=1x10^{-9} with $\chi 2$ on a 2x1 contingency table), but the iterative features were still used to find code for the majority of tasks. We further note that the searches that used the iterative features were 1.23 minutes faster than searches using only using keywords.

Somewhat unexpectedly, we observe that many participants 11 out of 18 (61%) did not stop their search after they found code (indicated by a copy) for their task. This behavior might be explained by the participant not knowing exactly what they are looking for and exploring to be sure of their copied code. For example, the participant could simply be wanting to make sure that there is not any "better" code and try some more queries to find other code to

Table 9. Average GitHub Search Behavior.

	Minecraft	JBoss	Android	RaspberryPi	ConvexHull	Oracle
Queries per User	10.67	4	6.67	9.67	2.33	4.67
Terms per Query	2.97	2.92	2.1	2.45	2.14	4.36
Terms Deleted per Query	1.03	1.27	1	1.25	0.67	0.62
Terms Added per Query	1.19	1.18	0.89	1.21	0.83	0.85

compare against (e.g., Minecraft 1 and 13), or, the participant might be searching for code candidates, copy code they think might be better than the previous copied code and search until they are satisfied (e.g., Minecraft 7 and RaspberryPi 5).

We also analyzed how the participants used GitHub. In general, we find that participants iteratively search with GitHub (88%) and sometimes issued more advanced queries (25%) using the advanced query form to restrict a query by repository, file extension, user, or the kilobytes of the repository. Further, we analyzed the average number of queries, terms, and modifications of queries per task and present the results in Table 4. We find that the average number of queries spans from approximately 2 to 10, contains about 2 to 4 terms, and constructing the next query often involves delete or adding about 1 term from the current query. Lastly, we found that the participants also continued to search for code after they copied code in 33% of the tasks.

An example of a search on GitHub by participant 3 for the Android task is: *wake lock android*, *newWakeLock android*, *newWakeLock*, *newWakeLock brightness*, *PowerManager*

newWakeLock, *PowerManager newWakeLock user:android*, and *PowerManager newWakeLock*. In this example, the participant is incrementally adjusting each new query and starts their search from a general keyword query *wake lock android* to a keyword query that contains code specific names *PowerManager newWakeLock* which the user copied directly from code they saw in the results (we identified 8 queries where the participants copied a portion of code in a result and tried to use it as a keyword).

We now switch from how our participants searched to what they said about the search engines during our unstructured interview after the experiment. When we talked to participants, some explained some of their behaviors that we saw in the patterns. One participant reported that complexity critiques helped (in search 15) when he found code that was too *"dumb"* and did not know how else to continue, enabling him to increase the desired complexity and find his desired search result. This might explain the role that critiques play in search. While it is used relatively infrequently, as other features seem to suffice most often, critiques still serve an important role when the user is "stuck" in their search.

Another participant said she never used CodeExchange's advanced search because searching through language concepts and automated keyword recommendations let her do similar searches. Participants often expressed that they would modify their queries, in either search engine, to *"back up a level"*, by removing a portion of their query. However, to "back up" in GitHub, participants reported they were forced to start from scratch, needing to remember the keywords they had used before. Participants reported that CodeExchange made it easier

to “back up”, as they could simply activate and deactivate query parts. Others reported that query parts helped them try different combinations of queries.

All the participants had positive reactions to CodeExchange, such as *"CodeExchange is better than GitHub, no doubt."* But participants also had several suggestions for improvement. For example, participants wished to be able to rapidly scroll through results and to click on a method call to navigate to the defining class. One participant wished to search for code not containing keywords or imports and another wanted to see additional “trustability” metrics, such as author reputation. However, none of the recommendations interfere with the features of CodeExchange, but rather might enhance the search experience it provides even further.

4.6.3 Threats to Validity

Several possible threats to validity exist in how we conducted our evaluations. First, for the field deployment, using copies and downloads as an indicator of whether searches were genuine is an approximation. It may be lower than the real number, as it is possible that a developer only needed to read the results to support whatever they needed to learn or remember, but did not copy or download the code. It also might be higher than the real number, since some developers might have been experimenting with the features and simply downloaded a random result. We note, however, that selection and downloads as an indicator of genuine searches has been used in previous work [7], and that copies and downloads is more conservative because copies requires more than just selecting text, as it

also requires actually issuing a copy command on the selected text. We also note that, for most of the searches involving copies and downloads, iteration took place. Overall, then, we believe our reported results are more of a lower bound.

Second, the user study stopped short of looking at the code results that participants selected; that is, we did not examine any qualities of the selected code. However, selection criteria for the copied code are inherently tied to the human performing the search and what they think is best for the search task, as in real world settings. We believe our experiment design mitigated the possible bias resulting from an individual's criteria, because we counterbalanced the experiment and each participant was free to apply their own selection criteria across both conditions. Further, the fact that, twice, participants could not find the code they needed with GitHub suggests that they put their best foot forward.

Third, the user experiment used graduate students as some of the participants. While they had industrial experience, it is possible they exhibited behaviors that are not representative of the broader developer population and how it searches. However, because our overall evaluation also includes a field deployment, and we saw similar behaviors in how the professionals searched and the students searched, we believe this risk is significantly reduced.

4.6.4 Early Evaluation Discussion

Our early evaluations of CodeExchange yielded a few important observations into code search.

1. From both the field study and user study we found programmers search iteratively for code using CodeExchange and GitHub. This confirms previous studies that search is an iterative process.
2. We found that explicitly supporting the programmer in iteratively searching for code plays an important part in the programmer's search. In most searches, in both the field study and lab study, programmers used one or more iterative features in CodeExchange to find code. Further, our lab participants where significantly faster in finding code with CodeExchange, could complete more tasks with CodeExchange, and reported having a more positive search experience with CodeExchange in comparison to GitHub.
3. Our data suggest that search is not only iterative, but that an iteration does not necessarily stop at copying code. In the lab experiment, 61% of the searches using CodeExchange and 33% using GitHub continued after copying code and sometimes copied other code later in the search. This data suggests that part of searching iteratively for code might entail attempting to find multiple options, or at least a desire to not be satisfied with just the first result and perform a more exhaustive search.
4. In contrast to the general patterns of iteration and searching beyond copying code, we also found that programmers often searched differently from each other in both

the field study and lab study. This suggests that the search process the developer engages in is both similar to other developers at a high level in terms of being iterative, but also unique to the individual at a more detailed level. Code search engines, thus, might need to support a more personalized iterative search process.

Chapter 5

CodeLikeThis

When a programmer is searching for code and is not quite sure what they are looking for initially, a result returned for a first query might trigger the programmer to try to find other results that are similar but in a rather fuzzy sense. That is, the programmer might want to find similar results but have no specific part or aspect in mind. They simply recognize the entire result as an example of code they would like to see more of. For instance, suppose a programmer, Bob, wants to build a Rubik's cube game, but is uncertain how to get started. He wants to see some examples of how to maintain game state, interfaces, and possible solving algorithms, but also just wants to discover what other people are doing to get ideas for his own game. To do so, he opens his favorite search engine and issues the keywords *rubik game*. The first result he sees is a Sudoku-Rubik's cube hybrid, where the model maintains the state of a Rubik's cube but includes extra state information accounting for where numbers on the cube are and logic in a function called isFinished to determine if the

rows and columns of numbers on each face of the cube satisfy the rules of Sudoku. Bob did not realize such a game existed or had been programmed before and was somewhat surprised by the result, but he did like how the model was written (it appeared concise with descriptive variable names and had a function testing if the model satisfied the end state of the game), so Bob thought that he could perhaps use code similar to this, but he was left with the task of choosing keywords that would find similar code but not the exact code he is looking at. Bob decides to try the keywords *rubik game model isFinished color match*, but received zero results. Bob then proceeds to try several different keyword queries, such as *rubik cube*, *rubik cube color*, and *rubik cube color isFinished*. After a great deal of thinking in choosing these queries and examining the results, Bob does find some examples of implementing a Rubik's cube and storing the color state of each side of the cube, but was unable to find one containing a function checking for the end state of the game. Bob hits the back button several times to the results from the first query *rubik game*, as Bob still is interested in finding other related code for solving the Rubik's cube game. After paging

```
/**
 *        |*************|
 *        |*U1**U2**U3*|
 *        |*************|
 *        |*U4**U5**U6*|
 *        |*************|
 *        |*U7**U8**U9*|
 *        |*************|
 * ************|*************|*************|*************|
 * *L1**L2**L3*|*F1**F2**F3*|*R1**R2**F3*|*B1**B2**B3*|
 * ************|*************|*************|*************|
 * *L4**L5**L6*|*F4**F5**F6*|*R4**R5**R6*|*B4**B5**B6*|
 * ************|*************|*************|*************|
 * *L7**L8**L9*|*F7**F8**F9*|*R7**R8**R9*|*B7**B8**B9*|
 * ************|*************|*************|*************|
 *        |*************|
 *        |*D1**D2**D3*|
 *        |*************|
 *        |*D4**D5**D6*|
 *        |*************|
 *        |*D7**D8**D9*|
 *        |*************|
 */
```

Figure 45. Rubik's Cube Documentation Example

through these original results some, he stumbles onto code that, apparently, solves the game and has detailed documentation mapping variable names to faces and blocks on the Rubik's cube (as shown in Figure 45). Impressed with the detail in the author's comments, Bob decides to take a closer look at the code. However, upon inspection, the code is hard to read for Bob, because all the values of the cube are encoded as bits in an array and turning the cube is implemented as bit shifts at various indexes in this array – making it hard for Bob to map the code onto what it does to a Rubik's cube. Bob wants something like this code, but hopes he can find code without using bit shifting operations to solve the puzzle. Bob is now tasked with choosing keywords to find such code. Deciding to postpone choosing such keywords, remembering the difficulty he ran into last time, Bob continues to look through the pages of results from the first query and stumbles over a nice UI for his Rubik's cube game. While Bob likes the UI, he wants to search a bit further to see if he might find something even more appealing. Bob is again left with the challenge of encoding the entire result as keywords to find more code like it.

The example of Bob is not unusual. It represents a common case, in which a programmer sometimes sees an entire result as holistically similar to code that could be helpful, without explicitly knowing which parts specifically cause them to do so. In the cognitive science literature, this phenomenon is termed "whole-similarity" matching [113] or "recognition" [57], and refers to when people quickly perceive something as similar to something else without necessarily understanding why. For example, humans very frequently recognize a whole human face without first explicitly understanding what parts of the face make it recognizable [57]. For a programmer, then, recognizing some code as broadly interesting or

helpful might trigger the programmer to reflect and issue a modified query that attempts to encode the result as keywords to find similar code. In some ways, this is an even harder task than our hypothetical programmer, Suzie, faced in Chapter 4, because at least Suzie identified a specific aspect of the result to encode, but Bob is left with the task of either summarizing a whole result as keywords or choosing and encoding multiple aspects of the code as keywords.

Like CodeExchange, CodeLikeThis, our second experimental code search engine, is specifically designed to aid the developer when, initially, the programmer is uncertain of exactly what they are searching for and is engaged in a more exploratory search involving the submission of multiple queries through which to explore what examples may be available. In such a scenario, the insight behind CodeLikeThis is that the next query tends to be relative to the results and sometimes specific to an entire result rather than any specific aspect of a result (unlike the scenarios supported by CodeExchange). As such, CodeLikeThis supports the developer in forming the next query out of the entire result simply by selecting if they want code that is similar to that result. In this way, CodeLikeThis changes how the next query is constructed, from choosing and entering keywords, to creating a query simply by choosing a result to see code that is similar to it.

While the interaction of creating the next query — clicking a result — is simple, supporting it is not as straightforward. While typically the results after the first keyword query are optimized to be the most topically related to the keywords, CodeLikeThis also needs to address giving the user results to select from to begin iteratively searching. If CodeLikeThis

only returns the most topically related results, then the results themselves may all be very similar to each other and hide examples that are different, but still on topic, that the programmer could recognize as similar to what they are searching for. For example, suppose Bob was looking for a Rubik's cube UI and saw only 2-D examples implemented with the Java AWT library in the top results, then 3-D examples implemented in OpenGL or textual examples, as just two kinds of results that may be useful to Bob, remain hidden from him. CodeLikeThis addresses this issue with its **diversity ranking algorithm** that first gives the user a diverse set of code matching the initial keywords. Each code result, then, can equally serve as a point of comparison that the programmer can use to find other similar or dissimilar code.

Once the programmer discovers a result they might want to use for finding similar code, the programmer can use **like-this queries**. Specially, like-this queries support the programmer to retrieve code in the entire search engine by how similar they are to the result the programmer has chosen. This is an important nuanced point: like-this queries do not act as modifications to previous queries (as was the behavior in CodeExchange), but rather replace the previous query with a like-this query.

Three types of like-this queries are supported by CodeLikeThis. If the programmer wants additional results that are very similar to the result they have chosen, then they can issue a *more-like-this* query to retrieve the code examples, from the search engine's index, closest in similarity. If the programmer wants code that is less like a result, they can issue a less-like-this query to retrieve code that is more different than similar, but still having some similarity,

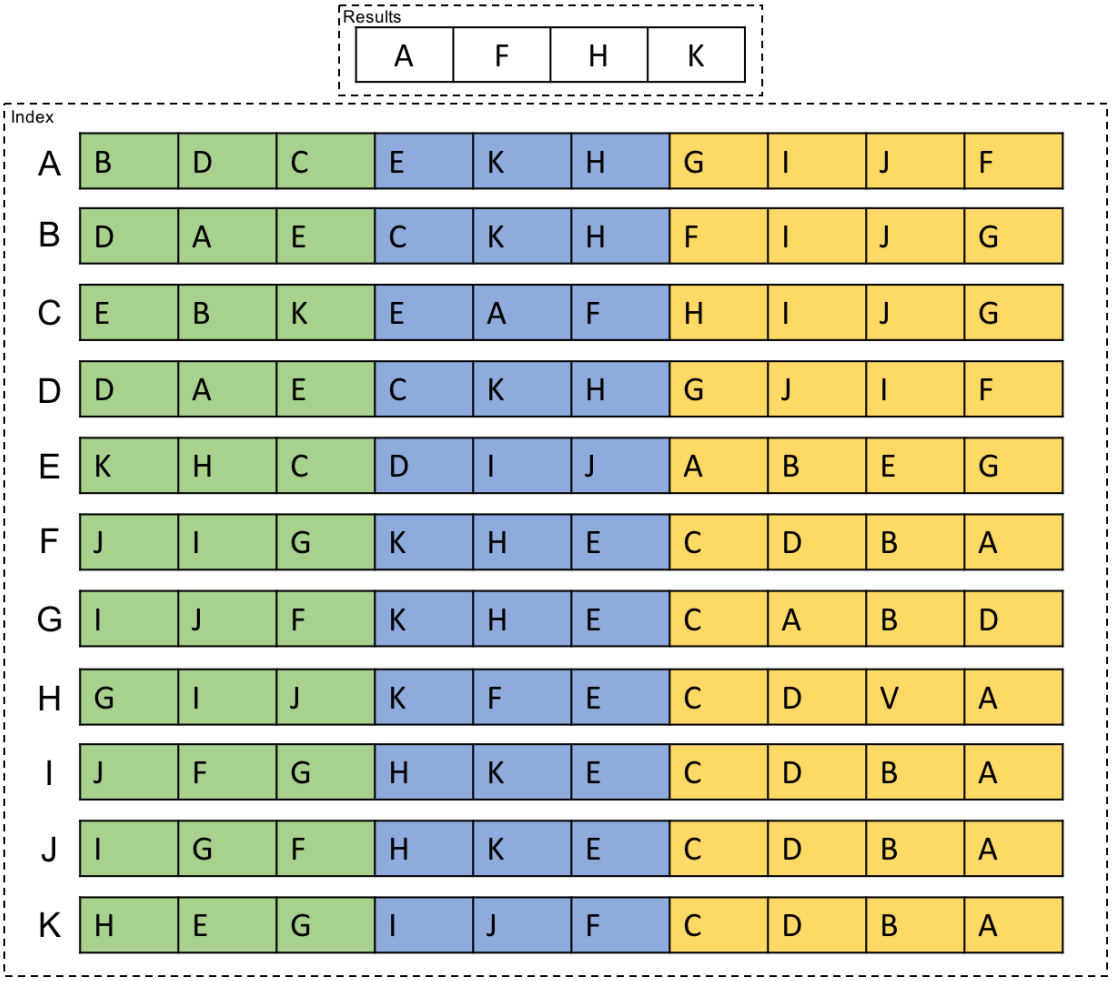


Figure 46. CodeLikeThis Source Code Space.

than the chosen result. Finally, if the programmer wants code that is like a result but feels they also are looking for different code, they can issue a *somewhat-like-this* query, which returns code examples that are not as similar as those returned from the more-like-this query but also not as different as those returned from the less-like-this query.

To illustrate at a high level what issuing like-this queries means, Figure 46 and Figure 47 illustrate how it works with an example. After the programmer's first query using keywords, suppose they receive the diverse set of classes A, F, H and K (as shown at the top of Figure 46). Further, suppose, for the sake of simplicity, that the search engine only has 11 classes

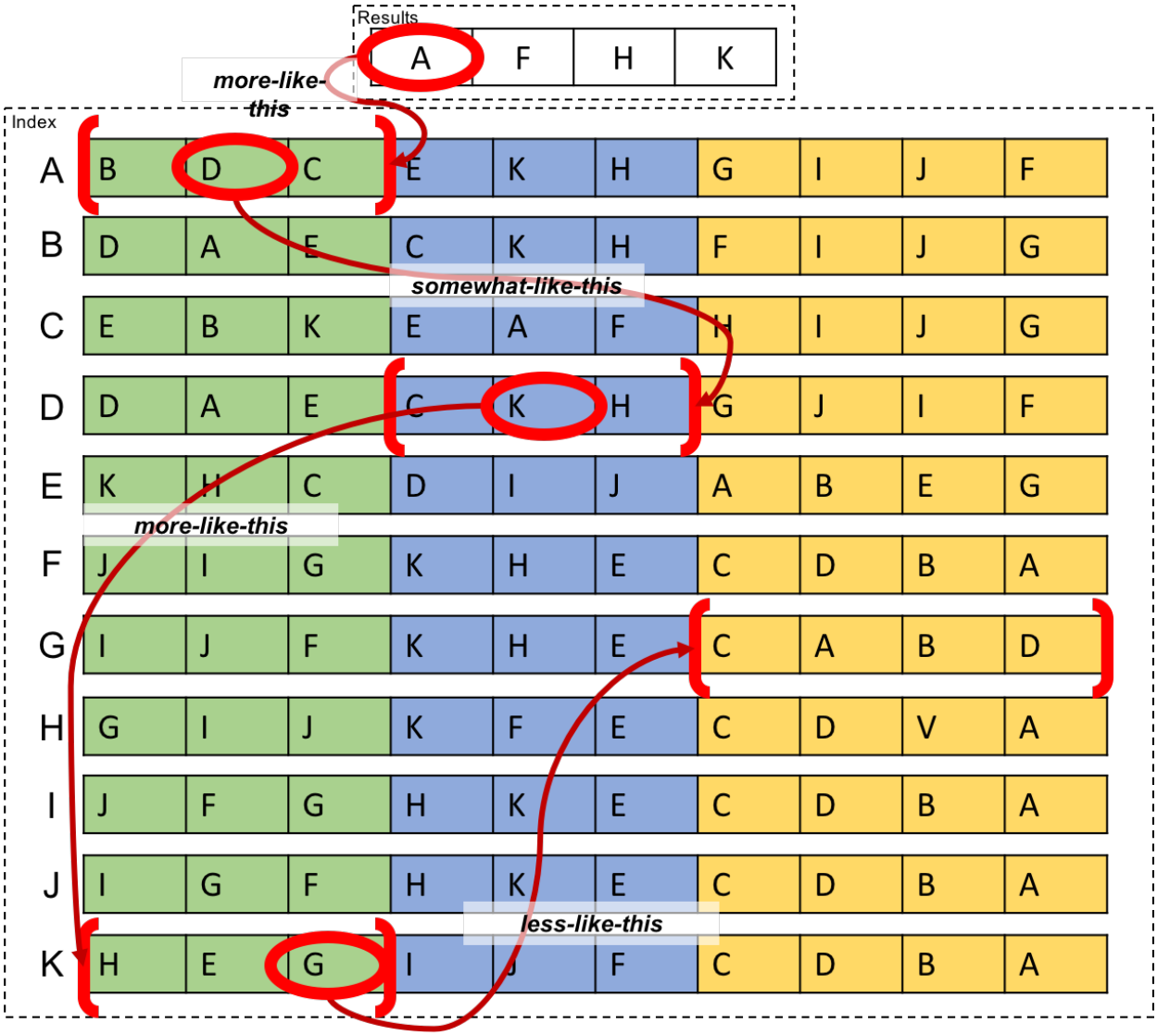


Figure 47. Path Through Space.

in the entire search index (A through K as shown in the left most column in Figure 46). Conceptually, for each of the 11 code snippets, A through K, the other 10 are some distance away in terms of their similarity. Each row in the index in Figure 46 shows the particular ordering of the other code snippets for each of the 11 code snippets in the search engine's index. For example, B is the most similar to A out of all the code in the index. However, relative similarity does not guarantee a total ordering, so it is possible that, in our case, the closest code to B is D, rather than A (though A is the next closest result). The green cells

highlight the top similar classes in the index to a class, the blue cells represent the code that are about an average similarity distance away from the class, and the yellow cells highlight the code that are the least similar to the class. Figure 47 illustrates an example of a programmer searching through this space with like-this queries. To begin searching with like-this queries, the programmer issues a more-like-this query on result A (highlighted with a red circle at the top), which will return the top most similar results (B, D, and C) back to the programmer (indicated by an arrow labeled "more-like-this" going from A to the set {B, D, C}). The programmer then issues a somewhat-like-this query on D to return the code that is an average distance away from D (C, K, and H), then issues a more-like-this query on K to return the most similar code to K (H, E, and G), and ends with a less-like-this query on G to return the least similar code to G (C, A, B, and D). In this way, like-this queries support the user to continually "jump" from one result set to the next by how similar they want the next result set to be relative to the current result.

For a concrete example, suppose a programmer wanted to search for sorting algorithms and decided to start their search by entering the name of a sorting algorithm they knew about, in this case *quick sort*. CodeLikeThis' diversity ranking algorithm would return a diverse set of results that contain, for example, different implementations of quick sort (e.g., different libraries, sizes, complexities), different uses of quick sort (e.g., animations of quick sort or quick sort used in the context of Android), or different API usage examples of calling quick sort functions. If the programmer found a quick sort implementation they thought was broadly appealing, they could click more-like-this to retrieve implementations from the search engine that share similar characteristics (e.g., an implementation with similar size,

imports, naming scheme). However, if the programmer was hoping for an implementation but found the ones presented having overall qualities they did not like (e.g., too difficult to read or the overall style was not quite right), they could issue a somewhat-like-this query on a quick sort implementation to retrieve quick sort implementations that remain varied across a number of dimensions (e.g., size, variable names, complexity, author, and libraries used) compared to the result. In this way, the programmer can attempt to find other quick sort implementations that are dissimilar to the result they selected. Lastly, perhaps the developer is exploring different kinds of sorting algorithms and wants to discover what else exists. To do this, the developer could issue a less-like-this query on a quick sort implementation in an attempt to find sorting algorithm implementations, but not necessarily quick sort (e.g., merge sort or heap sort).

In the following sections, we discuss the architecture of CodeLikeThis, the CodeLikeThis user interface, the design and evaluation of the diversity ranking algorithm, and the design of the ranking algorithm for like-this queries.

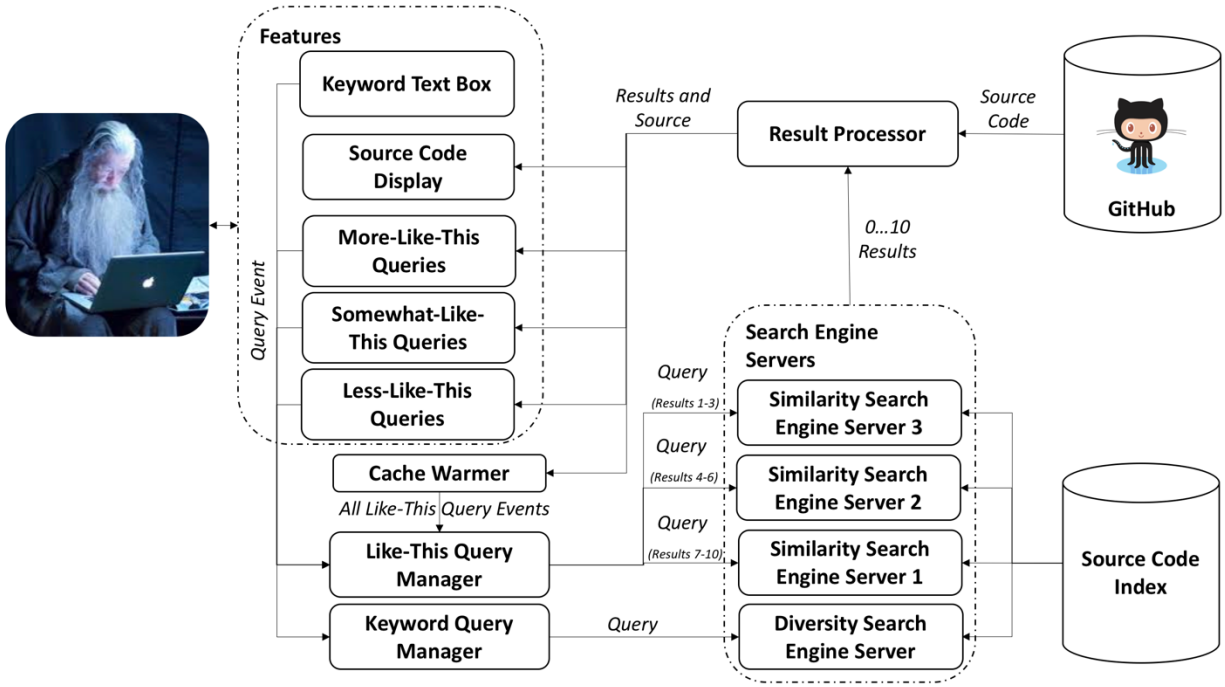


Figure 48. CodeLikeThis Architecture.

5.1 Architecture

The architecture of CodeLikeThis is presented in Figure 48 as a data flow diagram. The features with which the programmer interacts and other important functional components of the system are represented as rounded rectangles. Further, the features are grouped together in the dotted box named Features to show they are the main components with which the programmer interacts in the user interface.

The programmer starts their search with CodeLikeThis by submitting a keyword query with the Keyword Text Box that, in turn, sends a Query Event to the Keyword Query Manager. The Keyword Query Manager uses the event to construct a query (using the Apache Solr syntax) to send to the Diversity Search Engine Server (implemented with Apache Solr). Once the Diversity Search Engine server parses the query it will match and rank code in the Source

Code Index using the diversity ranking algorithm, which will return a set of 10 diverse results to the Result Processor (the rationale behind 10 results is explained in the diversity ranking algorithm section, Section 5.3.2.2). The results themselves consist not only of source code, but also include summaries (as described in Table 10 in the blue columns) of all the information needed by CodeLikeThis' features, as well as a URL to the source code on GitHub. For each of the results, the Result Processor fetches the source code from GitHub and returns the summaries and their source code to the CodeLikeThis interface.

The Source Code Display presents each of the results' source code, and each of the like-this queries (more-like-this, somewhat-like-this, and less-like-this) appear as buttons below each of the results, so that each result has buttons labeled "More Like This", "Somewhat Like This", and "Less Like This". Once one of the like-this buttons is selected, a Query Event is sent to the Like-This Query Manager that uses the ID of the result selected and the kind of like-this query (more, somewhat, or less) chosen to construct a query to submit to one of three Similarity Search Engines, which is determined by which result is selected (the motivation behind having multiple similarity search engines will be discussed soon). If one of results 1 – 3 is selected, then the corresponding like-this query is sent to Similarity Search Engine Server 3, if one of the results 4 – 6 is selected, then like-this queries are sent to Similarity Search Engine Server 2, and if one of the results 7 – 10 is selected, then like-this queries are sent to Similarity Search Engine 1. Once one of the similarity search engines receives the like-this query, the search engine performs a similarity matching algorithm by ordering code in the Source Code Index relative to the code snippet identified by the ID given in the like-this query and fetching results at different distances away depending on the type of like-this

Table 10. Schema and Description of Code Indexed.

Schema of Code Indexed	Description	Features that Reference Field	
Author Name	The name of the person to last edit the class.	Diversity Ranking Algorithm	Like-This Ranking Algorithm
Class Name	The name of the class.		
Cyclomatic Complexity	The cyclomatic complexity of the class as an integer.		
Number of Fields	The number of fields (global variables) in the class.		
Has Wild Card	A boolean value indicating if the class uses the wild card construct.		
Is Abstract	A boolean value indicating if the class is abstract or not.		
Is Generic	A boolean value indicating if the class is generic or not.		
Imports	The set of imports in the class.		
Number of Imports	The number of imports in the class.		
Method Call Names	The set of method call names in the class.		
Method Declaration Names	The set of method declaration names in the class.		
Owner Name	The name of the owner of the project this class is in.		
Package	The package name.		
Parent Class	The name of the parent class.		
Project Name	The name of the project.		
Size	The size of the class in number of characters.		
Variable Words	All, terms, split by camel case, in variables in class		
Terms in Class File	All terms occurring in file. Used for matching topically related classes before diversifying.		

query. The results are then sent to the Result Processor, which fetches the source code from GitHub and sends the results and source code to the features of CodeLikeThis. However, the Result Processor also sends the Results to the Cache Warmer to support making the next results return faster.

The Cache Warmer is responsible for prefetching results from all possible like-this queries the programmer might issue on the current results in order to support getting the next results faster. Specifically, while the programmer is looking at the current results, the Cache

Warmer submits all like-this queries on the current results to the Like-This Query Manager, but configures the events so that no results are actually returned to the programmer. The Like-This Query Manager, in turn, submits the queries to the appropriate Similarity Search Engine, but configured with HTTP parameters understood by the search engines to return zero results. Having multiple Similarity Search Engines enhances the performance of the system, because the queries being generated by the Cache Warmer can be processed in parallel (three at a time), making prefetch faster. There is no reason to stop at having three servers, however we do so because of limited resources. Once the Similarity Search Engines process the queries generated by the Cache Warmer, they keep the queries and the associated results in memory, rather than sending them to the Result Processor. The next like-this query received by a Similarity Search engine is then matched against the queries in memory and, if there is a match, the associated results are sent to the Result Processor, which avoids executing the costlier similarity matching algorithm on the Source Code Index at a time the user is waiting.

All code mined and indexed by CodeLikeThis is done with the same setup used for CodeExchange and described in detail in Chapter 4.1. A List Server maintaining all URLs to Java repositories on GitHub distributes the URLs to a cluster of computers, where each computer in the cluster clones the repository, walks the abstract syntax tree of each Java class in the repository, creates an instance of the schema described in Table 10, and uploads this instance and the URL to the source code to CodeLikeThis to be indexed.

5.2 Interface

The splash screen of CodeLikeThis is similar to most code search engines by presenting a single keyword text box, as shown in Figure 49. In this case, the programmer has entered the keywords *quick sort* and pressed the Enter key to issue the keyword query to CodeLikeThis. Once the results return, the programmer is presented with the main page of CodeLikeThis as shown on the bottom left in Figure 50 presenting the top 10 results for the query *quick sort*. At the bottom of each code result are buttons to issue a query to find other code that is less, somewhat, or more similar to that result. Shown above and to the right of the main page are the top two results after clicking on each of the like-this buttons for the highlighted implementation of quick sort on the bottom left. When the programmer selects the more-like-this button, she gets results (A) that are also quick sort implementations but use different styles and methods to implement quick sort (i.e., similar quick sort implementations but not exact clones). When the programmer clicks somewhat-like-this on the quick sort implementation, she gets results (B) that rely more on other classes (e.g., extending parent classes to implement quick sort) or include comments in other human languages. Lastly, when the programmer clicks less-like-this, she gets results (C) that are no longer quick sort implementations, but are examples of other kinds of sorting algorithms (in this case merge sort and heap sort).

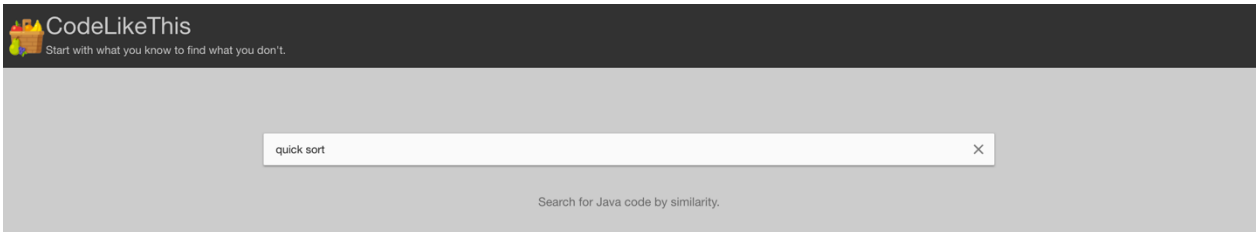


Figure 49. CodeLikeThis Splash Screen.

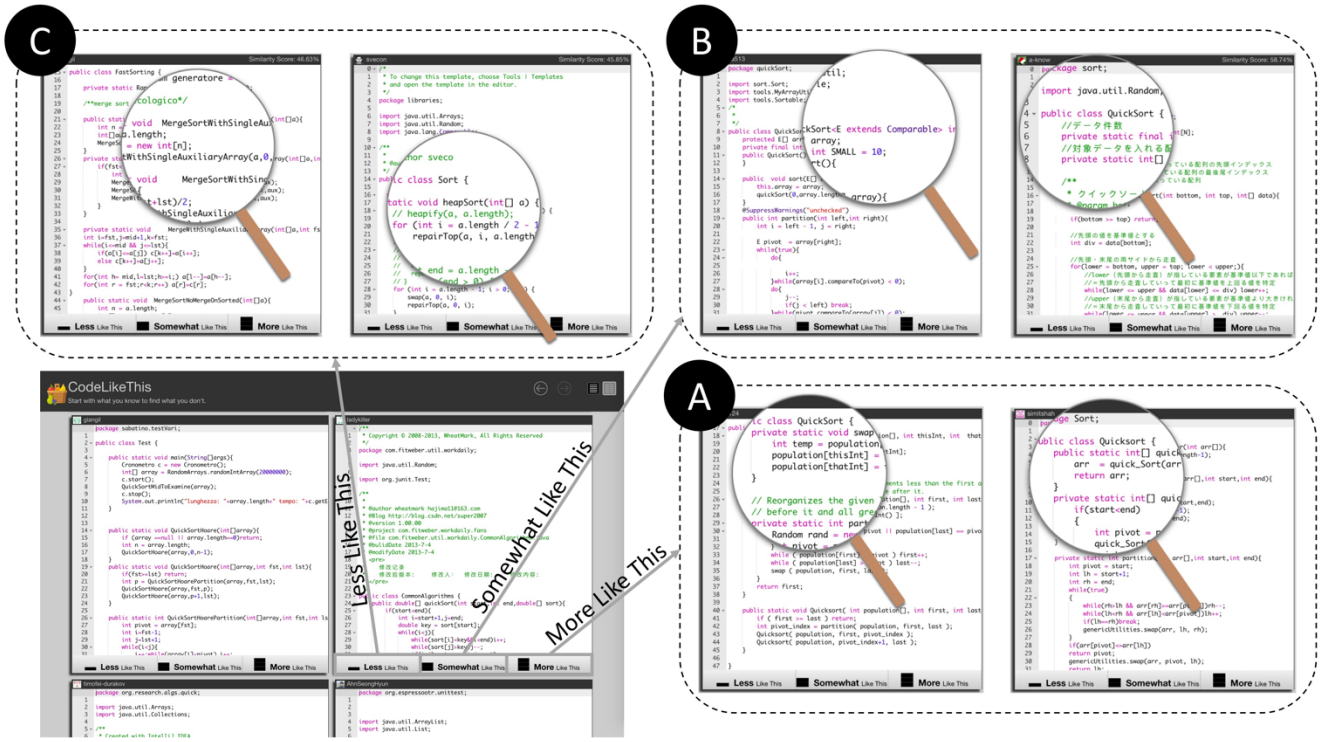


Figure 50. CodeLikeThis Interface.

5.3 The Ranking Algorithms of CodeLikeThis

Both the diversity ranking algorithm, used for the first query, and the similarity ranking algorithm, used for subsequent like-this queries, center around the idea of being able to measure similarity of source code. The diversity algorithm needs to be able to return a set of code results that are dissimilar to each other, while still topically related to the keywords. Depending on the like-this query, the like-this ranking algorithm needs to be able to match code in the index by various distances of similarity to a selected result. As such, both ranking algorithms need a *similarity function* that can organize the code by their similarity relative to each other. In our approach, we use a single similarity function in support of the diversity ranking algorithm and use an approximation of the single similarity function for the like-this ranking algorithm.

The way the similarity function measures similarity needs to match the user's perceptions of similarity for them to recognize and appreciate the differences in the results returned from the first query and the subsequent like-this queries. As such, our similarity function focuses on measuring similarity with well understood aspects/qualities of Java classes that can also be mined in a tractable amount of time and that can be used to produce a similarity measure in a reasonable amount of time for search engines (i.e., a few seconds at most).

The complete list of aspects/qualities of code used by the similarity function is defined by the schema in Table 10 in the like-this ranking algorithm row. Many social and technical aspects/qualities can be obtained (some more difficult than others) and we could have tried to obtain them all (e.g., likes or number of copies). Our decision of those chosen reflects more of a middle ground. We attempted to achieve a reasonable variety of aspects/qualities of code to be able to evaluate the feasibility of our approach without a large upfront cost, such that our single similarity function could use them to produce similarity measures that the diversity ranking algorithm could use to produce diverse sets of results and that the like-this ranking algorithm could use to order code on a spectrum of similarity. We chose to focus more on the aspects/qualities of Java classes, since we scoped CodeLikeThis to Java only.

5.3.1 Similarity Function

The similarity function we designed, called Sim_2^{ST}, measures the similarity between two Java classes by taking each of their schema instances (defined in Table 10, like-this ranking algorithm row) and summing up how much their corresponding values have in common. In

this way, the more they have in common across fields the higher their similarity score, and the less they have in common across fields the lower their similarity score. Further, the commonality between corresponding field values are normalized so that the commonality measurements between corresponding fields do not dominate other fields.

Measurements of commonality for fields that are composed of one or more names is done by taking the intersection of the values. For example, If Class A has imports $\{java.lang.String, java.awt.Graphics\}$ and Class B has imports $\{java.lang.String, java.util.Arrays, java.awt.Graphics\}$, then the commonality score between Class A and B for imports would be $\hat{2}$ (the hat means the 2 is normalized to between 0 and 1). Further, if the author of Class A is Alex and the author of Class B is also Alex then the commonality score between Class A and Class B for author name would be 1.

Measurements of commonality for fields that are composed of numeric measures (e.g., size and complexity) are done by taking the inverse difference of those numbers. For example, if Class A has a size of 1000 characters and Class B has a size of 1350 characters, then the commonality score between Class A and Class B for size is $\widehat{\frac{1}{|1350-1000|}} = \widehat{\frac{1}{350}}$. If both classes are the same size, a .5 is used in the denominator to avoid the undefined condition of zero in the denominator (final score is normalized to 1). For another example, if Class A has complexity 10 and Class B has complexity 1, then the commonality score between Class A and Class for complexity is $\widehat{\frac{1}{|10-1|}} = \hat{\frac{1}{9}}$.

The similarity function is formally defined in Equation 1. It takes in two Java classes (D_i, D_j) and sums up the commonality for each of corresponding field values. Once the final score (lower bound of 0 and upper bound of 17) is calculated from the similarity function, this overall score is normalized resulting in a final score between 0 and 1.

$$
\begin{aligned}
&Sim_2^{ST}\left(D_i, D_j\right) = \sum \\
&\left|authorName\left(D_i\right) \cap authorName\left(D_j\right)\right|, \\
&\left|className\left(D_i\right) \cap className\left(D_j\right)\right|, \\
&\left(\frac{1}{\max\left(abs\left(complexity\left(D_i\right) - complexity\left(D_j\right)\right), .5\right)}\right), \\
&\left(\frac{1}{\max\left(abs\left(\left|fields\left(D_i\right)\right| - \left|fields\left(D_j\right)\right|\right), .5\right)}\right), \\
&\left|hasWildCard\left(D_i\right) \cap hasWildCard\left(D_j\right)\right|, \\
&\left|isAbstract\left(D_i\right) \cap isAbstract\left(D_j\right)\right|, \\
&\left|isGeneric\left(D_i\right) \cap isGeneric\left(D_j\right)\right|, \\
&\left|imports\left(D_i\right) \cap imports\left(D_j\right)\right|, \\
&\left(\frac{1}{\max\left(abs\left(\left|imports\left(D_i\right)\right| - \left|imports\left(D_j\right)\right|\right), .5\right)}\right), \\
&\left|methodCallNames\left(D_i\right) \cap methodCallNames\left(D_j\right)\right|, \\
&\left|methodDecNames\left(D_i\right) \cap methodDecNames\left(D_j\right)\right|, \\
&\left|ownerName\left(D_i\right) \cap ownerName\left(D_j\right)\right|, \\
&\left|package\left(D_i\right) \cap package\left(D_j\right)\right|, \\
&\left|parentClass\left(D_i\right) \cap parentClass\left(D_j\right)\right|, \\
&\left|projectName\left(D_i\right) \cap projectName\left(D_j\right)\right|, \\
&\left(\frac{1}{\max\left(abs\left(size\left(D_i\right) - size\left(D_j\right)\right), .5\right)}\right), \\
&\left|\mathrm{var}\,iableWords\left(D_i\right) \cap \mathrm{var}\,iableWords\left(D_j\right)\right|
\end{aligned}
$$

Equation 1. Similarity Function.

5.3.2 Diversity Ranking Algorithm Designs

We designed two different kinds of diversity ranking algorithms, Social-Technical (ST) and Social-Technical-Hybrid (MWL-ST Hybrid), that use our similarity function in Equation 1. We evaluate these two ranking algorithms and others (discussed later) against each other to find the one that performs better. Both our diversity ranking algorithms leverage a technique introduced in the information retrieval literature that was created to control the similarity between natural language documents matching a keyword query [17]. The goal of the technique was to, given a keyword query, return a *diverse* set of documents, with each document matching a valid interpretation of the keyword query. In this way, the chance of a result matching the interpretation of the keywords by the user is increased and the diversity of the results themselves helps the user obtain an understanding of the kinds of documents that are in the search engine [19], thereby helping them in targeting their next query. The most well-known algorithm for controlling diversity in information retrieval is called Maximal Marginal Relevance (MMR) [17]. This algorithm specifics a procedure to re-rank results so that they are both on topic and different from each other. The flow chart specifying the high-level algorithm of MMR is presented in Figure 51.

As indicated by the green rounded rectangle in Figure 51, MMR starts with the set of results R returned from the topicality ranking algorithm for query Q, an empty set S (to populate with diverse results), and a size k specifying how large to make S. Further, $R = \{D_1, \cdots, D_i, \cdots D_n\}$, where D_i denotes the document with rank i, n is the number of documents found, and $1 \leq i \leq n$. First, MMR initializes S by putting the first result in R into S [42]. The next document put into S is chosen to be the document D_i in $R \backslash S$ (in R but not S) that has the lowest similarity score with any document in S and highest topicality score with Q. This is achieved by looping over each D_i in $R \backslash S$ and finding the document with the highest $\text{MMR}_{\text{score}}$, which is defined as:

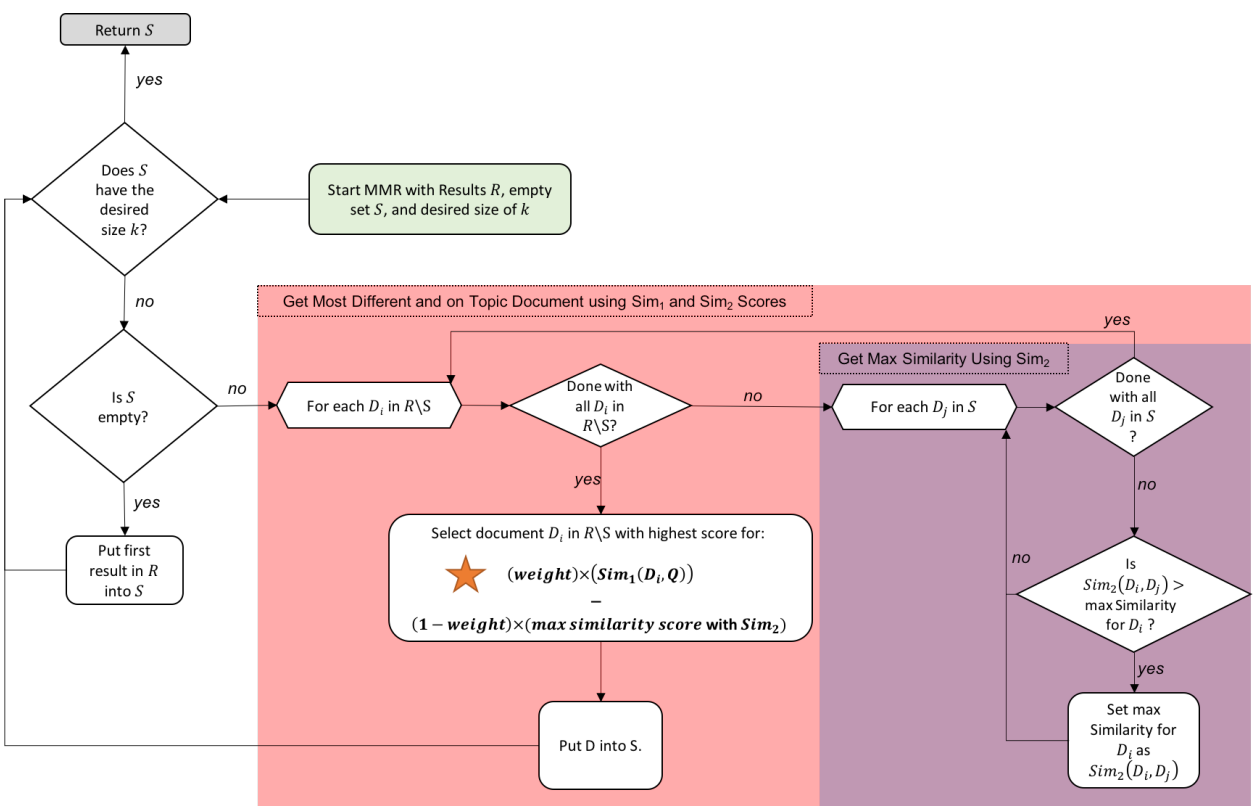


Figure 51. MMR Flow Chart.

$$MMR_{score} = (weight) \times \left(Sim_1(D_i, Q)\right) - (1 - weight)(\text{max similarity score of } D_i \text{using } Sim_2)$$

This MMR_{score} (highlighted with a star in Figure 51) adds the topicality ranking score defined by Sim_1 (this function could be TF-IDF, for example) and subtracts from the topicality ranking score the max similarity score D_i has with any document $D_j \in S$, where similarity is calculated with Sim_2 (this function can be substituted with any similarity function, but in our algorithms we use our similarity function Sim_2^{ST}). Further, MMR_{score} has a weight variable on the similarity score and topicality score in order to allow the MMR_{score} to be tuned to find the best balance of topicality and diversity. For example, if $weight = 1$, then similarity is not considered and only documents with the highest topicality scores are put into S. If $weight = 0$, then topicality scores are ignored (other than working within the set of results, R, returned from the topicality ranking algorithm) and only similarity scores are used to select documents to put into S.

The diversity ranking algorithms we designed use the MMR method to produce a diverse set of results and configure the similarity function, Sim_2, to be our similarity function Sim_2^{ST}, and Sim_1 to be TF-IDF in one algorithm and MoreWithLess (our new ranking algorithm optimizing on topicality and conciseness, see Section 5.3.3.4) in the other algorithm. However, we were concerned with the size of R, the result set produced from Sim_1, because the larger R is, the more time MMR takes to execute and we are constrained to return results in seconds (often the expected time for search engines). We were also concerned with the size for S, because not only does this influence the running time of MMR, but it also impacts

the number of results the programmer needs to consider. The next sections discuss the sizes we chose for R and S.

5.3.2.1 Size of R

MMR attempting to find the most diverse set of results in R becomes increasingly costly with the size of R, where n can grow up to 10M in size in our case. We wanted to see if we could reduce the number of results MMR needs to consider in R by finding a stopping rank in the list of results, where looking past that rank adds little to the construction of the diverse set S. In particular, we wanted to find a rank after which the results are similar to what has already been considered. For example, if the diversity of the results in R does not change much after the top 500 then anything after the top 500 would be similar to what has already been seen and would do little to make S more diverse. Finding such a rank P would allow us to assume all the top results from $1 \ldots P$ could be used by MMR to produce a diverse set of results that would be about as diverse as the top results from $1 \ldots P + i$ $(i \geq 1)$.

Our approach to find such a rank was to look at how diverse the top results are up to various ranks and assess if there existed a rank P where the diversity of the data "turns" (often called a turning point in data analysis [86]) after which the diversity of the results no longer increases in a meaningful manner. Turning points are often found by looking at peeks (maxima) and valleys (minima), where a peek is a point where there exist adjacent points to the left and right with lower values on the y-axis and a valley is a point where there exist adjacent points to the left and right with higher values on the y-axis. In our case, we wanted

to find the deepest valley (i.e., the rank at which the data turns from being most diverse to becoming more similar). To measure diversity of a group of results up to different ranks, we use average group similarity, which is measured by taking the similarity between all pairs of code in the results and then dividing by the total number of pairs [153].

Figure *52* presents a hypothetical graph showing the average group similarity at increasing ranks of the top results. For example, the value 100 on the x-axis identifies the group of results from 1 to 100; its corresponding y-value is the average group similarity of these results. In this example, the similarity of the results increases as we increase the top results from 1 to the 200th rank, but increasing the results past 200 decreases the similarity up until

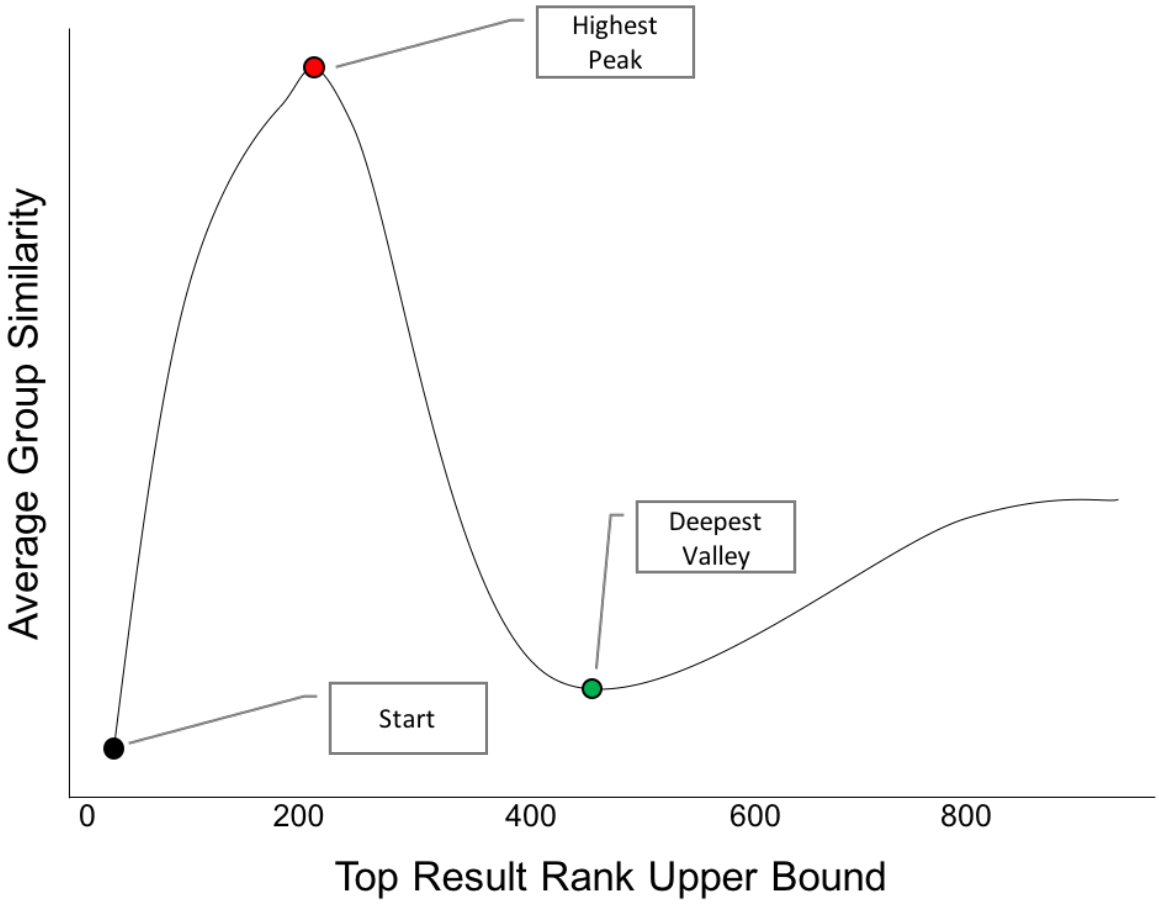


Figure 52. Average Group Similarity at Increasing Ranks of Top Results.

about rank 450. After that rank, the similarity of the results increases to never return to the low similarity found at 450. In this example, 450 is the deepest valley, the point at which adding more results changes from making the results more diverse to more similar. However, we note that the deepest valley does not always contain the most diverse results, sometimes the very first point (e.g., maybe first two results) can be the most diverse purely by having a few results that are very different from each other. However, a few diverse results is probably not representative of all the kinds of results returned from the search engine, so our approach is to collect as many results as we can up until collecting more means that average similarity starts increasing again.

To find a turning point, we created graphs like the example graph in Figure 52 for 21 queries and calculated the average turning point. The set of queries we used is listed in Table 11. To obtain these queries, we selected them from four different code search engine logs. Five were taken from Sourcerer logs [4], five from Mica logs [119], Six from Koders' logs [68], and five from the logs of CodeExchange [75]. The queries were categorized by three software engineering graduate students, not involved with our research, as either algorithm/data structure implementation queries (I) or as API/library example queries (A). We chose these two categories because in a study by Hoffmann *et al*. and in another study by Umarji *et al*. they show that searching for API usage examples or implementations are common among programmers [49], [125]. For 16/21 (76%) of the queries, the graduate students had a 3/3 level of agreement and for the remainder 5/21 (23%) they had an agreement level of 2/3.This yielded 10 API/library usage queries and 11 algorithm/data structure

implementation queries and gave us a balanced set of queries between the two category types.

The results of the turning point analysis are displayed in Figure 53 with all the turning points highlighted with a green square. Each graph plots the group similarity of the results for a query (given in the title of the graph) at 20 evenly spaced and increasing ranks. For example, the first graph (top left) shows the similarity of the results for *quick sort* at 20 progressively larger ranks and stops at the top 1000. The graph shows the similarity of the results quickly peaking at the top 100, then the similarity dropping as the top results grow to 400, but at

Table 11. Twenty-One Representative Queries.

Query	Category	Source
database connection manager	A (3/3)	Sourcerer
ftp client	I (2/3)	Sourcerer
quick sort	I (3/3)	Sourcerer
depth first search	I (3/3)	Sourcerer
tic tac toe	I (2/3)	Sourcerer
api amazon	A (3/3)	Koders
mail sender	A (3/3)	Koders
array multiplication	I (2/3)	Koders
algorithm for parsing string integer	I (3/3)	Koders
binary search tree	I (2/3)	Koders
file writer	A (3/3)	Koders
regular expressions	A (3/3)	Mica
concatenating strings	A (2/3)	Mica
awt events	A (3/3)	Mica
date arithmetic	I (3/3)	Mica
Jspinner	A (3/3)	Mica
prime factors	I (3/3)	CodeExchange
fibonacci	I (3/3)	CodeExchange
combinations n per k	I (3/3)	CodeExchange
input stream to byte array	A (3/3)	CodeExchange
spring rest template	A (3/3)	CodeExchange

rank 400 the results begin to climb again in similarity and gradually steady-out. In this graph, 400 is the “turning point” (highlighted with a green square), because it is the deepest valley where the results are most

Figure 53. Average Group Similarities for 21 Queries at Different Ranks.

diverse and after which they never return to become more diverse. In the next graph to the right (showing result similarity for *tic tac toe*), we find the deepest valley to be at rank 200.

We found the average turning point in the results presented in Figure 52 to be at rank 259 (min=65, max=450, standard deviation=107). This means that, on average, results with rank lower than 259 ($259 + i, i \geq 1$) would be similar to results higher in rank and suggests then that considering results at lowers ranks than 259 would do little in creating a more diverse set S. We therefore set R to be above 259 when using MMR. In our evaluation of the ranking algorithms in Chapter 5.3.4 we set R to 1000.

5.3.2.2 Size of S

MMR takes as input the results R from a query, an empty set S to populate with diverse results from R, and the size k to make S. In the previous section, we found a result size of R that would support creating a diverse set S without having to consider all results in R and thus saves time in running MMR. In this section, we are interested in determining how large the diverse set, S, should be or, in other words, what number to assign to k.

Our objective for setting the size of S was to make it small enough so that the user would consider all the results when choosing a like-this query to apply, but large enough to give the user a sufficient number of options to choose from. To address this, our approach was simply to do a literature search to determine how many results people, in general, look at. We discovered that, in general, they do not look past the top ten results from a query [21], [56], [117]. As such, we decided to set k, the size of S, to be 10.

5.3.2.3 ST Design

As stated, MMR forms the basis of both diversity ranking algorithms that we explored. Both designs are essentially different configurations of MMR, with important distinctions. Social-Technical is our first diversity ranking algorithm and is presented in Figure 54. Essentially, the algorithm takes the top results, R, and submits it to MMR to produce a diverse set of results. The configuration of MMR is listed inside the MMR component in Figure 54. The configuration specifies that MMR should try to find the most diverse set by setting weight to 0 (i.e., only the similarity scores are considered when constructing the set of diverse results and not topicality scores), use our similarity function by setting $Sim_2 = Sim_2^{ST}$, and produce a set of size 10 by setting $k = 10$. The topicality function is TF-IDF by setting $Sim_1 = TF - IDF$, but since weight is set to 0, Sim_1 is ignored and topicality scores are not considered when constructing the diverse set of results.

5.3.2.4 MWL-ST Hybrid Design

While the Social-Technical algorithm is configured to produce the most diverse set of results, we were concerned that just having diverse results for a keyword query may not be sufficient. Specifically, selecting the most diverse results might include results that, while very different, might only be different because they are lower in quality. A quality we were

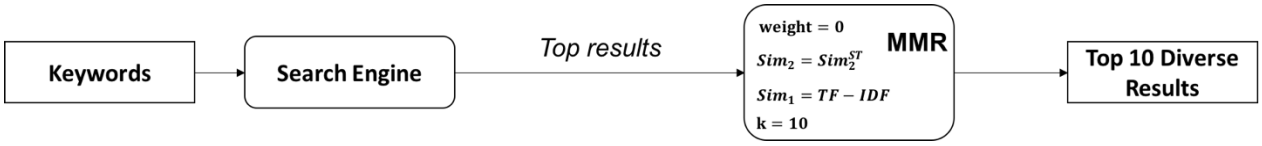


Figure 54. ST Algorithm.

particularly concerned about was conciseness, by which we mean results that are both complete and to the point. If the results were unnecessarily verbose and long or incomplete it might interfere with the programmer from recognizing that result as anything on which they could issue a like-this query. For example, we submitted the keywords *matrix multiplication* to CodeExchange when configured only to match results using TF-IDF and received the top three results presented in Figure 55 (with metadata presented in the top right corner). The first result is an empty class that appears to be the beginning of a matrix multiplication implementation. While brief, this result is not complete and, thus, not a concise example of matrix multiplication. The second result is a class for signal processing using a signal matrix, but not doing any multiplication – it has unneeded code. The third result models scalar matrix multipliers, but does not do any multiplication – it has unneeded code. These top three results have the highest TF-IDF scores, yet none of them demonstrate any sort of matrix multiplication that is brief and complete (i.e. concise). While these three results would not necessarily be chosen by the ST algorithm, they illustrate the existence of low quality results that could be chosen if not controlled for.

```java
package com.dream.algorithm.dp;
/**
 * matrix-chain multiplication problem<br>
 * 矩阵链乘
 * @author liushaohui
 */
public class MCMP {}
```

Rank number: 1
Complexity: 0
Size: 111
Calls: 0
TF-IDF score: 2.8090613

```java
package signals.types;

import matrix.MatrixHelper;
import org.apache.mahout.math.Matrix;
import signals.processing.Decoder;
import signals.processing.Encoder;

public class Sound implements SignalProcessing{
    private Matrix signalMatrix;
    public Sound(Matrix soundMatrix){
        this.signalMatrix = soundMatrix;
    }
    public Matrix getMatrix(){
        return this.signalMatrix;
    }
    @Override
    public void encode() {
        Encoder.encode(this);
    }
...
```

Rank Number: 2
Complexity: 0
Size: 591
Calls: 4
TF-IDF score: 2.5540679

```java
import org.apache.commons.math.linear.RealMatrix;
/**
 * This is a domain object for the parameters
 * needed when performing
 * scalar mulitiplication.
 * @author Jonathan Cohen
 *
 */
public class ScalarMultiplicationParameters {
    private double scalarMultiplier;
    private NamedRealMatrix matrix = null;
    public double getScalarMultiplier() {
        return scalarMultiplier;
    }
    public void setScalarMultiplier(double scalarMultiplier) {
        this.scalarMultiplier = scalarMultiplier;
    }
    public NamedRealMatrix getMatrix() {
        return matrix;
    }
    public void setMatrix(NamedRealMatrix matrix) {
        this.matrix = matrix;
    }
    public void setMatrix(RealMatrix matrix) {
        this.matrix = new NamedRealMatrix( matrix );
    }
}
```

Rank Number: 3
Complexity: 0
Size: 685
Calls: 0
TF-IDF score: 2.32024

Figure 55. Examples of not Concise Matrix Multiplication Results.

5.3.2.4.1 Complexity^{MC} Density

To control for conciseness of results we developed a new metric called Complexity^{MC} Density (M stands for method and C stands for character) that was designed to be a heuristic to measure conciseness by giving results that do more computation with less code a higher score than results doing less/equal computation with more code. Formally, Complexity^{MC} Density is defined in Equation 2. Cyclomatic complexity is used to measure the amount of computation (by counting all possible branch points in the result), where higher scores of complexity help raise the Complexity^{MC} Density score. The amount of code in a result that is performing the computations lowers the Complexity^{MC} Density score by multiplying complexity by the inverse of the amount of code used. The amount of code used is measured by the size (number of characters) of the result and the number of methods used that are external to the result (e.g., method calls in libraries or in other classes). The Complexity^{MC} Density scores for the results in Figure 55 come out to be all zero because each has a complexity of zero.

Figure 56 shows results returned in the previous example for the *matrix multiplication* query that, while lower in rank than the examples in Figure 55, have higher Complexity^{MC} Density scores (0.00986, 0.00479, and 0.00200) compared to the examples in Figure 55. In Figure 56, the first

$$Complexity^{MC}Density = (complexity)\left(\frac{1}{|externalMethodCalls|}\right)\left(\frac{1}{|characters|}\right)$$

Equation 2. Definition of ComplexityMC Density.

```java
public class MatrixChainMultiplication {
    public int matrixChainMultiplication(int [] src) {

        int [][] mc=new int[src.length+1][src.length+1];
        mc[0][0]=0;

        for(int len=0;len<src.length+1;len++) {
            for(int i=0;i+len<src.length;i++) {
                int j=i+len;
                mc[i][j]=Integer.MAX_VALUE;
                if(len==1 || len==0) {
                    mc[i][j]=0;
                    continue;
                }
                for (int k=i+1;k<j;k++) {
                     mc[i][j]=
                      Math.min(mc[i][j],
                                mc[i][k]+mc[k][j]+src[i]*src[k]*src[j]);
                }
            }
        }
        return mc[0][src.length-1];
    }
}
```

Rank Number: 41
Complexity: 5
Size: 507
Calls: 1
TF-IDF score: 1.6406572

```java
public class MatrixMultiplicationNaive {

    private static final String ERRMSG_FORMAT =
        "Columns [%d] of first argument not equal
to rows [%d] of second "+"argument";

    public static int[][] multiply(int[][] f, int[][] s){
        if (f[0].length != s.length) {
            String errMsg = String.format(ERRMSG_FORMAT, f[0].length, s.length);
            throw new IllegalArgumentException(errMsg);
        }
        int rows = f.length;
        int cols = s[0].length;
        int common = f[0].length;
        int[][] result = new int[rows][cols];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                for (int k = 0; k < common; k++) {
                    result[i][j] += f[i][k] * s[k][j];
...
```

Rank Number: 254
Complexity: 4
Size: 834
Calls: 1
TF-IDF score: 0.9363538

```java
public class StracensMatrixMultiplication {
 //add method which adds two matrices
    public int[][] add(int[][] a,int[][] b){
        int[][] sum= new int[a.length][a[0].length];
        for(int i=0;i<a.length;i++){
            for(int j=0;j<a[0].length;j++){
                sum[i][j]=a[i][j]+b[i][j];
            }
        }  return sum;
    }//considering a, b as a square matrix and multiplication is possible only if both has same
dimention
    public int[][] multiply(int[][] a, int[][] b){
...
```

Rank Number: 137
Complexity: 22
Size: 3651
Calls: 3
TF-IDF score: 1.1811008

Figure 56. Examples of More Concise Results for Matrix Multiplication.

result is an implementation of matrix chain multiplication, which not only performs matrix multiplication, but decides the most optimal sequence to perform the multiplication. The second result is an implementation of the straight forward naïve approach to matrix multiplication. The third result is an implementation of matrix multiplication using Strassen's algorithm (a more optimal approach than the naïve version). The results in Figure 56 appear to be examples of matrix multiplication that are shorter and more to the point than those results in Figure 55, which agrees with the higher scores assigned to the results by the Complexity^{MC} Density metric.

Given that the results in Figure 55 had a lower score than those in Figure 56 (because they had a complexity of zero), it is reasonable to think perhaps optimizing only on higher complexity might be the only property of a result that Complexity^{MC} Density should score. However, Figure 57 shows one of the results, also from the *matrix multiplication* query, as an example of why optimizing only on complexity is not sufficient. While this result contains matrix multiplication code, it also has many other mathematical functions and user interface code, which makes it not a very concise example of matrix multiplication. The Complexity^{MC} Density matches this intuition by scoring it at .00000044 (about 5 orders of magnitude smaller than the scores for the results in Figure 56).

```
package Display;

import Output.OutputFormat;
import Tree.BuildTree;
import java.io.IOException;
import java.net.MalformedURLException;
import java.text.ParseException;
import javax.swing.border.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.imageio.ImageIO;
import java.util.Stack;
import java.io.*;

import org.jdom.*;
import org.jdom.input.*;

/**
 * Class to display applet and control all GUI interactions
 * @author Alex Billingsley
 */
public class MainApplet extends javax.swing.JApplet {

    private AddComponent addComponent;
    private OutputFormat output;
    private BuildTree buildTree;
    private StatusBar statusBar;

    private MseSelectListener mouseSelectListener;
    private MseMotionSelectListener motionSelectListener;

    private boolean dragging;
    private InputComponent newComponent;
    private InputComponent[] inputComponents;
...
 ();

    /** Initializes the applet MainApplet
     * Initialises all fields
     */
    public void init() {
        try {
            java.awt.EventQueue.invokeAndWait(new Runnable() {
                public void run() {

                    // Set output format parameter
                    outputFormat = getParameter("outputFormat");

                    // Open with object parameter
...
jButtonMatrix1 = new javax.swing.JButton();
        jButton77 = new javax.swing.JButton();
        jButtonMatrix = new javax.swing.JButton();
        jButtonMatrix2 = new javax.swing.JButton();
        jButtonMatrix3 = new javax.swing.JButton();
...
```

Rank Number: 567
Complexity: 105
Size: 141129
Calls: 1686
TF-IDF score: 0.3379513

Figure 57. Matrix Multiplication Example with High Complexity.

5.2.2.4.2 Using Complexity^{MC} Density in MWL-ST Hybrid

We use the Complexity^{MC} Density to control for conciseness in a new ranking algorithm called MoreWithLess (MWL) that we then use in the MWL-ST Hybrid algorithm. MoreWithLess simply takes the top results produced from TF-IDF and re-ranks them in ascending order by their Complexity^{MC} Density scores. The MWL-ST Hybrid algorithm, as shown in Figure 58, passes the top most 100 concise results to the MMR ranking algorithm to find the 10 most diverse results among the most concise.

5.2.2.5 Kullback-Leibler Diversity Ranking Algorithm

While we now have two alternative diversity ranking algorithms using our similarity function (Sim_2^{ST}) for code, various machine learning approaches for measuring similarity between natural language documents already exist. While these approaches are not designed for code, we used one to create another configuration of MMR, as presented in Figure 59, so that we might gauge how well an approach for measuring similarity between natural language documents performs in MMR. The technique we chose uses the Kullback-Leibler divergence [63] similarity function that measures the similarity between two natural

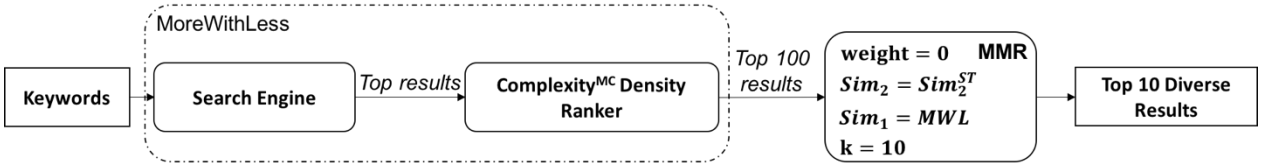


Figure 58. MWL-ST Hybrid Algorithm.

Figure 59. Kullback-Leibler Diversity Algorithm.

language documents by measuring how many of the topics are in common in both documents.

Topics are extracted from the documents using a popular technique called latent Dirichlet allocation (LDA) [11]. Essentially, LDA takes all documents, in our case the top results from the search engine, and creates topics that describe the results. Each topic is a distribution of words, where high probability words are more descriptive than low probability words for that topic. For example, if a topic's highest probability words are chicken, plow, cows, and horses, then that topic can be reasonably interpreted as something about farming and we can conclude some portion of the documents has something to do with farming. LDA lets us figure out which portion, because LDA models each document as a topic distribution. In this way, documents become a distribution of topics and are what LDA, in our case, sends to MMR in the Kullback-Leibler algorithm in Figure 59. Once MMR receives the results, each as a topic distribution, it will apply the KL-Divergence function, which measures how similar two topic distributions are, to create a diverse set of results.

5.2.3 Controls Used in Evaluation of Diversity Algorithms

To evaluate the diversity ranking algorithms, we conducted a pair-wise preference survey to examine which of the ranking algorithms produced a first set of results that programmers preferred.

However, to control for the effect of diversity of results, we also included other ranking algorithms in the evaluation that do not explicitly diversify their top results. In this way, we can also assess the impact diversity has on programmers' preferences.

We compare our diversity ranking algorithms against a variety of representative topicality ranking algorithms that do not focus on diversity. We chose the TF-IDF algorithm as a baseline control, because it is a very basic and a well-known approach for scoring a document's topicality, but many ranking algorithms also build on it. In particular, all the other algorithms we compare are enhancements of TF-IDF in some way (our diversity algorithms and other controls). This allows us to understand the differences in the performance of the algorithms in terms of the differences in the enhancements to TF-IDF. For the other controls, we chose WordNet [66], OpenHub, and MoreWithLess. WordNet was recently shown to improve TF-IDF with query expansion. We chose OpenHub [140], because it is a popular search engine that improves TF-IDF by boosting results that have variable names and other code symbols matching with the query. Lastly, MoreWithLess is a control, of our own making, to measure the impact of optimizing on conciseness in the MWL-ST Hybrid algorithm. In the next sections, we discuss our controls TF-IDF, WordNet, OpenHub, and MoreWithLess.

5.2.3.1 TF-IDF

The TF-IDF ranking algorithm used is the Lucene variant of TF-IDF [165], defined in Equation 3, which is what many Apache Solr implementations use (including CodeExchange). At a high-level, this function takes in a query q and a document d (document is a generic term to refer to any syntactical artifact) and returns a numeric score measuring how descriptive the query is of the topics in the document. TF-IDF does this by measuring how often terms in the query appear in the document and how common those terms are in the rest of the documents indexed. Terms that appear frequently in the document, but also appear frequently in other documents, are not very descriptive and contribute less to the score than terms that appear frequently in the document and more rarely in other documents. Specifically, for each term t in q, the function sums the frequency of t in d (with function tf), multiplies this frequency by the inverse document frequency (with function idf) that measure how common this term is in the other documents indexed, multiplies idf by any boosts for a term (with function $getBoost$) if some terms are more important, and finally multiplies this by a normalization factor that weighs document fields with less terms higher (with function $norm$). We do not make use of the $getBoost$ function, so all terms are weighted equally. The normalization function, however, impacts the final score, because if a document's contents are all in one field or the query is directly on the field containing all the document's contents, then shorter documents will be ranked higher because of the $norm$ function. The assumption behind norm is that the importance of a term in a document field is inversely proportional to how many terms the document field has in total.

$$TF-IDF_{score}(q,d)=\sum_{t\in q}\left(tf(t,d)\bullet idf(t)^2\bullet t.getBoost(\ )\bullet norm(t,d)\right)\bullet cord(q,d)\bullet queryNorm(q)$$

Equation 3. TF-IDF in Apache Solr.

The entire summation in Equation 3 is weighted by the $coord$ and $queryNorm$ functions. The $coord$ function is used to score documents higher that have more of the terms in query q, and $queryNorm$ is used to normalize the document score so that it can be compared across queries and indexes.

5.3.3.2 WordNet

Query expansion is a technique that attempts to match topically related documents with a keyword query, but where these documents might not contain any of the terms in the keyword query. A general approach is to expand the query with synonyms so that other related documents have a chance to be matched. Often these synonyms are taken from a prepared thesaurus, such as WordNet [135], that are publically available and can provide synonyms and antonyms to English words.

Our query expansion control, WordNet, is from Lemos' *et al*. [66] who present a technique to expand keyword queries for code using WordNet. For each term in the query, all synonyms for the term are added to the query and all antonyms are added but each are negated so that code with antonyms are not matched. Specifically, Lemos' *et al*.'s WordNet expansion algorithm is described as follows. Given a query $Q = \{t_1, \dots, t_k\}$, where t_i is a term in the query, the algorithm automatically expands the query such that Q becomes:

$$\left(t_1 \vee s_{11} \vee \cdots \vee s_{1m_1}\right) \wedge \cdots \wedge \left(t_k \vee s_{k1} \vee \cdots \vee s_{km_k}\right)$$
$$\wedge$$
$$\neg\left(a_{11} \vee \cdots \vee a_{1o_1}\right) \wedge \cdots \wedge \neg\left(a_{n1} \vee \cdots \vee a_{no_n}\right)$$

In the query expansion of Q, s_{ij} , where $1 \leq j \leq m_i$, is a synonym from WordNet for term t_i, and m_i is the number of synonyms for term t_i, Further, a_{ig}, where $1 \leq g \leq o_i$, is an antonym from WordNet for term t_i, and o_i is the number of antonyms for term t_i. To implement this control we used the JWI Java library [157] to interface with WordNet.

5.3.3.3 OpenHub

Our other control, OpenHub[2] [140], is a popular code search engine for a variety of different languages, but supports search over Java code (important since we apply all these ranking algorithms on an index of Java code). Since we do not have access to the internals of Open Hub, we cannot explain exactly how it works, but, in an email exchange with the former architect of OpenHub, we were told that OpenHub extends the basic Lucene TF-IDF. Specifically, when ranking, OpenHub boosts results containing identifier names or other code symbols matching the query, gives a smaller boost to results with parts of its identifier names or code symbols matching the query, and then gives an even smaller boost to results containing lower case parts of terms in the query.

[2] OpenHub's code search has, unfortunately, been discontinued at the time of this writing, but was available at the time it was used in our experiments.

5.3.3.4 MoreWithLess

This control is of our own design to discover if optimizing on conciseness without diversifying is preferred by our participants. Specifically, this control is the MWL-ST Hybrid diversity algorithm without the use of MMR to diversify the results as shown in Figure 60.

5.3.4 Evaluation of Ranking Algorithms

Given the diversity ranking algorithms (three in total) and control algorithms (four in total), our goal was to find which diversity algorithm performed best for *first* keyword queries in CodeLikeThis. Further, our goal was to discover how diversity ranking algorithms would perform compared to the traditional ranking algorithms among our controls. Since our approach to search is centered around supporting the programmer, we take a user centric point of view in assessing the ranking algorithms for the first query. To do so, we conducted a pair-wise preference survey by collecting preferences from programmers for the top ten results produced by the different ranking algorithms. The survey itself was taken through an online survey system and was approximately 2.5 hours long and done individually and remotely. The survey started by presenting the instructions in Figure 61 and, once read by the participant, presented different questions. Each question displayed a different query and

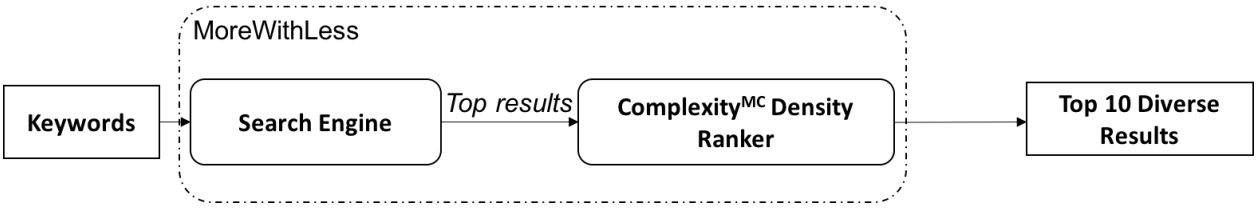


Figure 60. MoreWithLess.

the top ten results from two different ranking algorithms (labeled as “A” and “B” for the

You will be asked to complete 21 tasks. For each task, you will be given a keyword query and the top ten results from Java code search engine A and Java code search engine B. The results are presented in two lists and each list is labeled at the top by the search engine's name producing that list. The lists are not ordered by ranking and instead should be treated as a set of results. Carefully examine the result sets from search engine A and B. Decide which search engine results you would prefer getting. Your decision should consider each of the results in each set, but judge the set as a whole. Indicate your decision by clicking the label naming your preferred search engine's results. After making your selection you will be asked to briefly explain why. Once done, you can continue to the next task.

Figure 61. Instructions Given to Participants.

participant) for that query. Participants reviewed each of the top ten for the query and selected either "A" or "B" with the survey interface.

We designed the study so that each pair of ranking algorithms (21 pairs in total) were judged by each participant once and evaluated for each query once, balancing the design. This was achieved by recruiting 21 participants to evaluate each pair of ranking algorithms on 21 different queries using the Latin square assignment shown in Table 12. Each of the queries are identified in the header of the table (Q1…Q21) and each participant identifier is on the left most column (P1…P21). Not shown in Table 12 is the *order* in which the participants received their queries. To remove ordering bias, the order in which each person was presented queries was randomized; further, the order of the results in a top ten were randomized, and their assignment as results "A" or "B" in the survey was done at random.

Table 12. Latin Square Assignment Design
(m = MoreWithLess, h = MWL-ST Hybrid, k = KL, s = ST, t=TF-IDF, o = OpenHub, w = WordNet)

	Assignment Design																				
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21
P1	(w,t)	(w,o)	(k,s)	(k,m)	(h,k)	(t,k)	(o,k)	(w,k)	(s,m)	(h,s)	(s,t)	(o,s)	(w,s)	(m,h)	(m,t)	(m,o)	(w,m)	(h,t)	(h,o)	(w,h)	(o,t)
P2	(w,o)	(k,s)	(k,m)	(h,k)	(t,k)	(o,k)	(w,k)	(s,m)	(h,s)	(s,t)	(o,s)	(w,s)	(m,h)	(m,t)	(m,o)	(w,m)	(h,t)	(h,o)	(w,h)	(o,t)	(w,t)
P3	(k,s)	(k,m)	(h,k)	(t,k)	(o,k)	(w,k)	(s,m)	(h,s)	(s,t)	(o,s)	(w,s)	(m,h)	(m,t)	(m,o)	(w,m)	(h,t)	(h,o)	(w,h)	(o,t)	(w,t)	(w,o)
P4	(k,m)	(h,k)	(t,k)	(o,k)	(w,k)	(s,m)	(h,s)	(s,t)	(o,s)	(w,s)	(m,h)	(m,t)	(m,o)	(w,m)	(h,t)	(h,o)	(w,h)	(o,t)	(w,t)	(w,o)	(k,s)
P5	(h,k)	(t,k)	(o,k)	(w,k)	(s,m)	(h,s)	(s,t)	(o,s)	(w,s)	(m,h)	(m,t)	(m,o)	(w,m)	(h,t)	(h,o)	(w,h)	(o,t)	(w,t)	(w,o)	(k,s)	(k,m)
P6	(t,k)	(o,k)	(w,k)	(s,m)	(h,s)	(s,t)	(o,s)	(w,s)	(m,h)	(m,t)	(m,o)	(w,m)	(h,t)	(h,o)	(w,h)	(o,t)	(w,t)	(w,o)	(k,s)	(k,m)	(h,k)
P7	(o,k)	(w,k)	(s,m)	(h,s)	(s,t)	(o,s)	(w,s)	(m,h)	(m,t)	(m,o)	(w,m)	(h,t)	(h,o)	(w,h)	(o,t)	(w,t)	(w,o)	(k,s)	(k,m)	(h,k)	(t,k)
P8	(w,k)	(s,m)	(h,s)	(s,t)	(o,s)	(w,s)	(m,h)	(m,t)	(m,o)	(w,m)	(h,t)	(h,o)	(w,h)	(o,t)	(w,t)	(w,o)	(k,s)	(k,m)	(h,k)	(t,k)	(o,k)
P9	(s,m)	(h,s)	(s,t)	(o,s)	(w,s)	(m,h)	(m,t)	(m,o)	(w,m)	(h,t)	(h,o)	(w,h)	(o,t)	(w,t)	(w,o)	(k,s)	(k,m)	(h,k)	(t,k)	(o,k)	(w,k)
P10	(h,s)	(s,t)	(o,s)	(w,s)	(m,h)	(m,t)	(m,o)	(w,m)	(h,t)	(h,o)	(w,h)	(o,t)	(w,t)	(w,o)	(k,s)	(k,m)	(h,k)	(t,k)	(o,k)	(w,k)	(s,m)
P11	(s,t)	(o,s)	(w,s)	(m,h)	(m,t)	(m,o)	(w,m)	(h,t)	(h,o)	(w,h)	(o,t)	(w,t)	(w,o)	(k,s)	(k,m)	(h,k)	(t,k)	(o,k)	(w,k)	(s,m)	(h,s)
P12	(o,s)	(w,s)	(m,h)	(m,t)	(m,o)	(w,m)	(h,t)	(h,o)	(w,h)	(o,t)	(w,t)	(w,o)	(k,s)	(k,m)	(h,k)	(t,k)	(o,k)	(w,k)	(s,m)	(h,s)	(s,t)
P13	(w,s)	(m,h)	(m,t)	(m,o)	(w,m)	(h,t)	(h,o)	(w,h)	(o,t)	(w,t)	(w,o)	(k,s)	(k,m)	(h,k)	(t,k)	(o,k)	(w,k)	(s,m)	(h,s)	(s,t)	(o,s)
P14	(m,h)	(m,t)	(m,o)	(w,m)	(h,t)	(h,o)	(w,h)	(o,t)	(w,t)	(w,o)	(k,s)	(k,m)	(h,k)	(t,k)	(o,k)	(w,k)	(s,m)	(h,s)	(s,t)	(o,s)	(w,s)
P15	(m,t)	(m,o)	(w,m)	(h,t)	(h,o)	(w,h)	(o,t)	(w,t)	(w,o)	(k,s)	(k,m)	(h,k)	(t,k)	(o,k)	(w,k)	(s,m)	(h,s)	(s,t)	(o,s)	(w,s)	(m,h)
P16	(m,o)	(w,m)	(h,t)	(h,o)	(w,h)	(o,t)	(w,t)	(w,o)	(k,s)	(k,m)	(h,k)	(t,k)	(o,k)	(w,k)	(s,m)	(h,s)	(s,t)	(o,s)	(w,s)	(m,h)	(m,t)
P17	(w,m)	(h,t)	(h,o)	(w,h)	(o,t)	(w,t)	(w,o)	(k,s)	(k,m)	(h,k)	(t,k)	(o,k)	(w,k)	(s,m)	(h,s)	(s,t)	(o,s)	(w,s)	(m,h)	(m,t)	(m,o)
P18	(h,t)	(h,o)	(w,h)	(o,t)	(w,t)	(w,o)	(k,s)	(k,m)	(h,k)	(t,k)	(o,k)	(w,k)	(s,m)	(h,s)	(s,t)	(o,s)	(w,s)	(m,h)	(m,t)	(m,o)	(w,m)
P19	(h,o)	(w,h)	(o,t)	(w,t)	(w,o)	(k,s)	(k,m)	(h,k)	(t,k)	(o,k)	(w,k)	(s,m)	(h,s)	(s,t)	(o,s)	(w,s)	(m,h)	(m,t)	(m,o)	(w,m)	(h,t)
P20	(w,h)	(o,t)	(w,t)	(w,o)	(k,s)	(k,m)	(h,k)	(t,k)	(o,k)	(w,k)	(s,m)	(h,s)	(s,t)	(o,s)	(w,s)	(m,h)	(m,t)	(m,o)	(w,m)	(h,t)	(h,o)
P21	(o,t)	(w,t)	(w,o)	(k,s)	(k,m)	(h,k)	(t,k)	(o,k)	(w,k)	(s,m)	(h,s)	(s,t)	(o,s)	(w,s)	(m,h)	(m,t)	(m,o)	(w,m)	(h,t)	(h,o)	(w,h)

The queries we used are the 21 representative queries presented in Table 11 that provide us with realistic queries for the ranking algorithms to run on.

A screen shot of the survey system's interface is presented in Figure 62. The question prompt is displayed in the window on the left and, in this question, displays query "Fibonacci" at the top of the screen. The top ten from ranking algorithm "A" are on the left and the top ten from ranking algorithm "B" are on the right. The labels "A" and "B" are used to hide the real names of the ranking algorithms. The users can scroll down to see all the results and can scroll inside each of the editors to see the code entirely. Each editor color codes the syntax of the code to make it easier to read. The editors also serve to provide a uniform interface across the top ten produced from each of the seven ranking algorithms. The participants indicate their preference by selecting the button labeled as "A" or "B" occurring above the top ten they prefer. Once they make their selection, a popup box appears asking them explain why they made their selection. In Figure 62, the popup box is presented on the right and in this

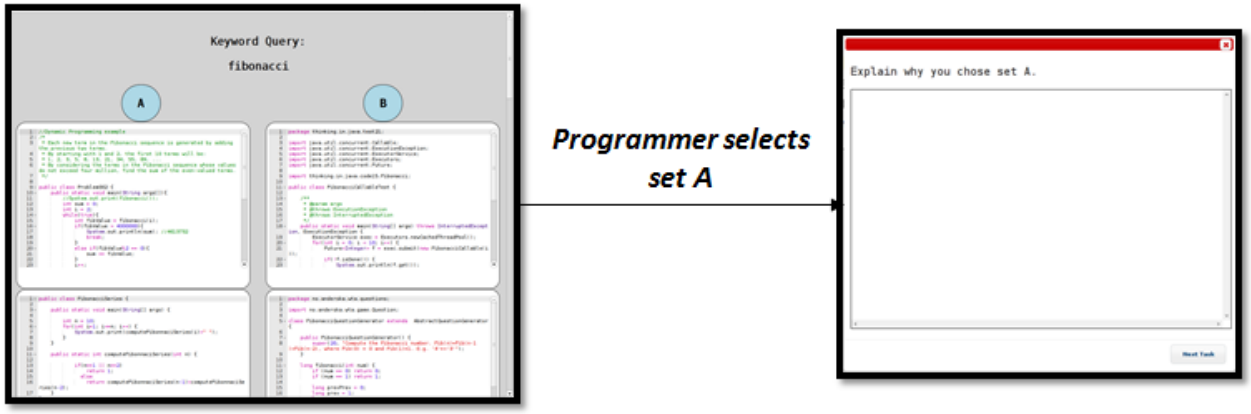


Figure 62. Survey System Screen Shot.

example appeared after the user selected button "A". The popup box asks the user to explain why they chose "A" and provides a text box for them to do write their answer (the popup box would have asked why they chose "B" if it was chosen). Once they are finished typing their explanation, they click the button labeled "Next Task" in the popup box to get the next survey question or, if they have answered all questions, reach the end of the survey.

To recruit 21 participants with professional programming experience and who were skilled in Java, we sent out an advertisement on UCI's Institute for Software Research [150] mailing list. Both professional programmers and students subscribe to this list. We recruited 21

Table 13. Participant Demographics.

	All	Student	Non-Student
Avg. age	32	30.3	34.1
Avg. years worked	7.8	4.8	11.7
Avg. Java skill (1-10)	7.5	7.3	7.8
Male	19	10	9
Female	2	2	0
Number of programmers	21	12	9

programmers who self-reported an average of 7.8 years of work experience and a high skill level in Java of 7.5 (1 being novice and 10 being expert). Table 13 presents the self-reported demographics of our participants.

5.3.4 Results from Assessment

The experiment yielded 441 preferences (21 programmers times 21 questions) from which we created the "preference" table shown in Table 14. Each row identifies a ranking algorithm and each cell shows how many times that ranking algorithm was preferred against the ranking algorithm identified by its column header. For example, MoreWithLess was preferred over Open Hub 14 times. Since each ranking algorithm competes against the other 21 times, this means MoreWithLess was preferred over Open Hub 67% of the time (percentages appear beside the preference totals). The total preferences for each ranking algorithm appear down the total column. MoreWithLess and MWL-ST Hybrid have the highest preference percentage by far (67% and 64% respectively). The others all have below a 50% percentage. The totals suggest that our participants prefer the results from MoreWithLess and the MWL-ST Hybrid more than the others.

Table 14. Preference Table.

	MoreWithLess	MWL-ST Hybrid	Open Hub	WordNet	TF-IDF	KL	ST	Total
MoreWithLess		8(38%)	14(67%)	13(62%)	17(81%)	16(76%)	16(76%)	84(67%)
MWL-ST Hybrid	13(62%)		12(57%)	14(67%)	13(62%)	15(71%)	14(67%)	81(64%)
Open Hub	7(33%)	9(43%)		10(48%)	10(48%)	12(57%)	13(62%)	61(48%)
WordNet	8(38%)	7(33%)	11(52%)		12(57%)	9(43%)	12(57%)	59(47%)
TF-IDF	4(19%)	8(38%)	11(52%)	9(43%)		11(52%)	12(57%)	55(44%)
KL	5(24%)	6(29%)	9(43%)	12(57%)	10(48%)		10(48%)	52(41%)
ST	5(24%)	7(33%)	8(38%)	9(43%)	9(43%)	11(52%)		49(39%)

To understand if the difference in preference for MoreWithLess and the MWL-ST Hybrid is significant, we conducted a chi-squared test on the win-loss table of MoreWithLess and MWL-ST Hybrid. The win-loss table is constructed by assigning a ranking algorithm a point for winning if it was preferred over another algorithm and assigning a point to losing if another algorithm was preferred over it. Table 15 presents the win-loss tables with the chi-squared results below each table. Both chi-squared test tables show the effect size (w), population (N), degrees of freedom (df), significance levels, and power. The population is 126 comparisons (each ranking algorithm is compared 126 times total). The significance level for both are below .05 and let us conclude the preferences for MWL and MWL-ST Hybrid are significant. Further, both effect sizes are above .40. The effect size is a way of measuring the strength of a phenomena and typically a Cohen's w above .3 is considered a medium effect and anything above .5 is considered large effect [9]. As such, MoreWithLess and MWL-ST Hybrid had a medium to large effect on the programmer's preference. While both

Table 15. Win-Loss Table for MWL and MWL-ST Hybrid.

MoreWithLess win-loss table		
	win	loss
observed	84	42
expected	63	63

MWL-ST Hybrid win-loss table		
	win	loss
observed	81	45
expected	63	63

Chi Squared Test
w = 0.4349398
N = 126
df = 1
significance level = 0.0106
power = 0.99

Chi Squared Test
w = 0.4000399
N = 126
df = 1
significance level = 0.03046
power = 0.99

MoreWithLess and MWL-ST Hybrid are preferred over other ranking algorithms, we note that MWL-ST Hybrid is preferred to MoreWithLess more often (13 to 8).

While our participants all had professional programming experience, 12/21 were currently students. We studied whether students' preferences differed from non-students. Further, since the algorithms were evaluated on two types of queries (one for API examples and the other for implementation examples), we examined if the participants' preferences differ for a ranking algorithm by type of query. Table 16 shows the results of this analysis. Each row identifies a ranking algorithm and each column presents the number of preferences for that ranking algorithm by all participants, students, non-students, API/library query type, and implementation query type. For example, Table 16 reports that MWL-ST Hybrid was preferred by everyone in 64% of comparisons, preferred by students in 65% of comparisons, preferred by non-students in 63% of comparisons, preferred in 67% of comparisons when the query type was API/library, and preferred in 62% of comparisons when the query type was implementation.

Table 16. Preference by Student Status and Query Type.

	Total	Student Total	Non-Student Total	API/library Total	Implementation Total
MoreWithLess	84(67%)	49(68%)	35(65%)	42(70%)	42(64%)
MWL-ST Hybrid	81(64%)	47(65%)	34(63%)	40(67%)	41(62%)
Open Hub	61(48%)	34(47%)	27(50%)	24(40%)	37(56%)
WordNet	59(47%)	30(42%)	29(54%)	30(50%)	29(44%)
TF-IDF	55(44%)	34(47%)	21(39%)	34(57%)	21(32%)
KL	52(41%)	30(42%)	22(41%)	33(55%)	19(29%)
ST	49(39%)	28(39%)	21(39%)	28(47%)	21(32%)

From Table 16, we find that the preference for MoreWithLess and MWL-ST Hybrid are fairly well balanced among students and non-students (3% max percentage difference is not significant with Chi Squared test), and they are fairly well balanced among query types (6% max percentage difference, which also is not significant with Chi Squared test). So, the overall higher preference for MoreWithLess and MWL-ST Hybrid is consistent across students, non-students, and the query types. However, we do find a significant preference difference for TF-IDF (p=.008 with Chi Squared test) and KL (p=.0045 with Chi Squared test) among query types. Specifically, TF-IDF and KL are more preferred for API type queries than implementation type queries.

5.3.4.2 Qualitative Analysis

While we found a preference difference among the ranking algorithms, we also needed to know why there was a difference. Specifically, we wanted to know if the MWL-ST Hybrid algorithm was often preferred because of the diversity it provided in the results, because, if so, this would confirm that our design of MWL-ST Hybrid worked as expected and could be adopted as the keyword ranking algorithm in CodeLikeThis. Further, it is also important to uncover additional or unexpected criteria that the programmers used for their preferences as this might indicate the ranking algorithms where optimizing on qualities not predicted beforehand; these qualities could then serve as general guidelines to optimize for in future ranking algorithms.

Our approach to answering why ranking algorithms were preferred was through a process of affinity diagraming (a grounded-theory inspired approach). This process entailed taking all 441 explanations from the programmers (the ones collected in the popup boxes as shown in Figure 62) and having a group of people cluster them into groups by similarity. Each cluster that emerged was carefully examined by the group to find common criterion across the explanations in the group. The criterion found for a cluster is then used to name it. The resulting criteria are interpreted to be the criteria the participants used to make their choices. For example, one cluster had the explanations *"For simple algorithmic queries, I would prefer to see shorter classes going cut to the chase, and I found that in set [MWL_ST Hybrid]"* and *"same amount of good examples, but the ones in set [MoreWithLess] were shorter, and therefore easier to understand and use"*. These explanations, along with other similar explanations in the cluster they occurred in, were used to name the cluster "Smaller Size".

The affinity diagramming consisted of three rounds and started with nine software engineering graduate students taking print outs of each explanation (which did not include the names of any ranking algorithms) and forming clusters by taping the print outs to a wall (as shown in Figure 63). The print outs that the students thought were similar were reorganized to be close together over multiple rounds of iteration, which sometimes led to clusters being combined or breaking up. Eventually, the group of students gave each cluster

Figure 63. Affinity Diagram on Lab Walls.

a descriptive name based on the criteria common to its explanations. For example, one cluster name was "No Clones", because all the explanations in that cluster had something to do with preferring the results that had no code clones. After two and a half hours the group finished the first round of affinity diagraming.

The second round entailed the author and another graduate student colleague correcting mistakes. Specifically, they only moved explanations to other clusters if they were clearly appeared out of place. For example, if an explanation was only about relevancy, but was taped too close to a cluster called "Smaller Size", then they moved over closer to the cluster called "Relevancy". About 12% of the explanations were adjusted[3].

The third, final, round entailed the author and another graduate student colleague merging clusters if the names of both were synonyms of each other. Individually, each graduate student wrote down the list of clusters that they thought should be merged. The clusters that the students both thought should be merged were then merged. For example, both lists stated the cluster "More examples" should be merged with cluster "Diverse" and so these clusters were merged. Once the third round was complete, 14 clusters remained, which gave us 14 criteria used for preferring a ranking algorithm. Using the 14 criteria, we counted how many times each ranking algorithm was preferred using one of the 14 criteria which yielded the results in Table 17. The columns list each of the 14 criteria found and the rows list each ranking algorithm. Each row maps how many times a ranking algorithm was preferred using

[3] The following list reports how many were moved to different sub-clusters before merging: 8 to Relevant, 3 to Quality, 2 to More Keywords, 2 to Many More Examples, 7 to No Category, 3 to Simple, 3 to Concise, 2 to Useful, 1 to Specific, 2 to Cleaner, 5 to No Preference, 4 to Diversity, 2 to No Clones, 3 to Implementation or Algorithm, 1 to Comprehensive, 1 to Size, 9 to More examples.

Table 17. Ranking Algorithm to Cluster.

	Diverse	Concise	Relevant	No Preference	Quality	Useful	No Category	More Keywords	Comprehensive	Less Test Code	Relevant and Clear	Number of Imports	Author of Source	Concise and Diverse
More WithLess	20 (24%)	29 (35%)	11 (13%)	10 (12%)	6 (7%)	1 (1%)	2 (2%)	2 (2%)	0 (0%)	1 (1%)	1 (1%)	1 (1%)	0 (0%)	0 (0%)
MWL-ST Hybrid	28 (35%)	16 (20%)	15 (19%)	9 (11%)	5 (6%)	3 (4%)	1 (1%)	0 (0%)	0 (0%)	2 (2%)	0 (0%)	0 (0%)	2 (2%)	0 (0%)
Open Hub	17 (28%)	13 (21%)	15 (25%)	6 (10%)	1 (2%)	2 (3%)	2 (3%)	1 (2%)	2 (3%)	1 (2%)	0 (0%)	0 (0%)	0 (0%)	1 (2%)
Word Net	14 (24%)	18 (31%)	15 (25%)	1 (2%)	2 (3%)	3 (5%)	3 (5%)	0 (0%)	1 (2%)	0 (0%)	1 (2%)	1 (2%)	0 (0%)	0 (0%)
TF-IDF	13 (24%)	12 (22%)	9 (16%)	7 (13%)	5 (9%)	0 (0%)	2 (4%)	3 (5%)	1 (2%)	1 (2%)	1 (2%)	1 (2%)	0 (0%)	0 (0%)
KL	14 (27%)	13 (25%)	8 (15%)	5 (10%)	5 (10%)	0 (0%)	1 (2%)	2 (4%)	1 (2%)	1 (2%)	1 (2%)	0 (0%)	1 (2%)	0 (0%)
ST	15 (31%)	7 (14%)	11 (22%)	3 (6%)	4 (8%)	4 (8%)	0 (0%)	0 (0%)	2 (4%)	0 (0%)	1 (2%)	2 (4%)	0 (0%)	0 (0%)
Total	121 (27%)	108 (24%)	84 (19%)	41 (9%)	28 (6%)	13 (3%)	11 (2%)	8 (2%)	7 (2%)	6 (1%)	5 (1%)	5 (1%)	3 (1%)	1 (0%)

each of the 14 criteria. For example, MoreWithLess was chosen 20 times for being diverse, 29 times for being concise, and 11 times for being relevant. Each count for a criterion is accompanied by what percentage it makes up of the total count of all criteria used for preferring that ranking algorithm. For example, since MoreWithLess is preferred 29 times because it is concise, then 35% of the explanations for preferring MoreWithLess are because it was concise.

The Total row in Table 17 shows that 70% of the reasons for preferring a top ten from any of the algorithms was because it was either diverse (27%), concise (24%), or relevant (19%). This suggests that these three reasons may play a role in the criteria for a preferred top ten in general.

The cluster "No Preference" indicates the number of times a ranking algorithm was preferred when the programmer had a hard time deciding between the algorithm's top ten set and another. Usually the programmers explained a choice was hard because they thought both top ten sets were good or bad. The category "No Category" counts the number of explanations that could not be categorized during the affinity diagramming process.

In Table 17, we find the most frequent criteria used to prefer an algorithm match the main design decisions behind the algorithms we created, which implies the design did have the intended effect. For example, MWL-ST Hybrid, KL, and ST were all chosen most often because of the diverse set of results produced. Further, MoreWithLess, designed to return concise results, was most often preferred because it produced concise results.

We also find in Table 17 that all algorithms appear to the participants as optimizing on the top three criteria, while optimizing on the lesser criteria more sporadically. However, based on the frequency of a criteria used in preferring an algorithm, we find that some algorithms are better at optimizing on some criteria than others. Specifically, we find the MWL-ST Hybrid algorithm is preferred for diversity more than any other algorithm (8 times more often than the next highest scoring algorithm), which suggests that it is the most preferable algorithm to optimize on diversity. Some examples of participants preferring MWL-ST Hybrid for its diversity are *"I prefer getting the set of [MWL-ST Hybrid] because the set of [Open Hub] only provides codes for DOM/SAX writers."* and *"as a whole, set [MWL-ST Hybrid]*

gave me more different ideas on how to implement a tic tac toe game including a simple ai to play against..."

The MoreWithLess algorithm is preferred for conciseness more than any other algorithm (11 times more often than the next highest scoring algorithm). When comparing MoreWithLess with Open Hub one participant said *"Set [MoreWithLess] contains more concise examples of how to convert an input stream into a byte array."* and one participant chose MoreWithLess because it had "…more short focused bits about prime factors."

From Table 14 we found that MWL-ST Hybrid was preferred more often than MoreWithLess 62% of the time. To determine why, we examined each of the explanations for choosing MWL-ST Hybrid over MoreWithLess. For six of the comparisons, programmers explained MWL-ST Hybrid gave more relevant code. For four comparisons, they said it was a tough choice. For the other three comparisons, MWL-ST Hybrid was chosen because of its diversity. Some explained "*The set [MWL-ST Hybrid] results seem to contain more varieties…"* and *"Set [MWL-ST Hybrid] seemed to have a larger variety of examples relevant to doing different tic tac toe things."* It appears that a part of the reason why MWL-ST Hybrid was preferred to MoreWithLess is because it offers diverse and relevant results. It also appears though it was not always easy to choose between the two.

5.3.4.3 Experiment Discussion

The results from our experiment suggest that the MWL-ST Hybrid algorithm is the best choice among the algorithms we evaluated to use as our diversity ranking algorithm in CodeLikeThis. It was one of the two significantly preferred algorithms and it was preferred more than any other algorithm for the diversity of results it gave the participants. As such, we use MWL-ST Hybrid ranking algorithm as the keyword ranking algorithm in CodeLikeThis. However, we do not view this as the only outcome of the experiment described. While our experiment allowed us to find a keyword ranking algorithm for CodeLikeThis, our experiment also has a few high-level takeaways.

While the diversification algorithms ST and KL diversified results, they were not often preferred. However, the MWL-ST Hybrid diversification algorithm was one of the two most preferred ranking algorithms and was preferred more than any other algorithm for the diversity of the results it produced. One of the key design differences between the hybrid algorithm and the other two (ST and KL) is that MWL-ST Hybrid diversifies concise and on-topic results, but ST and KL diversify only on-topic results. This design difference suggests that diversifying concise results rather than all on-topic results had a role in why MWL-ST Hybrid was more often preferred, suggesting that what is diversified is important. That is, the "quality" of the results may need to be improved first before the top diverse set is found. There are a few possible explanations for why this might be the case. First, returning very

different but low quality results is less likely to include what the programmer might be looking for. Second, even if the set of results is diverse and has on-topic results, if it takes a programmer significant time to read through the set because the results are in-concise, then the programmer may just give up on examining those results and never find anything preferable in them.

While the preferences we collected indicate that programmers prefer concise results, we find that size alone is not a good metric to optimize for it. Recall the MoreWithLess, preferred more than other algorithms for conciseness, used the Complexity^{MC} Density metric to optimize on conciseness. However, Lucene's TF-IDF ranking algorithm, in our case, scores shorter documents higher, yet is preferred less often for concise results (12 times) compared to MoreWithLess (29 times). This suggests that a metric for conciseness needs to not only consider how long a result is, but consider the content of the results and how efficiently it is expressed. We take a step closer to a general measure for conciseness by creating a complexity density heuristic (Complexity^{MC} Density) that considers how much the code does (as measured by its cyclomatic complexity) with how few calls and characters it does it with. However, more research is left to do in understanding the construct of conciseness.

5.3.4.3 Threats

While our participants all had substantial work experience, 12 of them were students. However, the average work experience among the students was almost 5 years – we thought

this acceptable experience. Further, the analysis in Table 16 revealed little difference between the preferences of students and non-students.

While we used 21 queries from four different sources that are representative of implementation and API usage queries, these are by no means all possible kinds of queries. It is important, then, that the results are scoped to the kinds and actual queries used in the survey.

5.4.3 Like-This Ranking Algorithm Design

While our experiment helped us choose MWL-ST Hybrid as the ranking algorithm for for keywords, CodeLikeThis needs a completely different ranking algorithm for handling each of the like-this queries. The like-this ranking algorithm was conceptually illustrated in Figure 47, which presented a view of the index as a similarity matrix, relating each code snippet to another by its similarity score. The like-this queries were used to act on this sorted index, such that a more-like-this query on a result A returns the 10 most similar code snippets to A, a somewhat-like-this query on result A returns the 10 snippets that sit at an average distance away from result A, and a less-like-this query on result A returns the least similar 10 snippets. However, implementing the like-this ranking algorithm has several challenges. In this section, we discuss the implementation of the like-this ranking algorithm and its rationale through a series of challenges and solutions, and conclude with the final solution.

Challenge 1 – 10 Million X 10 Million Matrix

The first major challenge in implementing the like-this ranking algorithm is creating the similarity matrix on which to apply the like-this queries. The size of the index of CodeLikeThis is approximately 10 million Java classes. This means that the total size of the similarity matrix would be a 10 million x 10 million matrix, which further means creating and storing 100 million entries. Creating such a large matrix poses an intractable problem – the time to create such a matrix exceeds any reasonable time. Creating this similarity matrix has a $O(n^2)$ time complexity, where n, in our case, is 10 million classes. In our early prototypes, to even construct one row of this similarity matrix took over an hour of running time (we stopped before completing the computation), which means to create the entire similarity matrix would take around 5 million hours to complete (when only computing the diagonal of the matrix), so in about 570 years the computation would be complete. This time is obviously unacceptable, so we derived a new approach described next.

Solution 1 – One Row of 10 Million Entries

Rather than precompute a 10 million x 10 million similarity matrix, we recognize we only need to compute the similarity measures of code in the index relative to the result on which the user issues the like-this query, giving us a maximum of 10 million similarity calculations. With those similarity measures, we can order the code in the index, as shown in Figure 64,

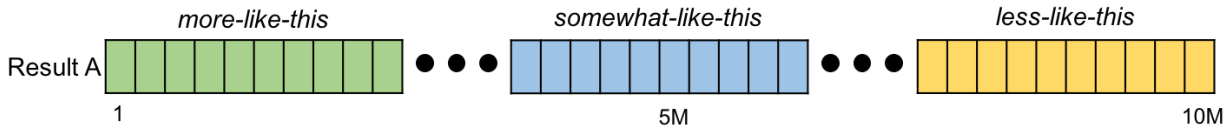


Figure 64. Code Sorted in Index Relative to Result A.

relative to the chosen result and then select 10 results from the top, middle, or the end, depending on which like-this query was issued.

Challenge 1.1 – 10 Million Entries

While computing 10 million similarity scores does reduce the time required to compute similarity scores, it still costs much more than a few seconds (on the order of hours in our experiments) to compute, which is still unacceptable for a search engine.

Solution 1.1 – Top N Entries

We recognize that the results returned from somewhat-like-this and less-like-this queries are determined by the size of the entire index (in this case 10 million). However, rather than using the size of the index to determine what the results should be, we instead select a rank N that determines the results returned, as shown in Figure 65. This requires us to compute only N similarity scores and allows us to control how many calculations are done and, thus, gives us control on the time to execute a like-this query.

Challenge 1.2 – Finding Top N Entries Without Finding 10 Million First

Restricting our similarity calculations only to the top N leads to yet another challenge that resembles the classic "which came first: the chicken or the egg?" dilemma. Specifically, how can we compute the similarity scores for only the top N similar code snippets without first computing all 10 million similarity scores to determine what the top N are?

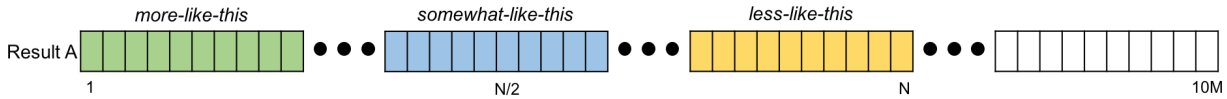


Figure 65. Results Limited by Rank N.

Solution 1.2 – Approximating Sim_2^{ST} with an Advanced Query

To calculate only the top N similar code snippets to a result A, we rely on the insight that the top N code snippets have more aspects in common, as measured by Sim_2^{ST}, with A than results at and below ranks $N + i$, $(i \geq 1)$. Acting on this fact, we use an advanced query (a query on individual fields defined by the schema of the documents indexed) to find, approximately, the top N results with the most aspects in common with result A. Specifically, given a result A, results having, approximately, the most aspects in common with A can be obtained by constructing an advanced query, where the fields in the advanced query map to aspects of A. For example, for simplicity, suppose only three aspects are compared in Sim_2^{ST}: isGeneric, Class Name, and Complexity. Figure 66 presents how the similarity function would score code in the index against result A (1), and Figure 66 also presents how the search engine would score code in the index against the advanced query, "Query A" (2), created out of the result A. The two approaches produce, relatively, similar scores, which suggests the top N results returned for the advanced query will be approximately the top N code snippets found by applying Sim_2^{ST} to the index and sorting. The N results returned from the advanced query may not be in the order produced with Sim_2^{ST}, however, so we can then sort these N using Sim_2^{ST} and use the sorted list to apply the like-this query.

Result Input

Input to Similarity Function	1 **Result A** isGeneric: True ClassName: Sort Complexity: 5
Input to Search Engine	2 $QueryA = OR\begin{pmatrix} isGeneric : true \\ ClassName : Sort \\ Complexity : [4,6] \end{pmatrix}$

Score Table

Index	*Seach Engine Score for Query A*	*Similarity Score Result A*
Snippet B isGeneric: False ClassName: Sort Complexity: 22	4.11	$Sim_2^{ST}(A,B) = \widehat{\left(\hat{0} + \hat{1} + \frac{\hat{1}}{22}\right)} = 1.04$
Snippet C isGeneric: True ClassName: Sort Complexity: 18	11.16	$Sim_2^{ST}(A,C) = \widehat{\left(\hat{1} + \hat{1} + \frac{\hat{1}}{18}\right)} = \widehat{2.05}$
Snippet D isGeneric: True ClassName: Sort Complexity: 4	16.79	$Sim_2^{ST}(A,D) = \widehat{\left(\hat{1} + \hat{1} + \frac{\hat{1}}{.5}\right)} = \hat{3}$

Figure 66. Result to Query and Scores.

The advanced query is created by mapping aspects of result A onto fields in the query, where aspects with boolean and string values directly map onto fields with boolean and string values in the advanced query. Numerical values (e.g., complexity) do not directly map for reasons we will discuss shortly. Each match with a field in the advanced query adds points to the overall score assigned by the search engine, which mirrors how Sim_2^{ST} adds points for each similar aspect between two code snippets. For example, the values of isGeneric and ClassName of result A (Figure 66,1) map directly onto fields isGeneric and ClassName in an advanced query (Figure 66,2), and for each match, the search engine assigns points, yielding the same relative order found with Sim_2^{ST}. However, using an advanced query to find results that have similar numerical aspects (e.g., complexity) to result A is less straight forward. Specifically, state of the art search engines cannot score documents by their distance from values specified in an advanced query. This means that, while Sim_2^{ST} can precisely score

similarity between numeric aspects of two code snippets, an advanced query cannot. However, an advanced query can approximate this somewhat by specifying numerical ranges to match against. For example, in Figure *66*, the complexity of result A (1) is mapped to the complexity field in query A (2) as the numerical range: [4,6]. Given query A, the search engine increases the rank score of results that has complexity of 4,5, or 6, which essentially means that code that is very similar in terms of complexity have their score increased. However, if code in the index has a complexity below 4 or above 7, then its similarity score for complexity is not counted, thus making the use of an advanced query an approximate means of measuring similarity.

The full advanced query we create out of a selected result is presented in Figure 67. The mapping between the selected result's boolean and name aspects map in a straightforward, one-to-one, manner. For numerical aspects (i.e., complexity, number of fields, number of imports, and size), we decided to map the selected result's values to the numeric range: $[result.aspect.value - 1, result.aspect.value + 1]$. Mapping to this range ensures that code that is very similar along numerical aspects has that similarity counted in their overall score from the search engine, which makes it more likely to be in the top N. However, code that is less similar along a numerical aspect will not have their similarity for that aspect counted, which, while that score would be lower in Sim_2^{ST}, it is still larger than zero. As such, using an advanced query is an approximate approach, however our advanced query still scores results higher the more they have in common with the selected result. How wide the numeric ranges in the advanced query need to be to get the best approximation is a matter of tuning the algorithm, as is often needed in many algorithms [11], [29], but would benefit

$$
Query = OR\begin{pmatrix}
authorName : result.authorName \\
className : result.name \\
complexity : [result.complexity - 1, result.complexity + 1] \\
numberOfFields : [result.numberOfFields - 1, result.numberOfFields + 1] \\
hasWildCard : result.hasWildCard \\
isAbsract : result.isAbstract \\
isGeneric : result.isGeneric \\
imports : OR(result.imports) \\
numberOf\,\mathrm{Im}\,ports : [result.numberOf\,\mathrm{Im}\,ports - 1, result.numberOf\,\mathrm{Im}\,ports + 1] \\
methodCallNames : OR(result.methodCallNames) \\
methodDecNames : OR(result.methodDecNames) \\
ownerName : resultOwnerName \\
package : result.package \\
parentClass : result.parentClass \\
projectName : result.projectName \\
size : [result.size - 1, result.size + 1] \\
\mathrm{var}\,iableWords : OR(result.\mathrm{var}\,iableWords)
\end{pmatrix}
$$

Figure 67. Advanced Query to Match Code in Top N Similar.

from future work. The risk of making the range too wide would be counting code that is very dissimilar along a numerical aspect the same as code that is very similar along that same aspect.

Now that we have a method to get the top N similar code snippets we need to decide how large N should be. For the purposes of evaluation, we "tuned" the N to 300, because this let us return results within 5 to 30 seconds (using all 40 cores on an Amazon server rented for our experiment). However, because of the Cache Warmer (in Figure 48), that prefetches possible like-this results while the user is looking at their current results, times to return results often appear to be closer to 1 to 10 seconds to the user. Further, we saw in Section 5.3.2.1 that the top 259, on average, contain a diverse set of results, which means sorting the

top 300 would give us a distribution of different kinds of code so that code found at the tail will be less similar to code at the head and make less-like-this queries work closer to our designed intention.

Final Solution

The Like-This ranking algorithm fetches the top 300 code results similar to the result selected by the user by transforming the selected result into an advanced query (Figure 67). Once the algorithm has the similar results, it sorts them with the Sim_2^{ST} function. While these results may not be exactly the top 300 that would be found by sorting the entire index with Sim_2^{ST} relative to the selected result, they are the top 300 similar results found with an advanced query and sorted exactly as Sim_2^{ST} would sort them and done in an acceptable amount of time (on the order of seconds) for search engines. On the sorted list of results, the Like-This ranking algorithm selects either the top 10, middle 10, or last 10 from the list, depending on the type of like-this query issued by the user.

Figure 68 shows the dataflow diagram of how the advanced query gets created after the user issues a like-this query. The result selected, on which to apply a like-this query, and the like-this query type (more, somewhat, or less) are sent to the Advanced Query Constructor (inside the Like-This Query Manager in Figure 48) that turns the result selected into a query. The Matching Algorithm (using Apache Solr's algorithm) matches N code snippets in the index that best match the query and outputs the results to the Sorter. The Sorter takes the results and sorts them using the Sim_2^{ST} function and uses the type of like-this query to then

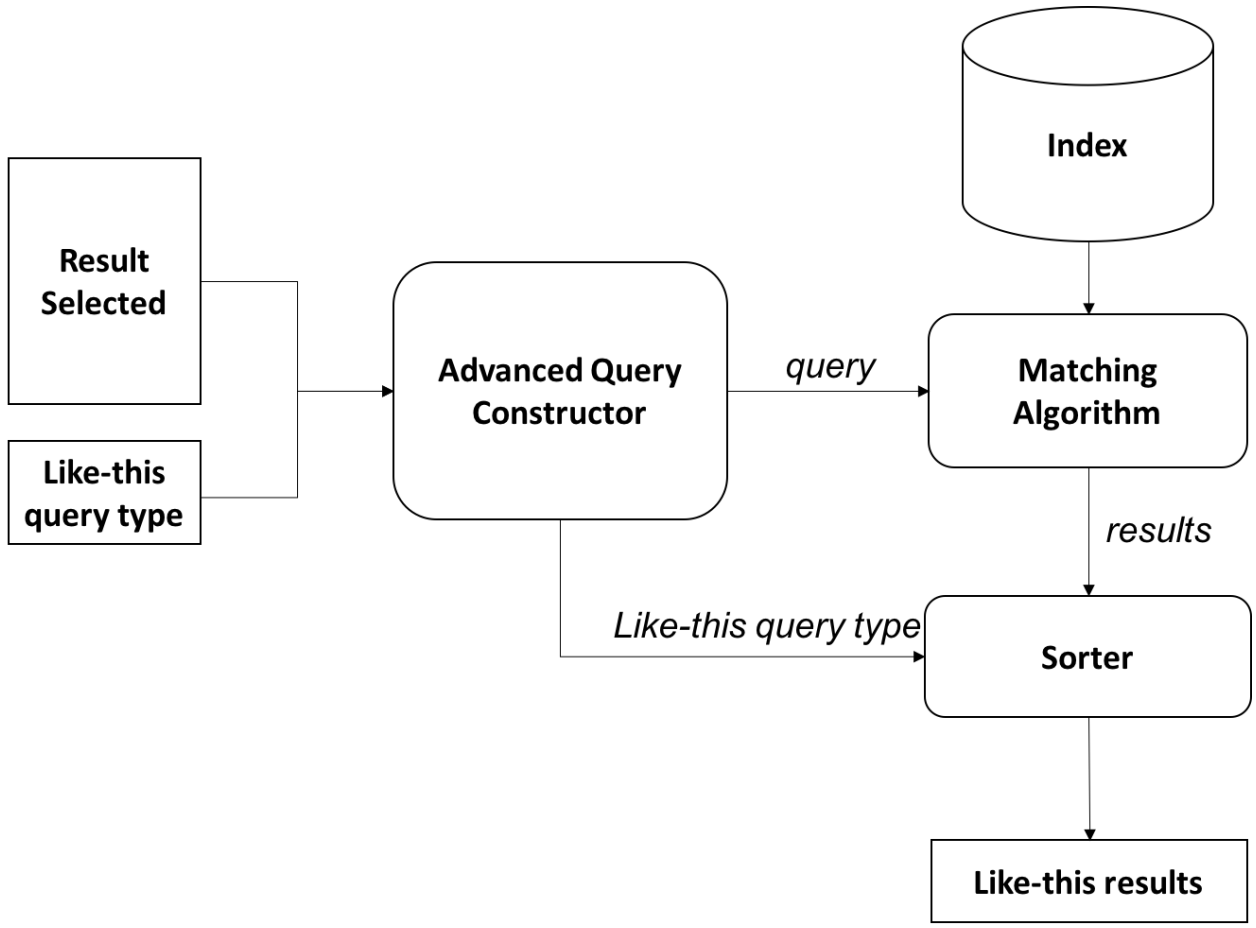


Figure 68. Like-This Ranking Algorithm Architecture.

select either the top 10, middle 10, or last 10 from the list, depending on the like-this query type, and return them as the results.

Chapter 6

Evaluation

In Chapter 4, we introduced CodeExchange and presented a preliminary evaluation between CodeExchange and GitHub, which yielded data suggesting that providing developers tool features to iteratively search with aspects of the results of a previous query may help in searching for code in terms of success and time. In Chapter 5, we introduced CodeLikeThis and an evaluation of its diversity ranking algorithm that supported the use of the MWL-ST Hybrid algorithm as a preferable ranking algorithm to begin iteratively searching with the results. In this chapter, we return to the overarching research question motivating the development of both CodeExchange and CodeLikeThis:

> *What is the impact of explicitly supporting software developers in searching iteratively on the experience, time, and success of the code search process?*

6.1 Experiment Design

We begin answering our research question by evaluating the impact of the approaches of both CodeExchange (using an aspect or quality of a result in creating the next query) and CodeLikeThis (using the entire result in creating the next query) in searching iteratively. To do so, we conducted an empirical study measuring the experience, time, and success of 24 participants in searching for code in eight different search tasks using our two new iterative approaches as well as two non-iterative approaches (a baseline search engine that mimics existing code search tools on the Internet and Google)[74]. Both the baseline and CodeExchange use the more state of the art Specificity ranking algorithm [6] (making this version of CodeExchange differ from the one in Chapter 4 by replacing the TF-IDF ranking algorithm). Replacing the ranking algorithm in CodeExchange was easily achievable because the iterative features provided by CodeExchange are orthogonal to any ranking algorithm. We used Specificity in both the baseline and CodeExchange so that the baseline's ranking algorithm represents a modern ranking algorithm for code and so that we can measure the impact of adding iterative tool features to a search engine equipped with a state of the art ranking algorithm.

6.1.1 Non-Iterative Approaches

Our baseline was a control created to measure the impact of a lack of iteration support in a traditional non-iterative search engine, while maintaining the same code index used in the iterative approaches. The baseline was created by changing CodeExchange's interface into the traditional non-iterative interface presented in Figure 69. The process resulted in a

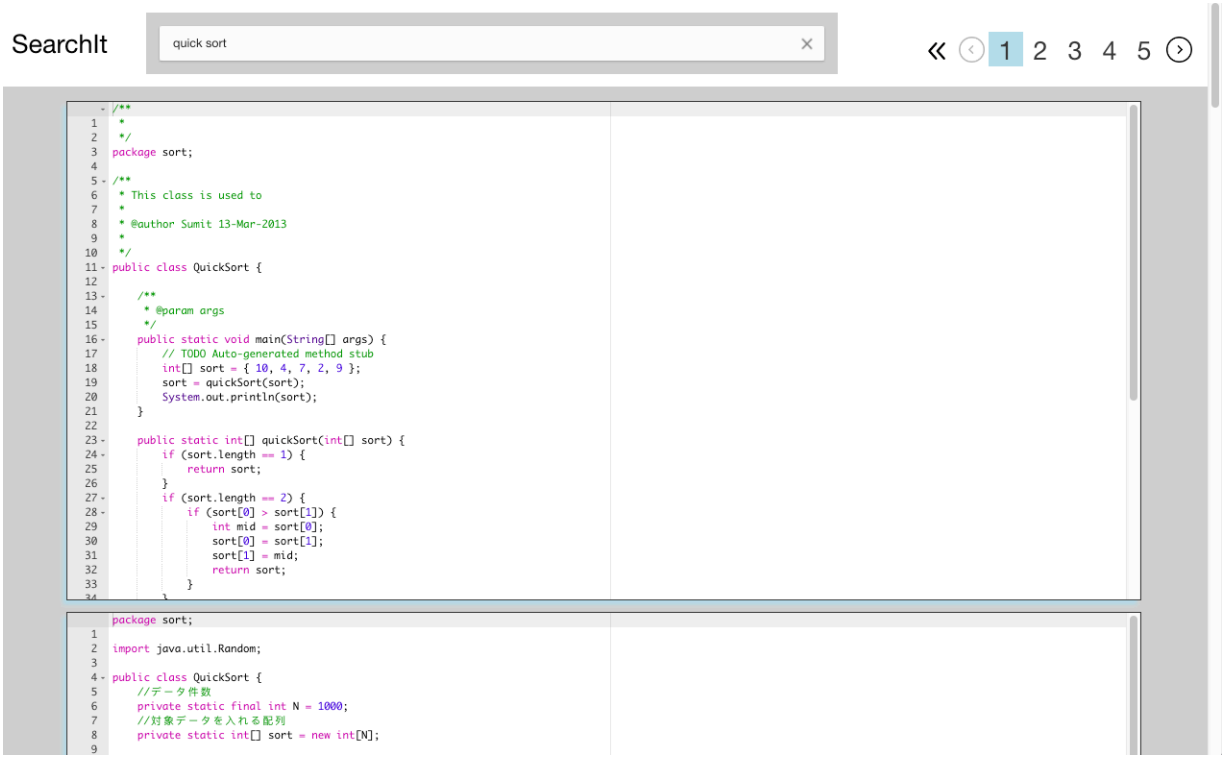


Figure 69. SearchIt, the Baseline Search Engine.

traditional non-iterative search engine that used the same index as the iterative approaches and used a state of the art ranking algorithm, Specificity (same as CodeExchange). The basic features included in the baseline are a keyword text box with autocomplete, a list of code results, and a paging mechanism. We called our baseline search engine "SearchIt" when introducing it to participants in order to hide the fact we were using it as a baseline.

Our other non-iterative approach, Google, is not a code search engine, but is familiar to many people. It indexes a vastly greater amount code on web pages compared to the other three search engines in this experiment (iterative and baseline search engines). While this presents a significant confounding variable and can make interpretation of our results more challenging, we felt it still important to evaluate the most popular and ubiquitous form of search today [79] to gain an understanding of how developers iteratively search with it and

to be able to place our results in this context. Further, evaluating code search engines with Google gives us a unique opportunity to address a more meta-level question: are code search engines useful for finding code, or is Google simply sufficient?

6.1.2 Participants

The participants in this study consisted of 24 developers who reported to have approximately 4 years of professional development experience on average (standard deviation = 2.67), above intermediate Java skill level (median of 5 skill level on an ordinal scale of 1 as beginner and 7 as expert and with a standard deviation of 1.03), an average age of 26.2 (standard deviation = 3.61), and a 20/4 male-female ratio. The complete demographic data is presented in *Table 18*. While we had more participants that are currently students, they had a somewhat higher average for age and experience and a higher median of skill than the non-students. This might be due to the fact that many of the student participants were graduate students with many years of professional developer experience behind them already.

Table 18. Self-Reported Demographics.

	ID	Age	Gender	Professional developer experience (Years)	Java skill level (1=novice,7=expert)	Student Status
	P1	21	male	0.75	6	Yes
	P2	24	male	2	4	
	P3	26	female	2	5	
	P4	23	male	1.5	4	
	P5	30	male	8	5	
	P7	35	male	7	5	
	P8	23	female	1.5	5	
	P9	26	male	5	5	
	P10	25	male	3	5	
	P12	33	female	9	5	
	P13	32	male	3	4	
	P16	24	male	2	5	
	P18	29	male	5	6	
	P21	23	male	1.5	5	
	P22	28	male	7	5	
	P23	25	male	3	6	
	P24	25	male	2	5	
sub totals	*17 particpants*	*26.58823529*	*14 M, 3F*	*3.720588235*	*5*	
	P6	26	female	4	3	No
	P14	23	male	1.8	6	
	P15	23	male	1	4	
	P19	27	male	3	3	
	P20	30	male	10	6	
	P17	24	male	2	2	
	P11	24	male	2	5	
sub totals	*7 particpants*	*25.28571429*	*6M, 1F*	*3.4*	*4*	
Grant Totals	*24 Partipants*	*26.2*	*20M, 4F*	*3.6*	*5*	

6.1.3 Assignments

The laboratory experiments were held in a closed lab setting where each participant sat alone in a room completing eight different and independent search tasks, in sequence. Each participant was assigned two search engines for the entire experiment and completed each

search task using only one. We chose to set the number of search engines to two per person to reduce the learning curve effect that would have been needed had each participant used all four search engines. Using two, however, still allowed them to make comparisons between the search engines in their feedback.

We designed the assignments as follows:

- We used the Latin Square [118] design for our experiment to evenly distribute the tasks among the search engines and participants. The assignments are presented in Table 19. In the table, each row identifies a participant by ID, the two treatments (search engines) they used, and the tasks assigned to each treatment (indicated by shading). For example, participant P1 used CodeExchange and Google and completed tasks 1,3,5, and 7 with CodeExchange (indicated by CodeExchange being shaded and tasks 1,3,5, and 7 being shaded) and completed tasks 2,4,6, and 8 with Google (indicated by Google and tasks 2,4,6 and 8 not being shaded). The tasks are rotated left one with each participant so that each search engine was used in 48 tasks in total and used for each task six times, which yielded a total of 192 data points.
- To address ordering effects, the search engines alternated with each task and all the tasks came in a random order for each participant (while still alternating among search engines). This was accomplished by assigning each search engine to a task (done with Latin Square above) creating pairs (S_a, T_1), (S_b, T_2), (S_a, T_3), $\cdots$, (S_b, T_8), where S_a is one assigned search engine and S_b is the other assigned search engine and each T_j, $1 \leq j \leq 8$, is one of the eight tasks. When a participant finished a

task, then a pair with the other search engine was chosen at random and used as the next task and search engine to use.

- Each participant received a different task for each search engine, so each participant never repeated a task on two different treatments – making the experiment a between subject design to reduce carry over effects. In addition, each participant used different search engines, making it also a within subject design. As such, our experiment design is called a hybrid or mixed design.
- Lastly, to push participants to give each task some thought and effort, we asked them to include explanations for what the code does and why they chose it. This was a tactic to get the participants to genuinely attempt each task.

Table 19. Task to Search Engine Assignment.

Person ID	Treatments		Task Rotation							
P1	CE	G	1	2	3	4	5	6	7	8
P2	CE	CLT	2	3	4	5	6	7	8	1
P3	CE	SI	3	4	5	6	7	8	1	2
P4	CLT	SI	4	5	6	7	8	1	2	3
P5	CLT	G	5	6	7	8	1	2	3	4
P6	SI	G	6	7	8	1	2	3	4	5
P7	G	CE	7	8	1	2	3	4	5	6
P8	CLT	CE	8	1	2	3	4	5	6	7
P9	SI	CE	1	2	3	4	5	6	7	8
P10	SI	CLT	2	3	4	5	6	7	8	1
P11	G	CLT	3	4	5	6	7	8	1	2
P12	G	SI	4	5	6	7	8	1	2	3
P13	CE	G	5	6	7	8	1	2	3	4
P14	CE	CLT	6	7	8	1	2	3	4	5
P15	CE	SI	7	8	1	2	3	4	5	6
P16	CLT	SI	8	1	2	3	4	5	6	7
P17	CLT	G	1	2	3	4	5	6	7	8
P18	SI	G	2	3	4	5	6	7	8	1
P19	G	CE	3	4	5	6	7	8	1	2
P20	CLT	CE	4	5	6	7	8	1	2	3
P21	SI	CE	5	6	7	8	1	2	3	4
P22	SI	CLT	6	7	8	1	2	3	4	5
P23	G	CLT	7	8	1	2	3	4	5	6
P24	G	SI	8	1	2	3	4	5	6	7

6.1.4 Search Tasks

In Section 2.1.4, we summarized the previous research on why programmers search for code. Previous research found that programmers search for a wide range of reasons that can be broad in focus (e.g., getting ideas or learning) or more narrow in focus (e.g., remembering or copying and pasting specific code). In our experiment, we try to capture this range of focus in our search tasks. In particular, the search tasks were designed to cover a space of tasks that are broad to more focused. The more focused tasks were designed to constrain the search task by limiting the number of snippets to find (mirroring finding one snippet for remembering how to perform some programming activity) or by specifying the kind of code to find in more detail (mirroring occasions when the programmer has a clearer idea of what they want). In contrast, the more broadly focused tasks were designed to loosen the constraints of the search tasks by increasing the number of snippets to find (mirroring occasions when the programmer has no one snippet to find, such as when learning, generating ideas, or finding alternatives) or by specifying a higher level or less specific description of the code to find (mirroring occasions when the programmer is not quite sure what they are looking for, such as when learning or generating ideas).

This space of tasks is presented in a 2x2 matrix shown in Table 20. The broader tasks are found in the "Find 4" row and the "No Specific Role for Code" column. The more focused tasks are found in the "Find 1" row and the "Algorithm/Data Structure" column. We designed the space so that there is some range in the broadness and focus of the tasks. The broadest tasks (T3 and T7) appear in the upper left of the matrix, and the most focused tasks appear in the

Table 20. Task Matrix.

			No Specific Role for Code – Broader			Algorithm/Data Structure – More Focused
Find 4 – Broader (20 min limit)	T3	Scenario	You are building a sketching application.	T1	Scenario	You are making your favorite video game.
		Task	Find 4 snippets of Java source code that you think will help.		Task	Find 4 snippets of Java source code that you think will help implement the algorithms and data structures.
	T7	Scenario	You are building a program to survey people's preferences.	T5	Scenario	You are building a program to help teach students basic algebra.
		Task	Find 4 snippets of Java source code that you think will help.		Task	Find 4 snippets of Java source code that you think will help implement the algorithms and data structures.
Find 1 – More Focused (10 min limit)	T2	Scenario	You are building a new email client.	T4	Scenario	You are teaching a class and need to split your class up into groups of four.
		Task	Find Java source code that counts new messages.		Task	Find Java source code that implements an algorithm and data structures to find all possible groups of students.
	T6	Scenario	You are building a large healthcare patient record keeping system.	T8	Scenario	You are building the world's first online phone book.
		Task	Find Java source code to add or edit records in the system.		Task	Find Java source code that implements an algorithm and data structures to retrieve phone numbers by name.

bottom right (T4 and T8). The bottom left tasks (T2 and T6) and the upper right tasks (T1 and T5) are mix of both broader and more focused tasks. This design creates a more representative sample of tasks by covering a range of broad and focused tasks, but also allows us to take different "perspectives" by looking at the experiment results in terms of dimensions of broadness (Find 4 and No Specific Role for Code) and focus (Find 1 and Algorithm/Data Structure).

The topics of the tasks were created to mirror real-world topics for code search. The topics were derived by reverse engineering scenarios from eight of the real queries identified in Chapter 5 (*tic tac toe* – T1, *mail sender* – T2, *AWT events* – T3, *combinations n per k* – T4, *array multiplication* – T5, *database connection manager* – T6, *JSpinner* – T7, and *binary search tree* – T8) found across four different code search engine logs [73]. With the topics, we created the tasks in a style similar to those used in other code search studies [48], [71], where the

tasks do not give the participant a query or the actual code snippet to find, as this would not be representative of how programmers search in the real world. Rather, participants are expected to formulate the queries themselves and figure out which snippets work according to them for the given tasks. Each scenario with task is composed only of one or two sentences expressing a the problem for which the participant needs to find code to help solve. The participants had 20 minutes for the "Find 4" tasks and 10 minutes for "Find 1" tasks. We set the time limits for completing tasks based on our pilot studies where we found our participants could finish tasks in the given time limits.

6.1.5 Survey System

To start the experiment, the participants watched a tutorial video on each of their assigned search engines that explained all features (including the advanced search feature for Google). After watching the videos, each participant "warmed up" by playing around and interacting with each of their assigned search engines, as they pleased, for a few minutes. Once done, each participant started the survey system (shown in Figure 70), which presented the time the participant had for a question (A), a search engine hyper-link (B) indicating which search engine to use (when clicked, this would open the search engine), the search task (C), five tabs (D) each containing an editor to paste the code found for the search task, a text box to explain what the code does (F), and a text box to explain why the participant chose the code (G). We gave the participants five tabs in case they wanted to find more than the required number of snippets. When a participant hit the done button (H) to indicate finishing, or when time ran out, the participant was prompted to rate their experience (I), from 1 to 7, for using the

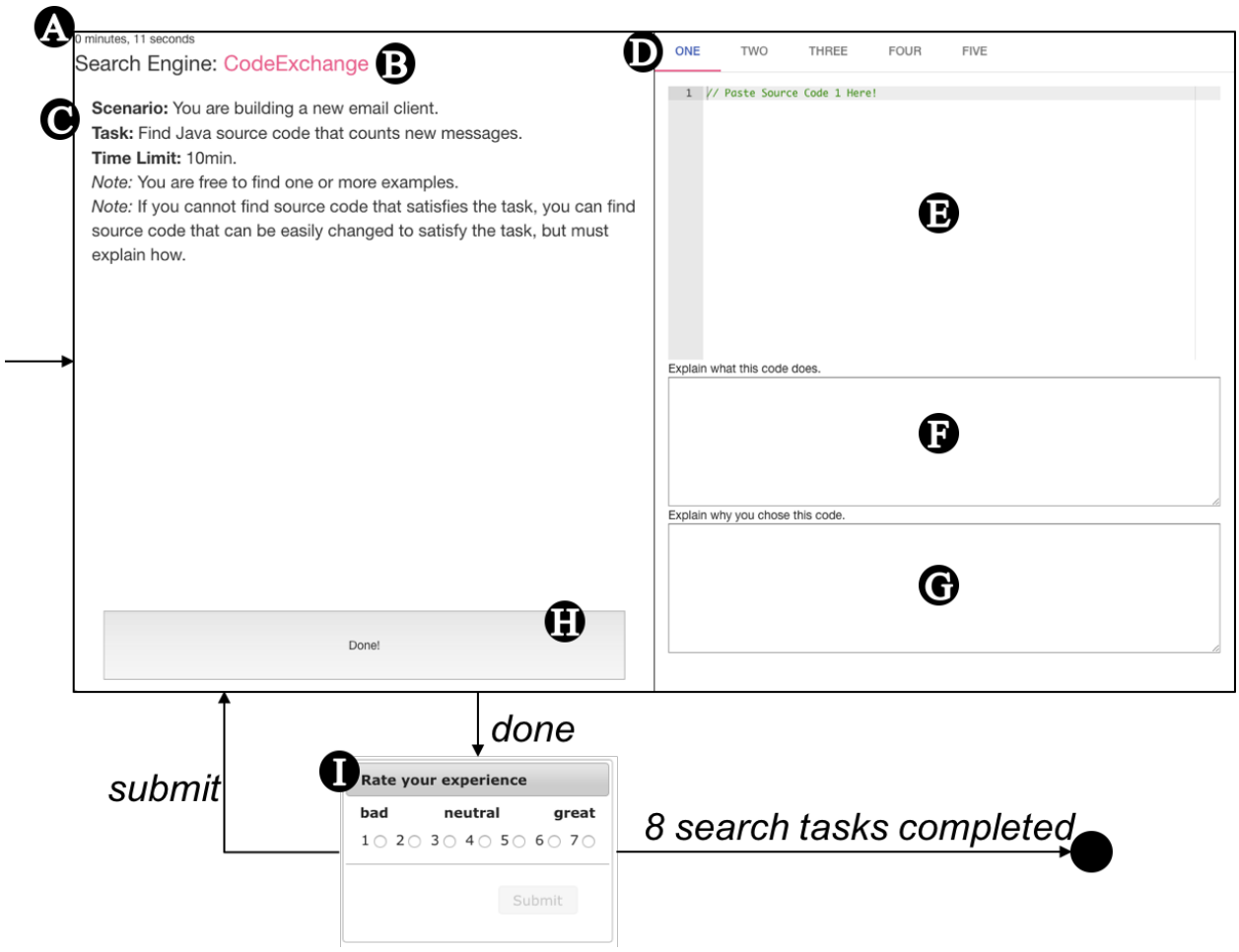


Figure 70. Survey Interface.

assigned search engine for the given task, where 1 was labeled “bad”, 4 was labeled “neutral”, and 7 labeled “great”. Once the participant was done giving their experience rating, they hit submit and received the next task and treatment. Finally, once the participant finished all eight tasks, they filled out a questionnaire about the treatments used and then had an open and unstructured interview with the researcher about their experience.

The rationale behind getting the user to rate their experience after each task is grounded from prior research demonstrating that user satisfaction scores taken immediately after the task are highly correlated with task performance [102]. Further, giving user satisfaction scores after a task allows the participant to make a judgment while their experience is still fresh in their mind, making the score more reliable. A survey about experiences after the entire experiment would have measured overall impressions, which is shown to be a different kind of measure [101]. We limit the user feedback to one seven point Likert scale question on experience (called a “Single Question Likert”), because it limits interference with the participant’s main objective of finding code and one answer has been shown it can a strong indicator of task performance [101], [102].

6.2 Results

From the entire experiment, we collected a total of 192 experience scores, 192 time durations to find one snippet (including times to find first snippet in the “Find 4” tasks), 96 time durations to find four snippets, and 24 questionnaire answers and interviews. Further, we logged the search behavior across all the search engines, giving us data for what features were used and how often. Further, we collected 463 (out of 480 possible) reasons why code

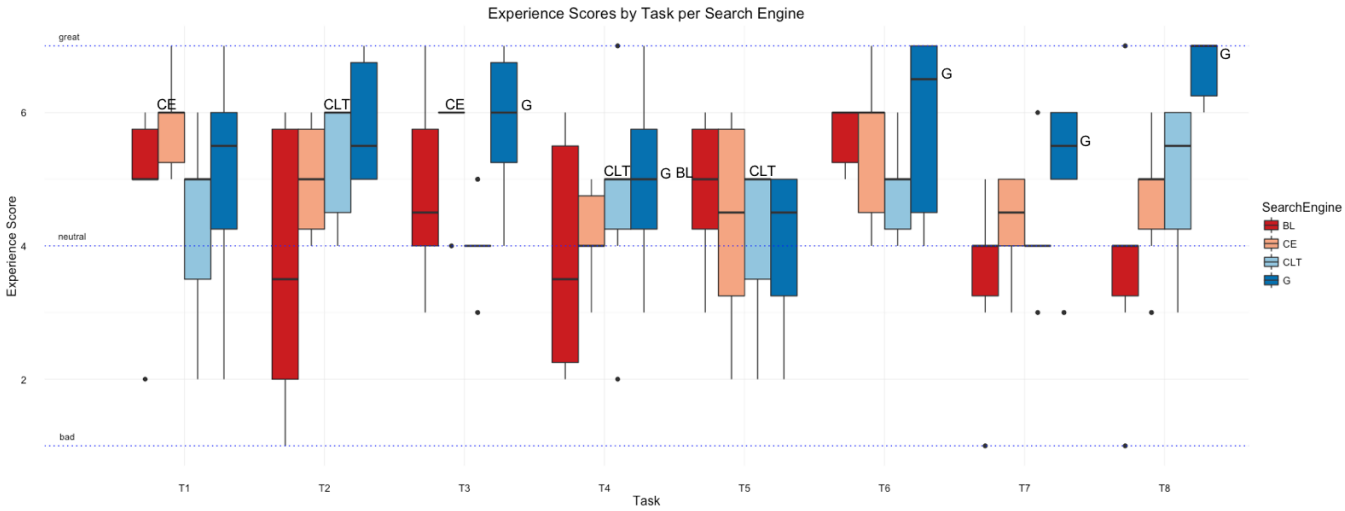


Figure 71. Experience Scores as Box Plots per Task.

was chosen (each of the 24 participants was asked to find 20 snippets in total and give a reason why they chose each snippet). We use this data to examine the impact of supporting iterative search on the experience, time, and success of searching for code. Further, we look at how the search engines were used and the reasons people reported for searching for code.

6.2.1 Experience Scores

Our first analysis compares the experience scores (as rated by participants from 1=bad to 7=great) for using each search engine for each search task. To compare experience scores, we observed when one approach had a higher median score than others. It is standard practice to use the median and not mean when the data is ordinal, such as the experience scores. We created the box plot summary, shown in Figure 71, of the experience scores for each search engine by task. Each box is color coded by search engine, shows the median score with a black horizontal bar through it (sometimes the bar is at the top or bottom), and the height summarizes the spread of the scores. For each task, the search engine with the highest

median scores has its plot annotated with its initials. If there are ties between search engines, then both names appear above their corresponding plots.

From our box plot, we count how many occasions an iterative approach to search had a higher median score than the baseline and Google. With these counts, we then perform $\chi 2$ tests to look for statistical significance, were p-values are obtained by comparing the number of higher, equal, and lower median scores between treatments on a 2x3 contingency table with $\chi 2$. We set alpha to .1 as it is not an uncommon value in smaller-scale experiments and is done in other studies in software engineering [66]. The reason from the statistics literature to set alpha to .1 for smaller-scale experiments is that the standard error (impacting significance) depends on the variance and size of the sample and is often higher in small sample sizes and lower in larger sample sizes [136]. In fact, the literature encourages alpha to be set to lower and lower values (e.g., .01 or .001) the higher the sample size is, because size drives the standard error to lower values [136].

The first analysis, iterative experience versus baseline experience, is presented in Table 21. The first row labeled "Iterative and Baseline" presents the 2X3 contingency tables of the actual and expected wins, ties, and loses of the iterative approaches versus the baseline. For each task, if an iterative approach has a higher median than the baseline this counts as a win, an equal median counts as a tie, and a lower median counts as a loss. If the baseline has a higher median than both iterative approaches then it counts as a win, an equal median counts as a tie, and a lower median counts as a loss. The first row of Table 21 indicates there is always an iterative approach either providing higher experience (on six occasions) or equal

experience (on two occasions) and never an occasion where the baseline provides a better experience than both iterative approaches. These results are found significant (p=.002) with a $\chi 2$ test. This suggests that an *iterative design* provided a better user experience. Untangling the two iterative approaches, the second and third row show the contingency tables and $\chi 2$ test results per respective iterative approach. These rows reveal that many of the higher scores for the iterative approach come from CodeExchange, significantly often (p=0.028), however CodeLikeThis did sometimes provide a higher experience than CodeExchange to cover cases in which CodeExchange did not score higher than the baseline.

We next compare the median experience scores of the iterative approaches to Google in Table 22 by looking at contingency tables between the iterative approaches and Google. The first row shows the contingency tables between the iterative approaches and Google, just as reported in Table 21. We find that there exists an iterative approach providing a higher score than Google on 3 occasions, equal on 2 occasions, and on 3 occasions both iterative approaches score lower than Google. These results suggest that an iterative design to search can provide an equal or better user experience compared to Google. When we examine the results by each iterative approach, we see that Google outperforms each on five occasions, significantly so for CodeExchange, but interestingly not so for CodeLikeThis.

In many ways, these results are not necessarily surprising, given Google's much larger index and high performing ranking algorithm, RankBrain. Perhaps most impactful, however, is people's general familiarity with Google; it is just something they know how to use. Overall, Google provided a good experience for the participants, but our results point to appears to

be occasions where an iterative approach would have provided an even better or equal search experience for the user. Interestingly, the results between the baseline and CodeExchange suggest that adding iterative features to Google would provide an even better user experience for using Google to find code. That is, when the ranking algorithm and index are the same between two search engines (as with the baseline and CodeExchange) adding iterative features to one can significantly improve the user experience.

Table 21. Experience Medians of Iterative Approaches Compared to Base Line.

Iterative and Base Line		Actual			
		W	***T***	***L***	***Total***
	I	6	2	0	8
	BL	0	2	6	8
	Total	6	4	6	16
		Expected			
		W	***T***	***L***	***Total***
	I	3	2	3	8
	BL	3	2	3	8
	Total	6	4	6	16
				χ^2	
				p=	0.00247875

CodeExchange and Base Line		Actual			
		W	***T***	***L***	***Total***
	CE	6	1	1	8
	BL	1	1	6	8
	Total	7	2	7	16
		Expected			
		W	***T***	***L***	***Total***
	CE	3.5	1	3.5	8
	BL	3.5	1	3.5	8
	Total	7	2	7	16
				χ^2	
				p=	0.02811566

CodeLikeThis and Base Line		Actual			
		W	***T***	***L***	***Total***
	CLT	3	3	2	8
	BL	2	3	3	8
	Total	5	6	5	16
		Expected			
		W	***T***	***L***	***Total***
	CLT	2.5	3	2.5	8
	BL	2.5	3	2.5	8
	Total	5	6	5	16
				χ^2	
				p=	0.81873075

Table 22. Experience Medians of Iterative Approaches Compared to Google.

		Actual			
		W	***T***	***L***	***Total***
Iterative and Google	***I***	3	2	3	8
	G	3	2	3	8
	Total	6	4	6	16
		Expected			
		W	***T***	***L***	***Total***
	I	3	2	3	8
	G	3	2	3	8
	Total	6	4	6	16
				χ^2	
				p=	1
		Actual			
		W	***T***	***L***	***Total***
CodeExchange and Google	***CE***	1	2	5	8
	G	5	2	1	8
	Total	6	4	6	16
		Expected			
		W	***T***	***L***	***Total***
	CE	3	2	3	8
	G	3	2	3	8
	Total	6	4	6	16
				χ^2	
				p=	0.06948345
		Actual			
		W	***T***	***L***	***Total***
CodeLikeThis and Google	***CLT***	2	1	5	8
	G	5	1	2	8
	Total	7	2	7	16
		Expected			
		W	***T***	***L***	***Total***
	CLT	3.5	1	3.5	8
	G	3.5	1	3.5	8
	Total	7	2	7	16
				χ^2	
				p=	0.27645305

To study the participants' experiences with iterative approaches in more detail, we compared experiences scores for CodeExchange with CodeLikeThis by kinds of tasks. Interestingly, we find that CodeExchange and CodeLikeThis are complementary in the kinds of tasks they support. Table 23 presents the results, where each cell shows which iterative approach had a higher experience median for each task. The table shows that CodeExchange provided a better experience for tasks that were broader ("Find 4" and "No Specific Role") and that CodeLikeThis provided a better experience for tasks that were more focused ("Find 1" and "Algorithm/Data Structure"). The data in Table 23 suggests that using aspects or qualities of the results, supported by CodeExchange, helps in finding code when the search task is broad, but, when the search task is more focused, then using the entire result to search with, as supported by CodeLikeThis, provides a better search experience. A complementary pattern of support for the two opposing kinds of tasks (broad and focused), as shown in Table 23, only has a 16/6561 (.2%) chance of occurring (given equal probability of CodeExchange and CodeLikeThis occurring in each cell as well as both occurring in each cell for ties). Illustrating the rarity of such an occurrence, Figure 72 provides a visualization of the total space of possible patterns that could have occurred between CodeExchange and CodeLikeThis (annotated with examples). Each dot represents a possible outcome (6561 in total), where the darker dots (5709 in total) represent non-complementary patterns, the

Table 23. CodeExchange and CodeLikeThis Median Comparison.

	No Specific Role		**Algorithm/Data Structure**	
Find 4	CE(T3)	CE(T7)	CE(T1)	CLT(T5)
Find 1	CE(T6)	CLT(T2)	CLT(T4)	CLT(T8)

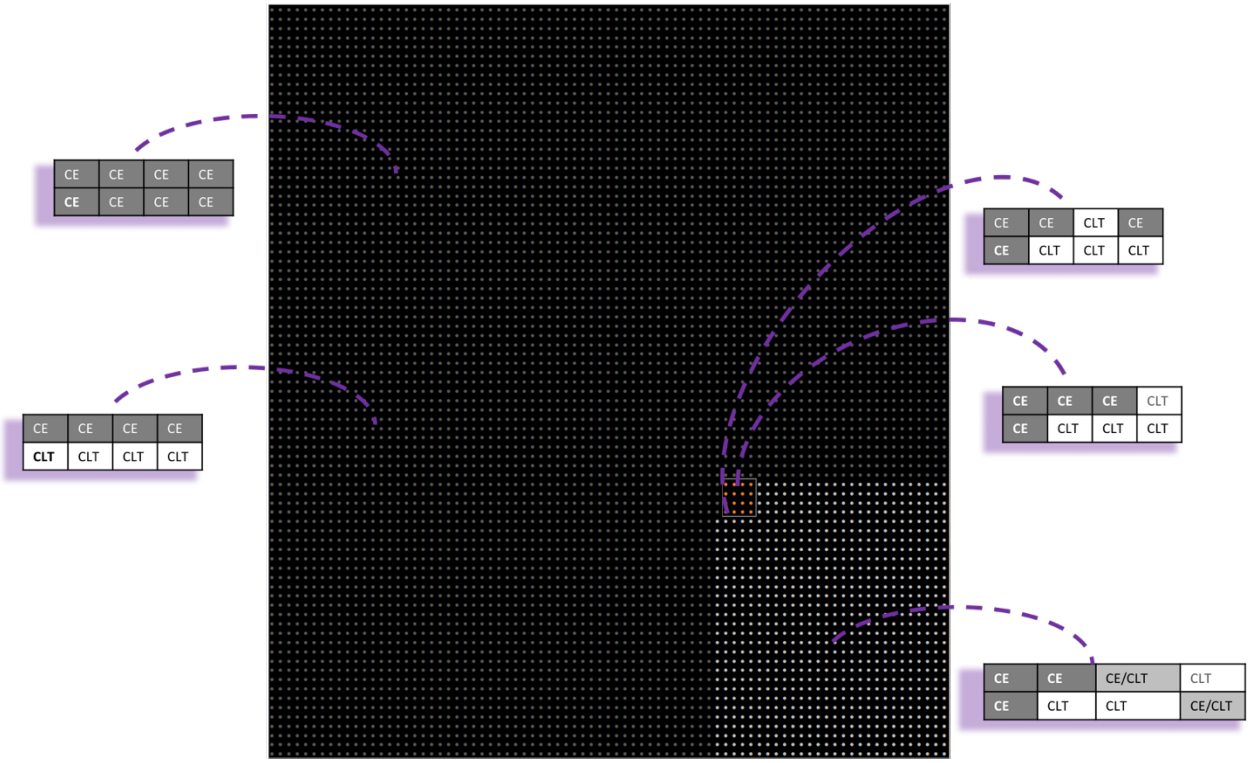


Figure 72. Space of Possibilities.

white dots (836 in total) represent complementary patterns, but are not "perfect complements" because there exist ties in the cells, and the red dots (16 in total) represent complementary patterns with no ties. Complementary patterns with ties are not "perfect", because it means that one of the search engines will provide a better experience than the other but not as much as found in a complementary pattern with no ties. For example, the complementary pattern illustrated in Figure 72 shows one possible outcome where CodeLikeThis provided a better experience four times and CodeExchange two times for the Algorithm/Data Structure category (4/2 = 2). However, in a complementary pattern with no ties where CodeLikeThis provided a better experience for the Algorithm/Data Structure category, the outcome would be 3/1, which provides a higher experience more times than what would be found in a complementary pattern with ties. In general, complementary patterns with ties will always have a search engine providing a higher experience on a row

Table 24. CodeExchange and Google Median Comparison.

	No Specific Role		Algorithm/Data Structure	
Find 4	CE and G(T3)	G(T7)	CE(T1)	CE and G(T5)
Find 1	G(T6)	G(T2)	G(T4)	G(T8)

or column that is less than 3/1. On the other hand, a complementary pattern with no ties will always have one search engine providing a better experience ratio of 3/1 on one row and column and the other search engine providing a better experience ratio of 3/1 on the other row and column. The illustration in Figure 72 is intended to show how rare a "perfect" complementary pattern is and how it makes the "perfect" complementary pattern we found between CodeExchange and CodeLikeThis less of a chance occurrence.

The complementary nature of CodeExchange and CodeLikeThis appears to be why that, when combined, the scores for the iterative approaches are more balanced with Google. When we compare CodeExchange with Google by experience scores, as shown in Table 35, we find that CodeExchange performs at its best against Google in the Find 4 category. Similarly, when we compare CodeLikeThis with Google by experience scores, as shown in Table 36, we find CodeLikeThis performs at its best against Google in the Find 1 and

Table 25. CodeLikeThis and Google Median Comparison.

	No Specific Role		Algorithm/Data Structure	
Find 4	G(T3)	G(T7)	G(T1)	CLT(T5)
Find 1	G(T6)	CLT(T2)	CLT and G(T4)	G(T8)

Table 26. Iterative Approaches Median Comparison.

	No Specific Role		Algorithm/Data Structure	
Find 4	I and G(T3)	G(T7)	I(T1)	I(T5)
Find 1	G(T6)	I(T2)	I and G(T4)	G(T8)

Algorithm/Data Structure Category. When we compare the Iterative approaches, combined, with Google by experience scores, as shown in Table 38, we find that the experience scores are balanced across Google and the Iterative approaches, where Google had higher scores in Find 1 (3 to 2) and No Specific Role (3 to 2), and where the Iterative approaches had higher scores in Find 4 (3 to 2) and Algorithm/Data structure (3 to 2).

We also look at the Iterative approaches and the baseline by category. CodeExchange and the baseline search engine are compared by category in Table 27. We find that CodeExchange provided a better search experience across all categories. These results are especially telling because the major difference between CodeExchange and the baseline search engine are the iterative features provided by CodeExchange, suggesting the iterative features alone caused the improvement in experience. The index and ranking algorithm are exactly the same in both search engines.

Table 27. CodeExchange and BaseLine Comparison.

	No Specific Role		Algorithm/Data Structure	
Find 4	CE(T3)	CE(T7)	CE(T1)	BL(T5)
Find 1	CE and BL(T6)	CE(T2)	CE(T4)	CE(T8)

Table 28. CodeLikeThis and Baseline Comparison.

	No Specific Role		Algorithm/Data Structure	
Find 4	BL(T3)	CLT and BL(T7)	CLT and BL(T1)	CLT and BL(T5)
Find 1	BL (T6)	CLT(T2)	CLT(T4)	CLT(T8)

In contrast with CodeExchange, CodeLikeThis does not provide a better search experience across all categories against the baseline, as shown in Table 28. In particular, the baseline appears to have provided a better experience in the Find 4 (4 to 3) and No Specific Role categories (3 to 2). However, CodeLikeThis again excels in the Find 1 (3 to 1) and Algorithm/Data Structure (4 to 2) categories. This is consistent with the other findings above that CodeLikeThis appears to be better on more focused tasks.

6.2.2 Task Times

Time differences for finding code by search engines were examined by measuring how long it took before code was pasted for each of the tasks by search engine (time writing why/what explanations was not counted). If the task was not completed, the maximum time allotted was used. We used ANOVA to find significant differences among the groups of search engines for each task, and if we did find a significant difference, we conducted a post hoc pair-wise analysis on the corresponding group using Tukey's honest significant difference [45] to see what might be causing the difference.

Table 29. Mean Seconds Until First Paste.

	T1	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**
CE	219 *(s=105)*	294 *(s=188)*	167 *(s=152)*	392 *(s=147)*	488 *(s=386)*	263 *(s=150)*	301 *(s=207)*	372 *(s=223)*
CLT	185 *(s=109)*	302 *(s=167)*	359 *(s=253)*	357 *(s=153)*	373 *(s=273)*	285 *(s=159)*	103 (CLT>G p=0.083) *(s=21)*	189 *(s=204)*
BL	163 *(s=96)*	178 *(s=60)*	174 *(s=95)*	446 *(s=182)*	154 *(s=54)*	278 *(s=122)*	158 *(s=99)*	301 *(s=176)*
G	176 *(s=97)*	244 *(s=175)*	85 (G>CLT p=0.03) *(s=43)*	393 *(s=216)*	410 *(s=283)*	182 *(s=109)*	349 *(s=243)*	169 *(s=102)*
ANOVA	p=.8	p=.5	p=0.04	p=.8	p=.2	p=.5	p=0.06	p=.2

Our time analysis is presented in Table 42 for finding the first snippet (standard deviation abbreviated as "s"). In most cases, we did not find significant differences. However, we did find that Google was significantly faster than CodeLikeThis for finding the first code snippet on task three and CodeLikeThis was significantly faster than Google for finding the first code snippet on task seven. The lack of many significant differences in time might be due to the large variance we find in the data (which affects finding statistical significance [136]), making the mean time less representative of the times collected. Figure 73 shows the spread of the times for each task by search engine. We find that, for many of the tasks, several of the search engines have a wide spread of data around their mean, making it harder to find statically significant differences of the means. One possible explanation for the spread in the data is that the tasks were more exploratory in nature and that we gave the participants no incentive to finish before the allotted time. However, we found time differences that, perhaps, are worth noting, for example, the time difference found between the baseline and

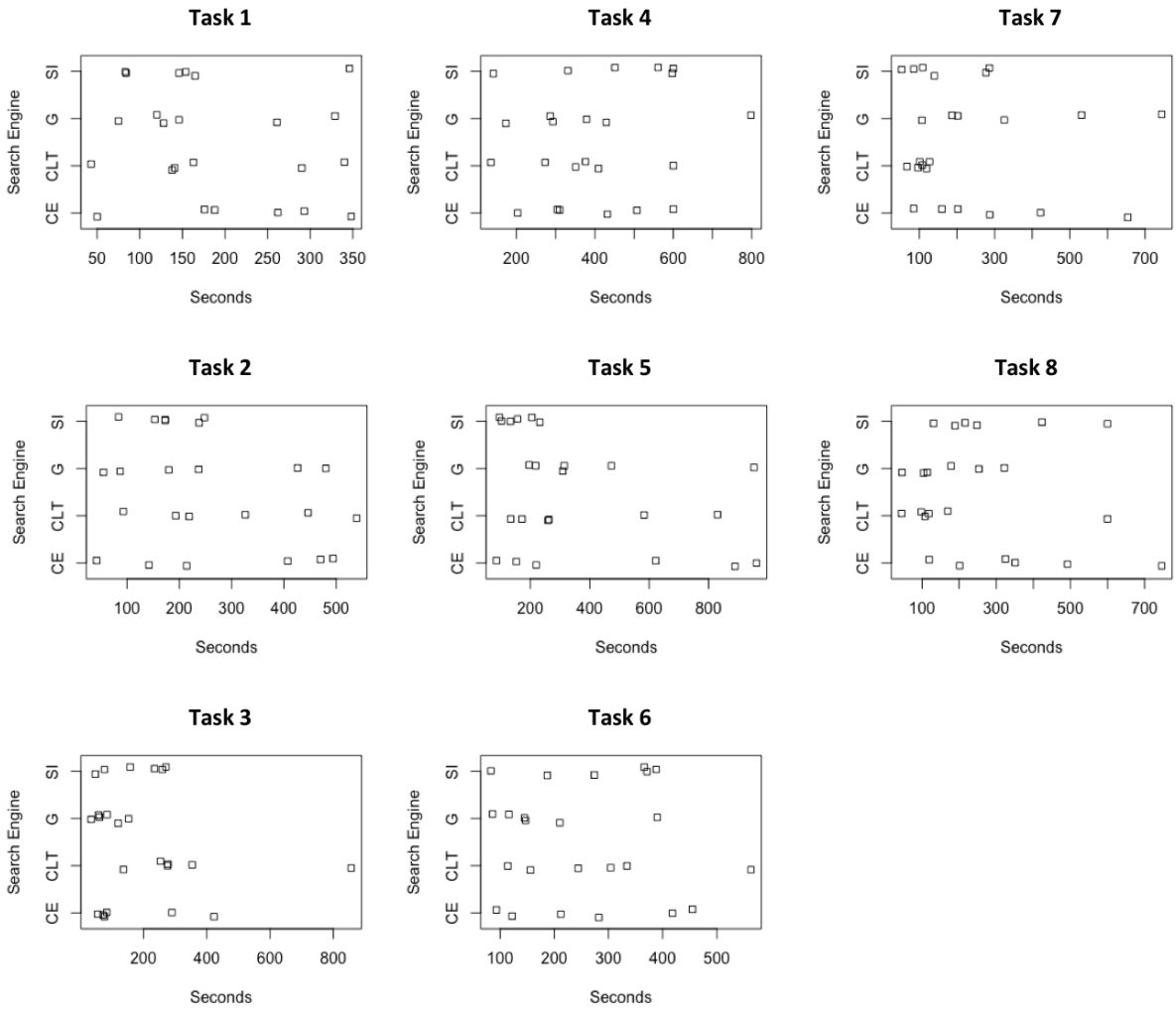


Figure 73. Spread of Time till First Paste by Search Engine and Question.

the other search engines for task 5. We found the baseline had a mean time for task 5 twice as fast as any of the other search engines. In the upcoming section, we explore the significance of time in search.

Similarly, we find no statistical significant differences in time for the Find 4 tasks. In Table 30 we present the time analysis for the Find 4 tasks (standard deviation abbreviated as “s”). The spread of the data (as shown in Figure 74), again, might explain why we did not find any

Table 30. Mean Seconds Until Fourth Paste.

	T1	T3	T5	T7
CE	618 (s=349)	536 (s=259)	925 (s=378)	865 (s=366)
CLT	675 (s=182)	750 (s=251)	808 (s=237)	852 (s=390)
BL	567 (s=184)	572 (s=376)	676 (s=421)	840 (s=347)
G	611 (s=249)	528 (s=397)	943 (s=283)	902 (s=334)
ANOVA	p=.9	p=.62	p=.51	P=.99

statistical significant differences. There are times, again, that some search engines have a mean time that is faster by a few minutes than others. For example, baseline appears a few minutes faster than Google and the other search engines for task 5. However, the baseline often led to a worse search experience in comparison to the other search engines. These differences prompt us to take a closer look at the role of speed on code search in the upcoming section.

A similar story of finding no statistical significance holds true by task type as well. The large spread of data we saw in individual tasks holds when clustered by type as well. Table 31 reports the mean time to find the first snippet of code for all task types and Figure 75 shows the spread of the data. Figure 76 shows the spread of the data to finish the Find 4 tasks and,

Table 31. Mean Time till First Paste by Task Type.

	Find 4	**Find 1**	**No Specific Role**	**Algorithm/Data Structure**
CE	294 *(s=254)*	330 *(s=176)*	256 *(s=173)*	368 *(s=245)*
CLT	255 *(s=216)*	283 *(s=172)*	262 *(s=187)*	276 *(s=203)*
BL	162 *(s=83)*	301 *(s=166)*	197 *(s=102)*	266 *(s=177)*
G	255 *(s=224)*	247 *(s=173)*	215 *(s=179)*	287 *(s=213)*
ANOVA	p=0.157	p=0.447	p=0.446	p=0.329

again, we find a spread of times with no statically significant difference, as shown in Table 32.

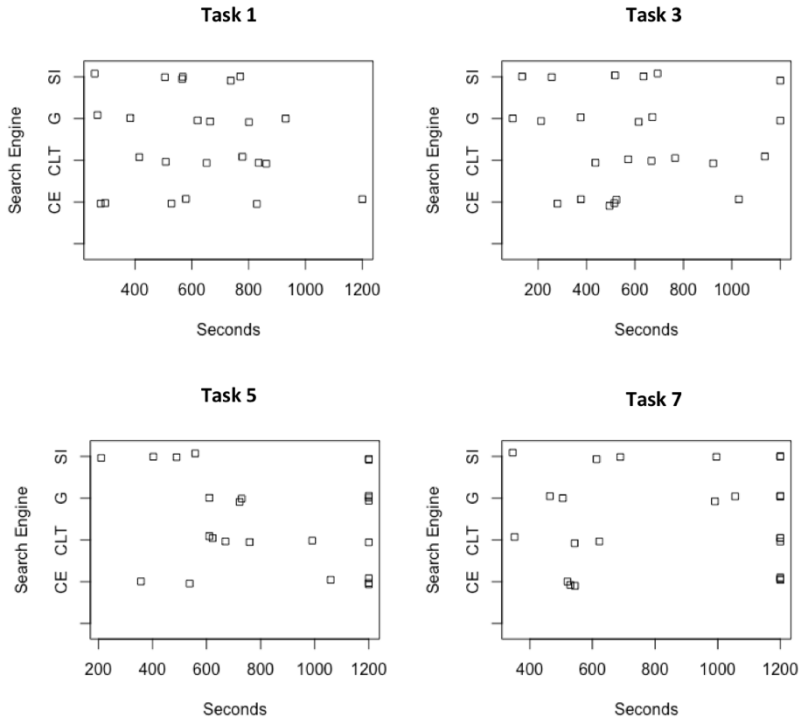


Figure 74. Spread of Time till Fourth Paste by Search Engine and Question.

We briefly considered if removing outliers would help narrow down the variance in our data, however it raises several difficulties. The main challenge of removing outliers when the data is categorized is that removing points with methods like symmetrical truncation (as we did in Chapter 4) leads to uneven amounts of data in categories and biases the comparisons. Further, because the data is generally spread out, removing points toward the extremes are

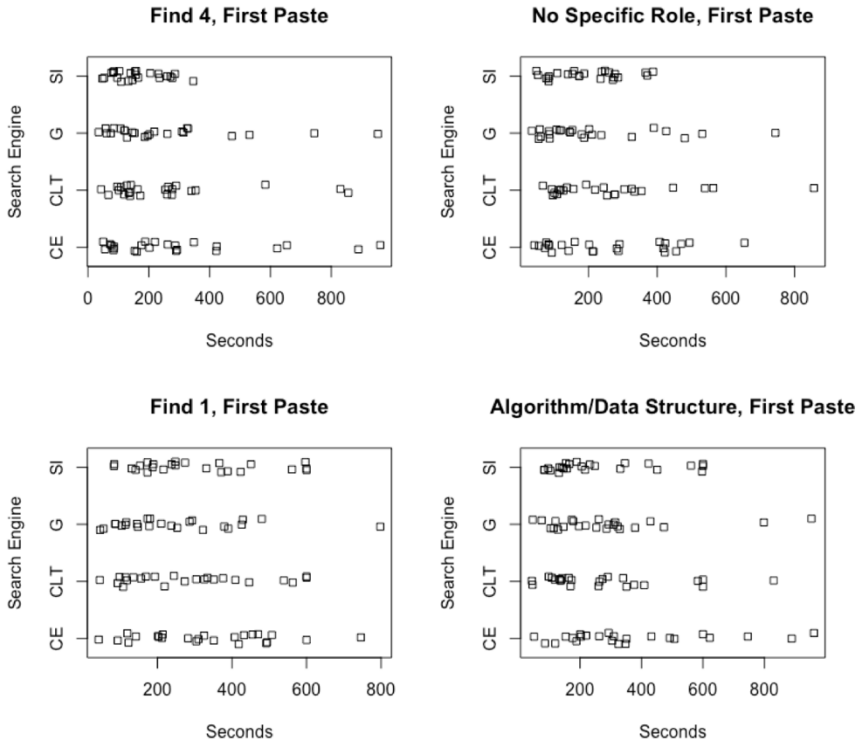


Figure 75. Spread of Time till First Paste by Search Engine and Task Type.

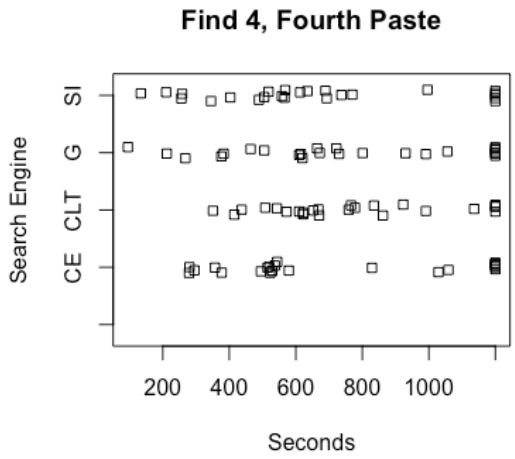


Figure 76. Spread of Time till Fourth Paste by Search Engine and Find 4 Task Type.

unlikely to impact the mean very much. For these reasons, we consider all the data in our analysis and take a closer look at the role of time in searching for code.

6.2.2.1 Experience and Task Times

Experience, in many ways, is a critical metric for a tool, and, thus, for search engines. If the user is frustrated, confused, exhausted, or otherwise left not satisfied with a search engine, then they will be quick to find a replacement. Time, at face value, would also seem critical, because the user will certainly not want to waste their time. However, more time to search with a search engine may not always mean a worse search engine if it is supporting the user to do what they need to do. Consider the baseline search engine. We found that CodeExchange provided a better experience significantly more often than the baseline.

Table 32. Mean Time till Fourth Paste.

	CE	**CLT**	**BL**	**G**
Find 4	736 *(s=359)*	771 *(s=266)*	664 *(s=340)*	746 *(s=351)*
ANOVA	p=.709			

However, the baseline does appear faster than CodeExchange sometimes, not necessarily significantly, but still faster. Perhaps the participants settled, unsatisfactorily, for code sooner with the baseline, or perhaps the features of CodeExchange encouraged the user to explore more. In this section, we look at when time might play a role for the user in terms of the impact it has on the user experience.

In Figure 77, we present a correlation analysis (using Pearson's method) between experience and time to paste first snippet by task type. The correlation scores appear at the bottom left with the correlation strength beside it. The red line through the graphs is the regression line showing the slope of the correlation. The top row shows the results for the broader tasks and the bottom row shows the results for the more focused tasks. Interestingly, we find no correlation between experience and time when the tasks are

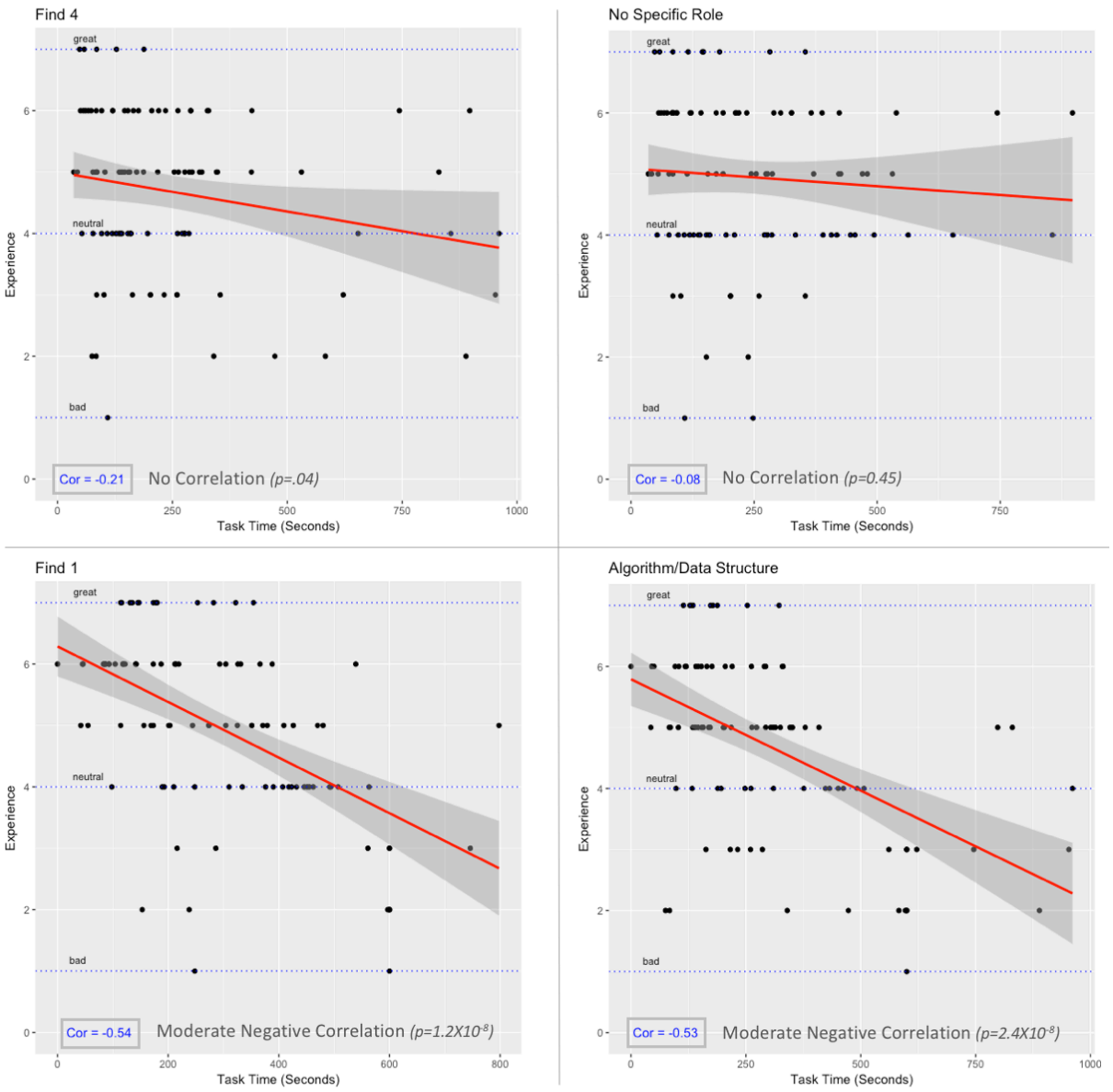


Figure 77. Experience and Time to First Paste Correlation by Task Type.

broader, but when they are more focused we do indeed find a correlation. The results suggest that time may be less of a factor or other factors might be at play for impacting the user's search experience when they have less of an idea of what they are looking for and are engaged in a more exploratory search. However, when the user has a clearer idea of what it is they want, then finding it faster makes it a better experience for them. For the more focused tasks, the regression line crosses the neutral experience line and into the bad experience region at around 500 seconds (≈ 8.3 minutes), which suggests that when the user has a more focused search task, their search experience will turn negative after about 8 minutes of searching.

Figure 78 presents the correlation analysis between experience and time to paste the fourth snippet. We find a weak correlation between time and experience for finding four code

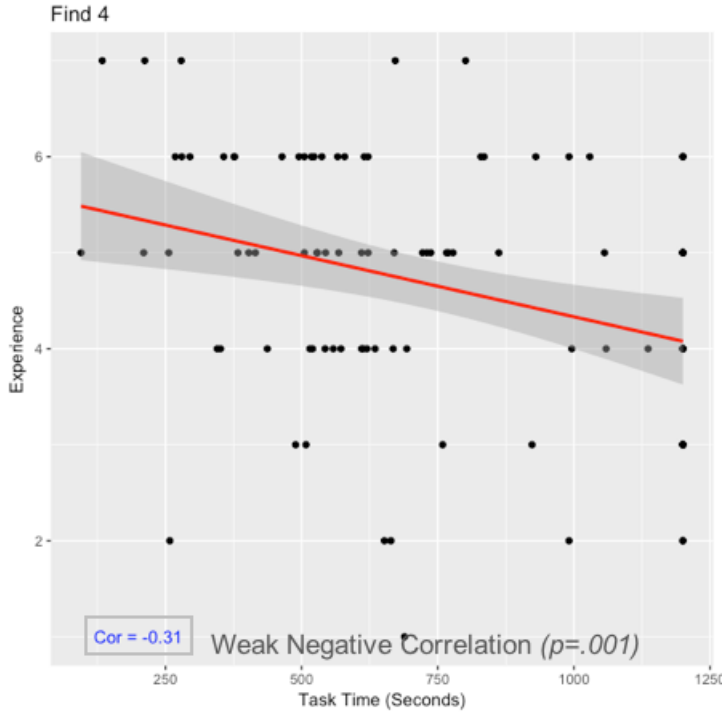


Figure 78. Experience and Time to Fourth Paste Correlation.

snippets. This is consistent with the little correlation we found between time and experience for finding the first code snippet for broader tasks, and further suggests time may be less of a factor or other factors might be at play that impact the user's search experience when the tasks are broader.

Figure 79, Figure 80 and Figure 81 present the experience and time correlations by task. Not surprisingly, we find tasks with both negative correlations and no correlations, as one would expect given our previous results. However, Task 7 yielded a positive correlation between experience and time. That is, for Task 7, it appears that the more time the participants searched, the better their experience was. Task 7 is in the No Specific Role and Find 4 categories, so it is one of the broadest search tasks we designed. This positive correlation result suggests that sometimes when the search tasks are more exploratory, taking more time might yield a better search experience.

While we have looked at correlations between experience and time by task type and have found little to no correlation with time on broader tasks and a negative correlation for more focused tasks, we next look to see if the same is true for each search engine. Figure 82, *Figure 83*, and Figure 84 present correlation results between experience and time by broader tasks and search engine. CodeLikeThis, Google, and the baseline all have no or weak correlations between search experience and time, which agrees with our previous findings that there is little to no correlation between the experience in searching and time when the search task is broader. However, we do find there is a strong and moderate correlation for CodeExchange. Upon closer inspection, we find that the majority or many of the scores tend to cluster in the

top left in the positive experience area (relatively shorter times and more positive experience scores). This correlation suggests that, while participants still appreciated saving time when searching with CodeExchange, the time they did spend searching with CodeExchange did not usually make them have a negative experience.

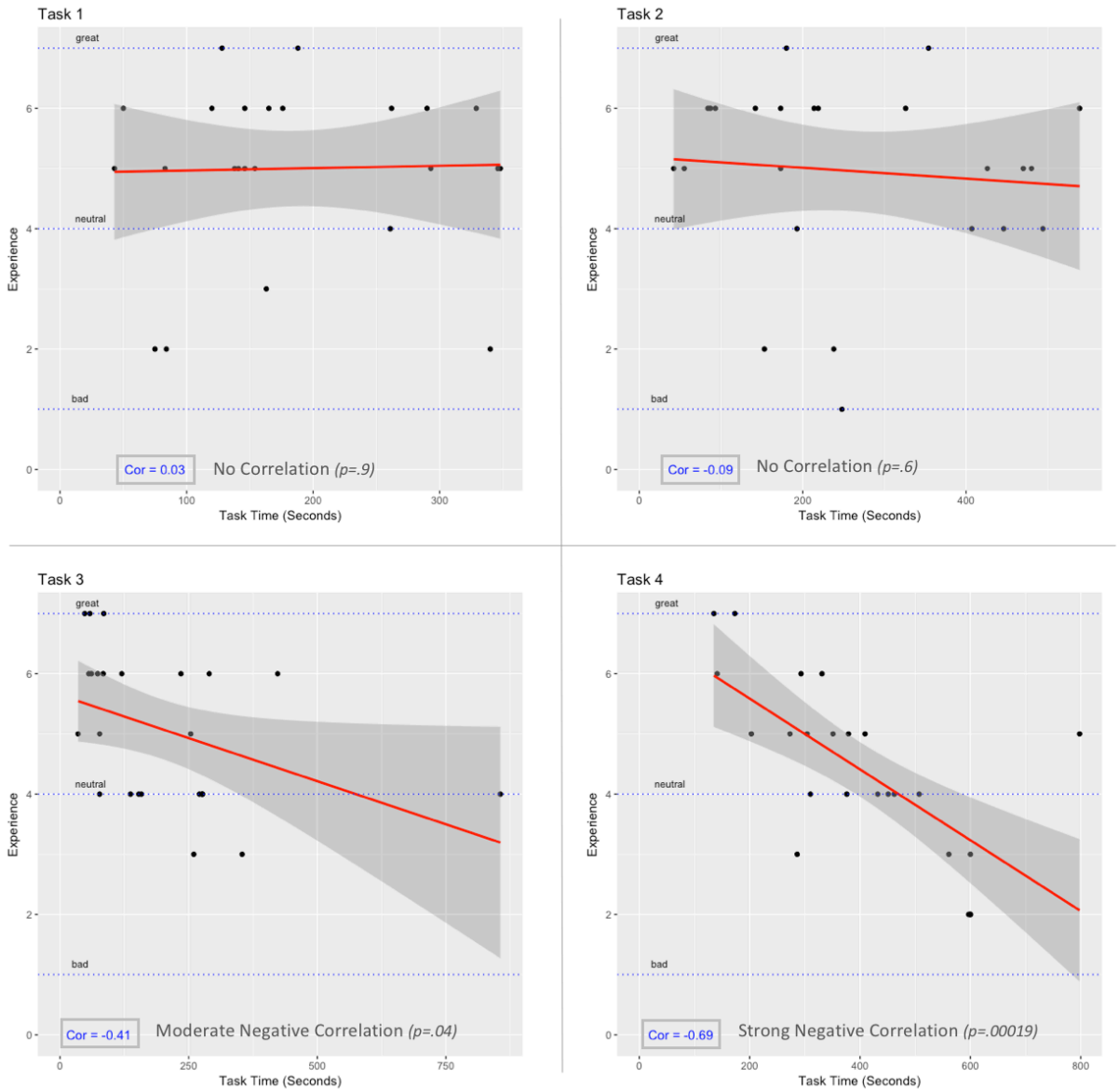


Figure 79. Experience and Time to First Paste Correlation Tasks 1 to 4.

Next we look at the experience and time correlations for more focused tasks by search engine. Figure 85 and Figure 86 present the correlation results between experience and time for more focused tasks by search engine. In all cases, we find that search engines tend to provide a better search experience when the users can complete the more focused tasks more quickly. This agrees with our results above that time may impact the search experience when the search tasks are more focused.

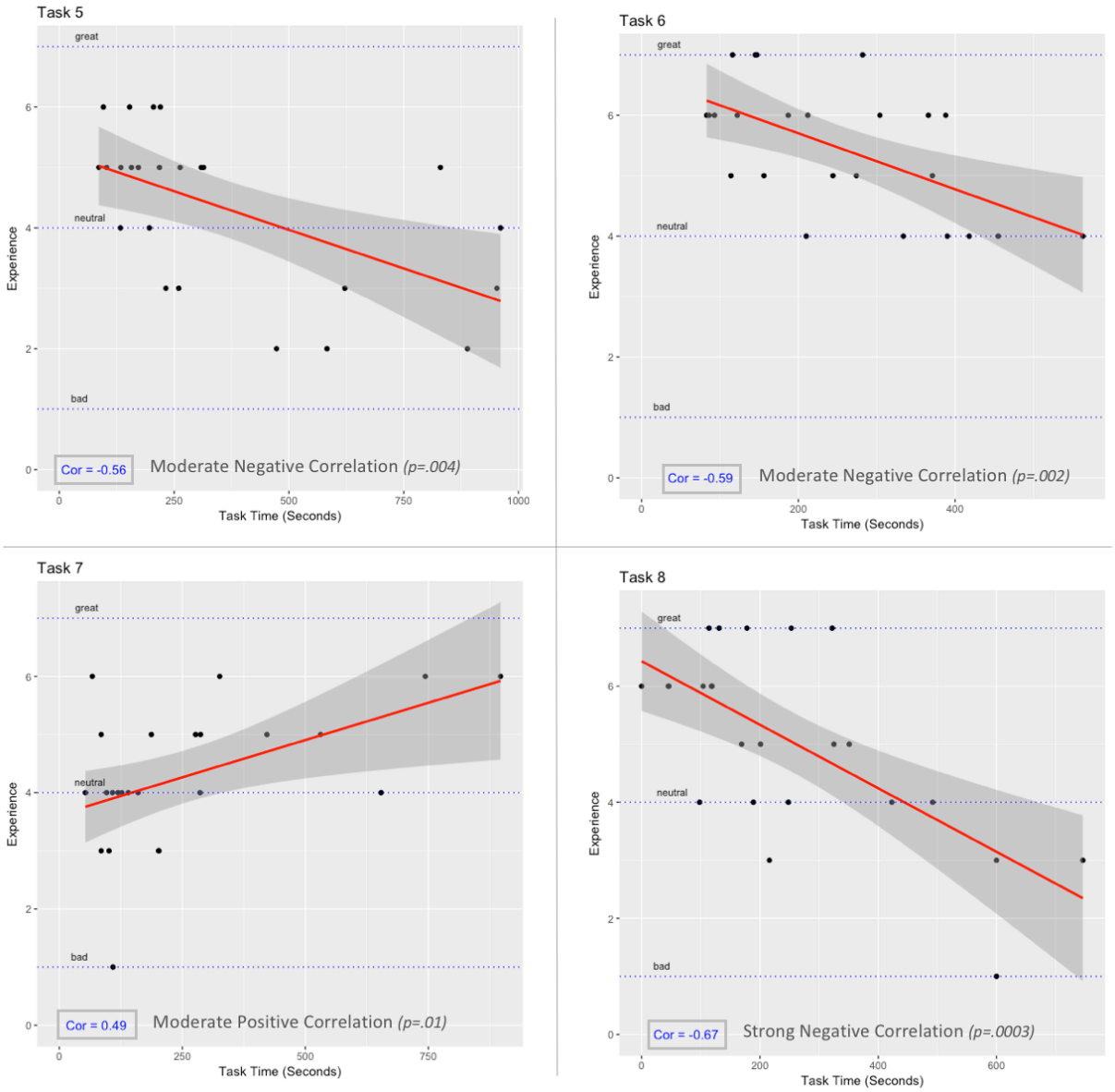


Figure 80. Experience and Time to First Paste Correlation Tasks 5 to 8.

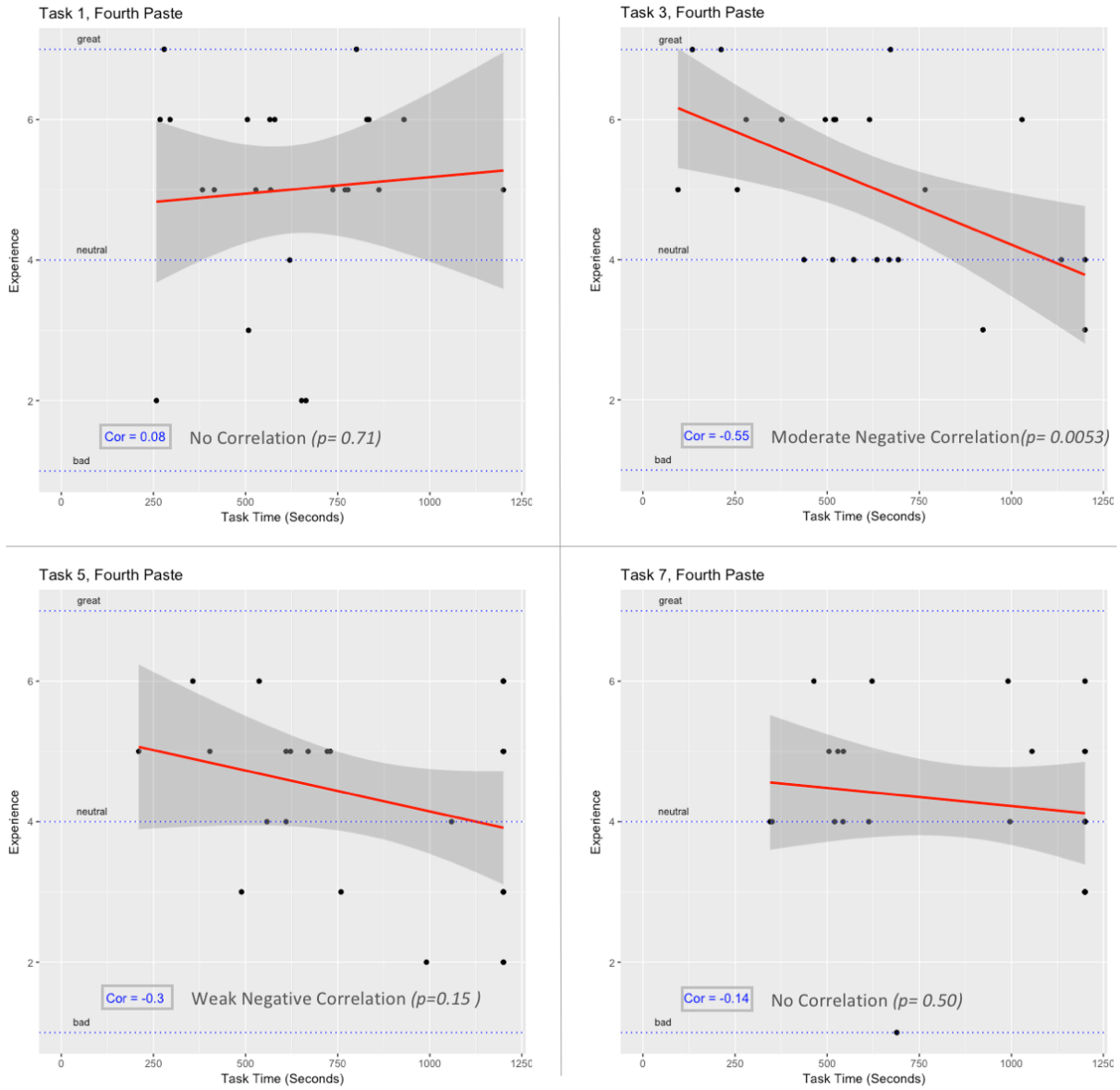


Figure 81. Experience and Time to Fourth Paste Correlation by Task.

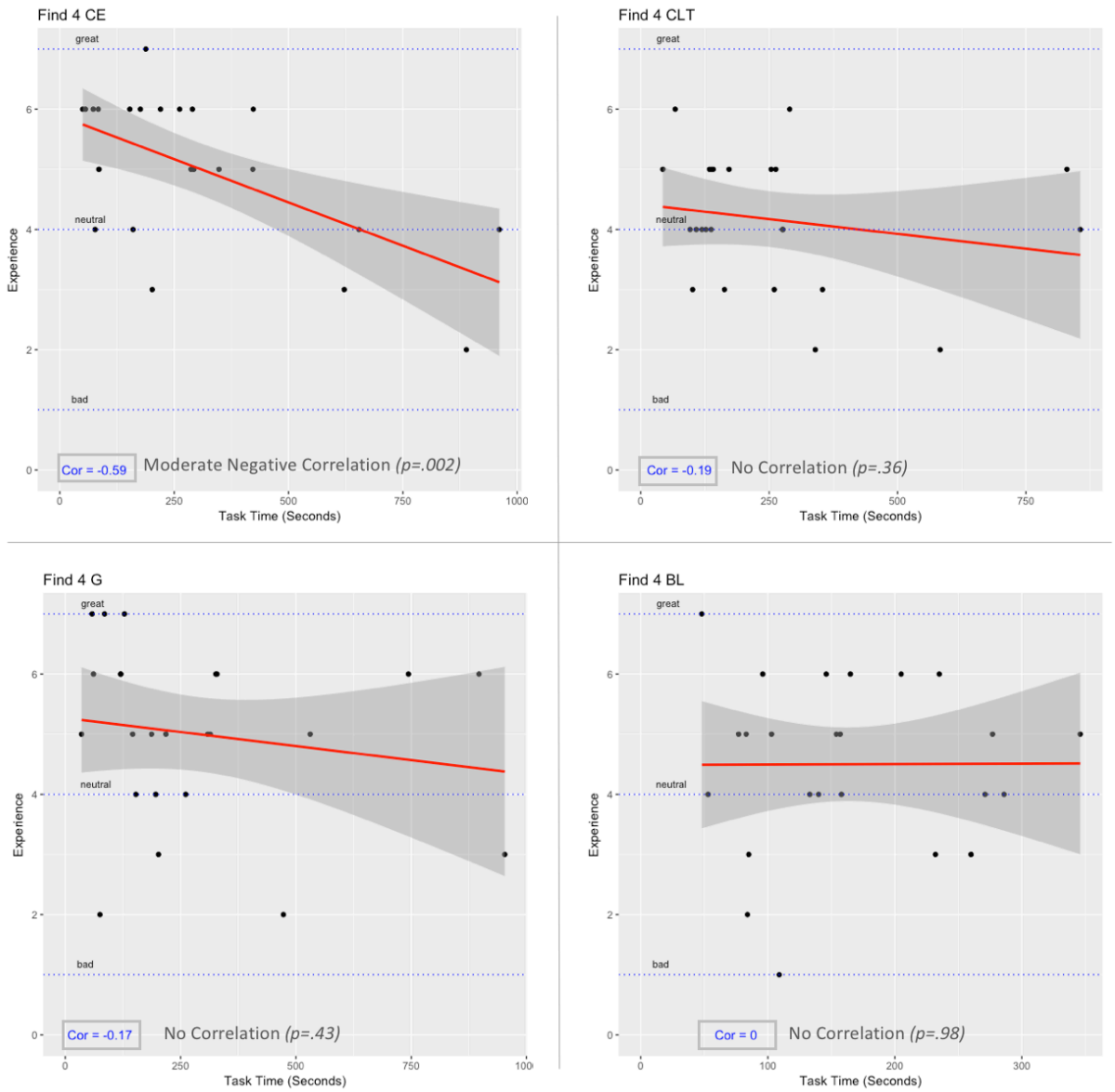


Figure 82. Find 4 and Time to First Paste Correlation by Search Engine.

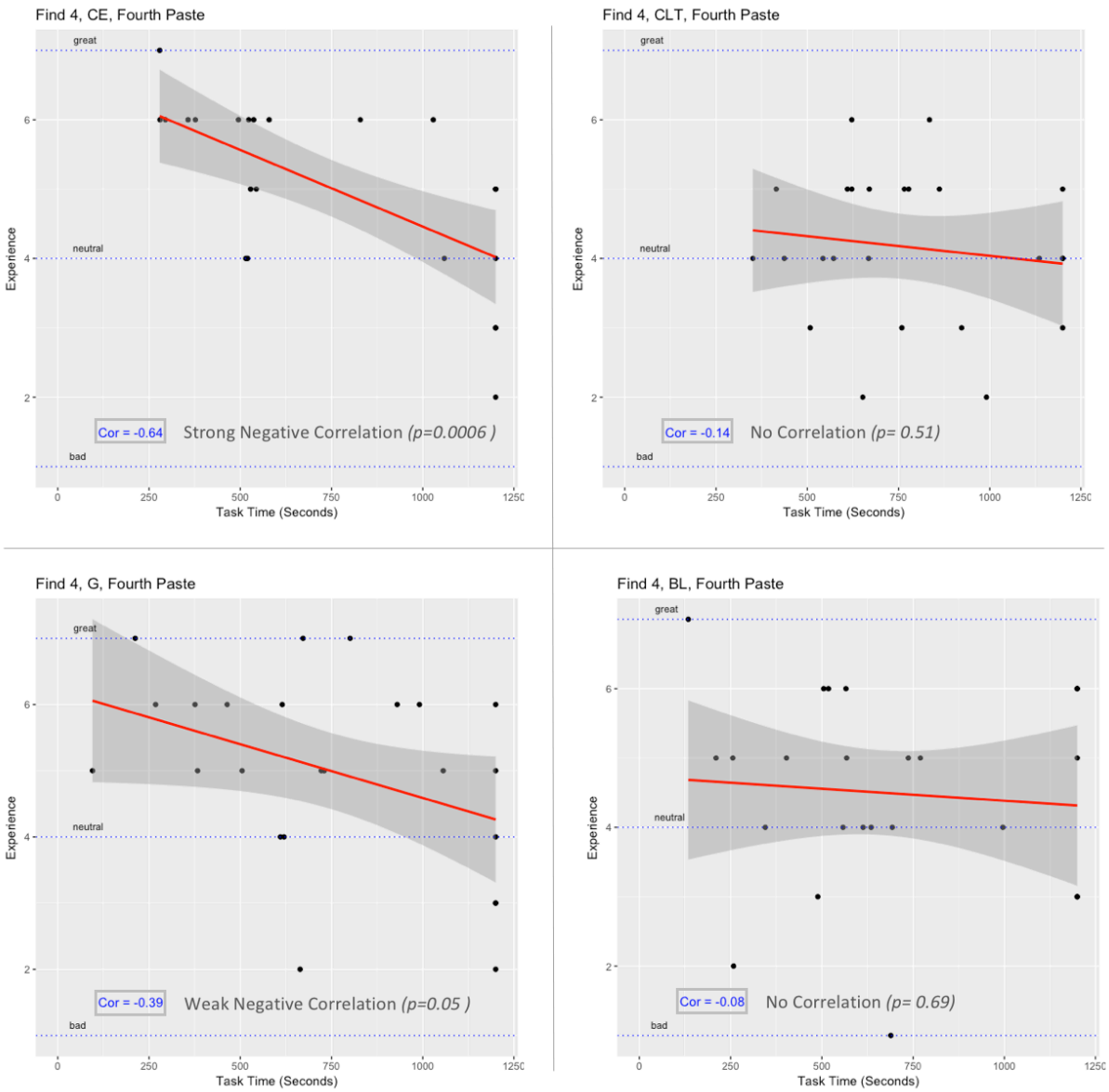


Figure 83. Find 4 and Time Fourth Paste Correlation by Search Engine.

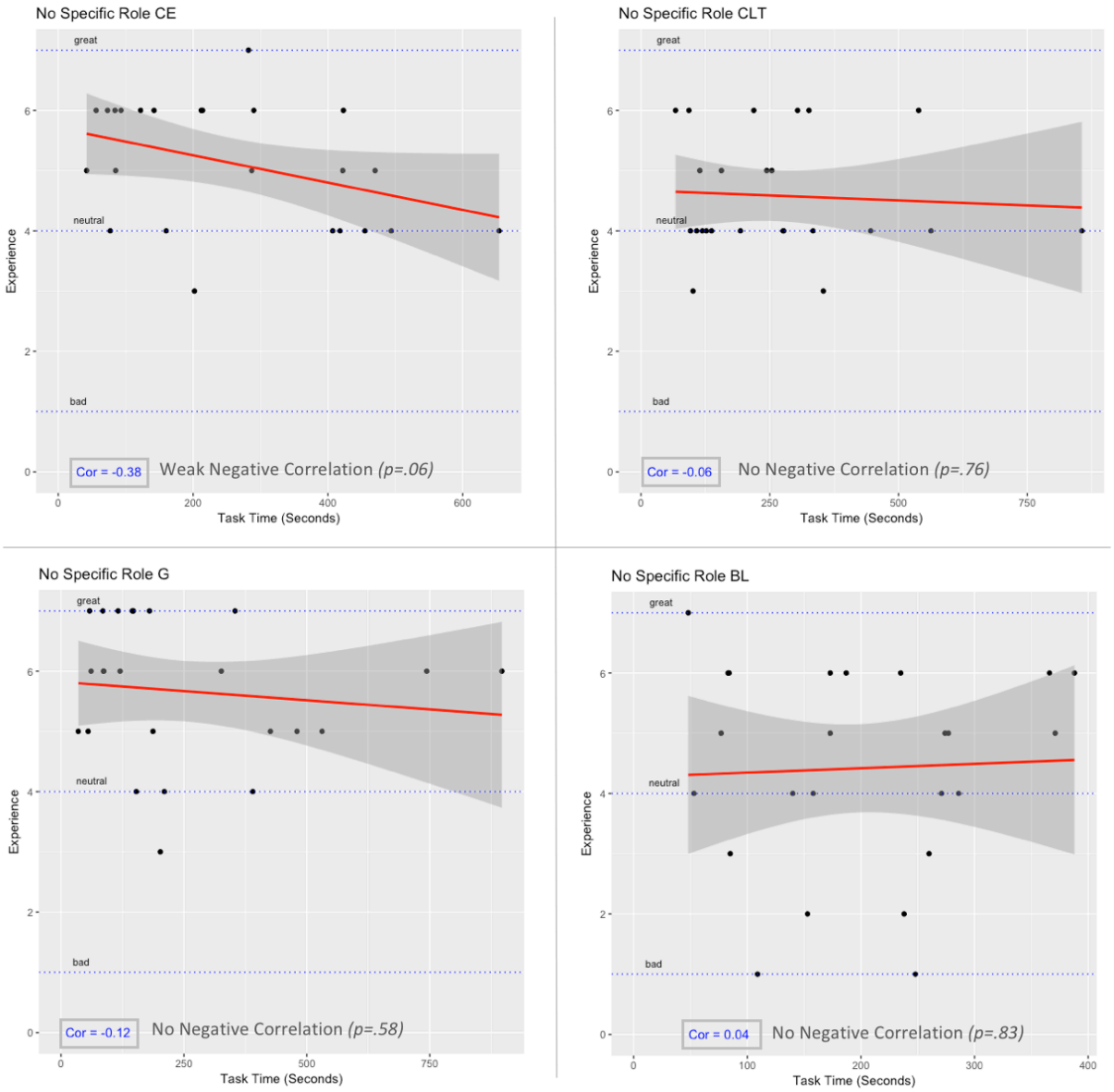


Figure 84. No Specific Role and Time to First Paste Correlation by Search Engine.

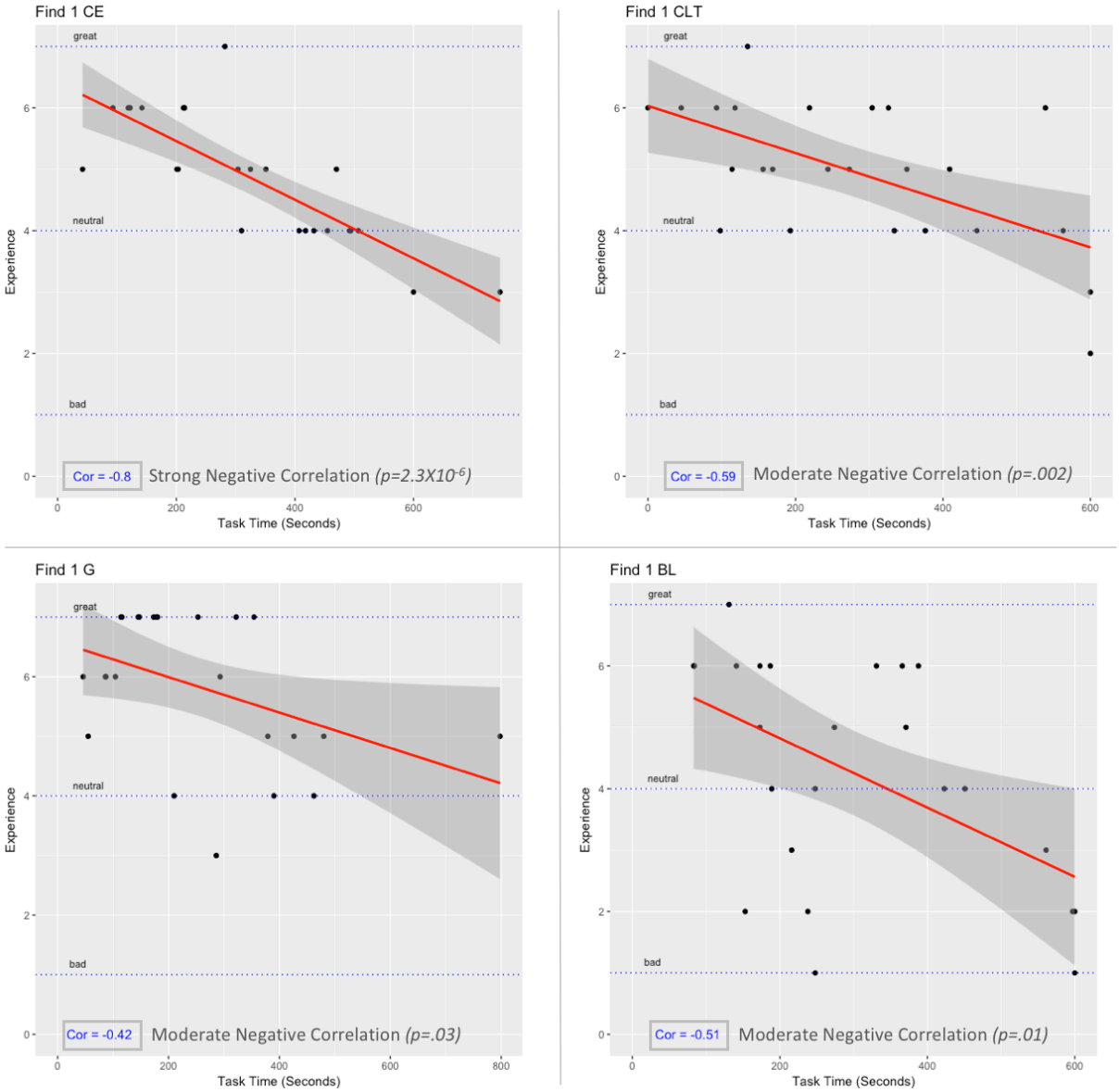


Figure 85. Experience and Time to First Paste Correlation by Find 1 and Search Engine.

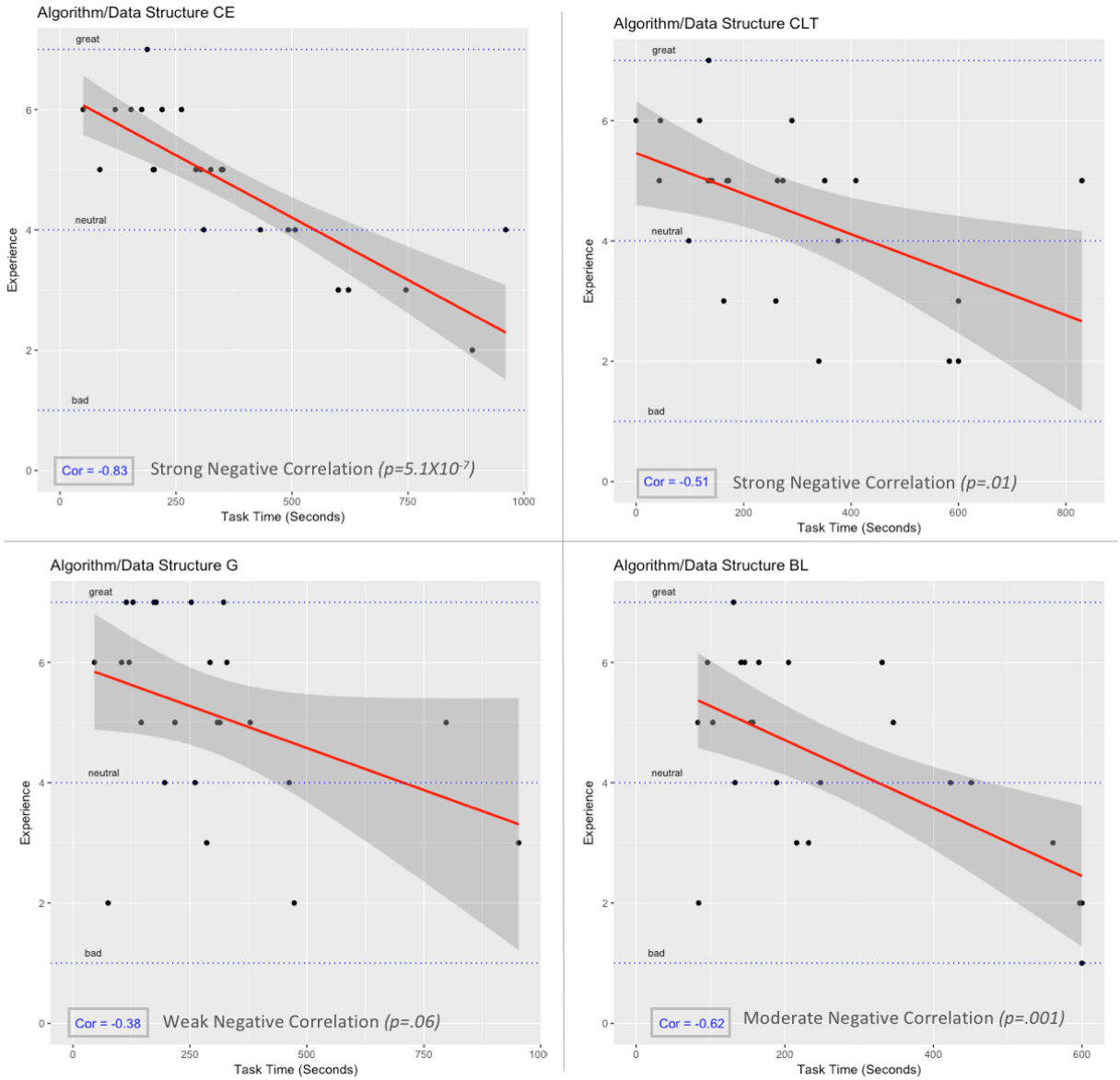


Figure 86. Experience and Time to First Paste Correlation by Algorithm/Data Structure and Search Engine.

The upshot of the time and experience analysis above is that time is indeed important, but other factors are at play that impact the participant's search experiences. The results showed that there is a negative linear relationship between how much time is spent searching and the search experience. However, spending more time to search for code does not necessary mean a negative experience, but it could mean just a lower positive experience. We saw CodeExchange tended to always provide a positive experience, just lower positive experience with time. Likewise, the baseline tended to provide neutral or negative experiences and with more time an even more negative experience. As such, on some occasions when participants finished faster with the baseline than CodeExchange, the participants still had a worse experience with the baseline. This means something else was creating a better search experience for participants using CodeExchange, even when the search took longer. We explore this more in the next section.

6.2.2.2 Number of Queries and Task Times CE versus BL

The difference between the baseline and CodeExchange are the iterative features CodeExchange provides, since the index and ranking algorithm are the same. In fact, the participants can use CodeExchange and the baseline in exactly the same way if they choose to ignore the iterative features and get exactly the same results. As such, the cause for the experience and time difference between CodeExchange and the baseline likely resides with the participants using the iterative features of CodeExchange. One possible explanation is that the iterative features supported the user in more easily issuing the next query, and, as

Table 33. Average Number of Queries per Task.

	T1	T2	T3	T4	T5	T6	T7	T8
CE	10	7.6	13.6	9.1	17.83	9.1	15.3	6.3
CLT	8.83	4.33	8.17	5.00	9.83	2.50	10.17	4.67
G	6.33	3	5.83	5.67	11.3	3	5.83	1.67
BL	7.17	3.83	4.67	3.5	9	3.17	7.17	6

such, made it easier to issue more queries than the baseline when the participant wanted to issue multiple queries to search for code. However, enabling the participants to issue more queries means they will look at more results, which takes more time.

To evaluate the plausibility that the participants were simply searching more with CodeExchange and therefore taking longer to search, we examined the average number of queries issued per task and kind of task for each search engine and present those results in Table 33 and Table 34. We find that, on average, participants issued 3 to 10 more queries per task with CodeExchange (more than any other search engine) than the baseline. In fact, for tasks T2, T3, T4, T5, T6, and T7 the participants issued 2 to 3 times as many queries with CodeExchange compared to the baseline, which means they had 2 to 3 times more results to look through. Further, we find issuing more queries with CodeExchange is moderately correlated with spending more time in searching (as shown in Figure 87). Based on the average number of queries issued and the positive correlation with number of queries issued

Table 34. Average Number of Queries per Task Type.

	Find 4	Find 1	No Specific	Algorithm/Data Structure
CE	14.21	8.08	11.46	10.83
CLT	9.25	4.13	6.29	7.08
G	7.33	3.33	4.42	6.25
BL	7.00	4.13	4.71	6.42

and time spent, the time difference between CodeExchange and the baseline appears to be because the participants searched more with CodeExchange. Further, since the participants did not have an issue more queries than the baseline with CodeExchange (based on their experience scores), them doing so suggests they wanted to spend more time searching and had a better experience doing so with CodeExchange. In later sections, we will dive deeper into how often features were used for all the search engines.

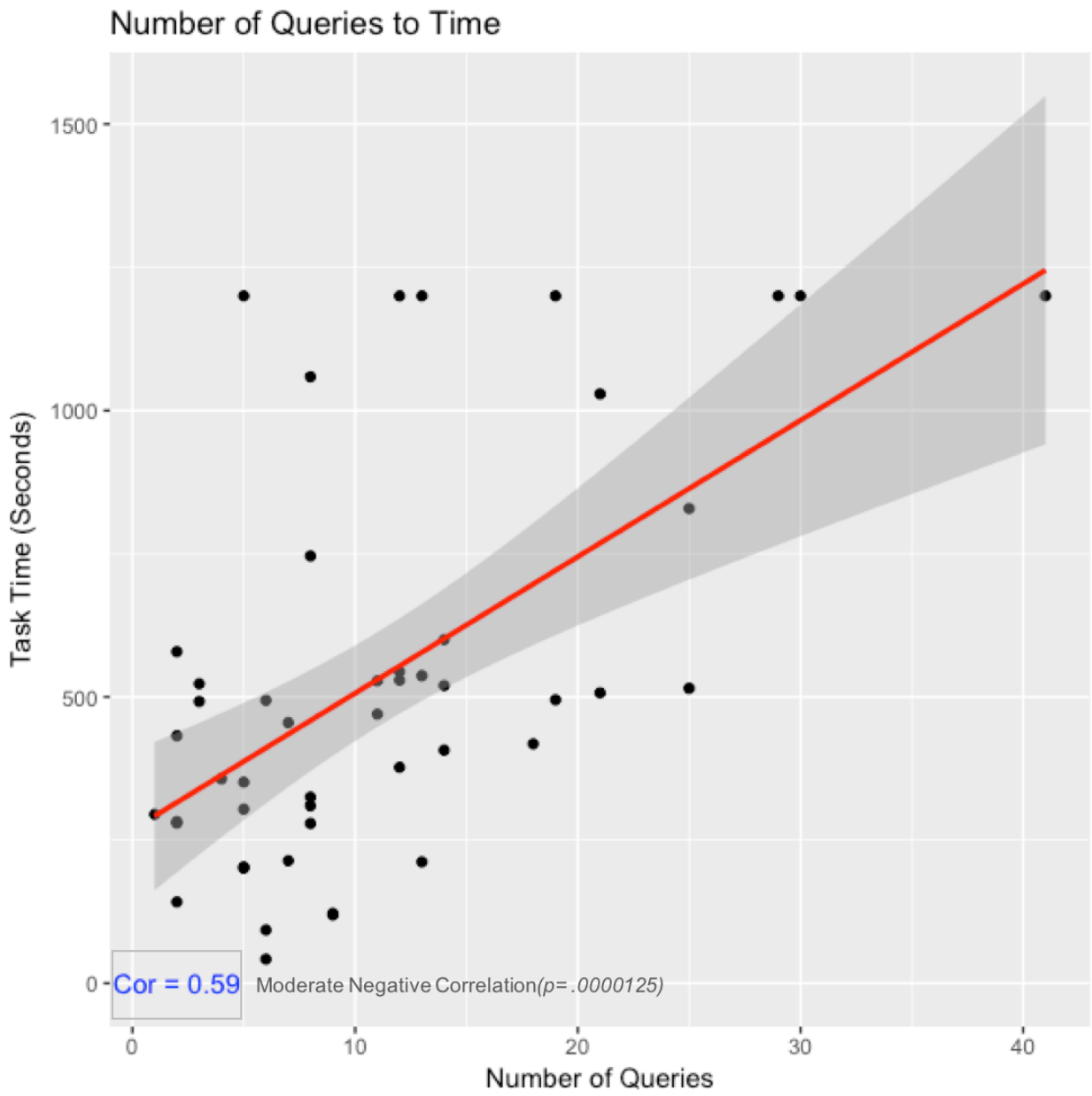


Figure 87. CodeExchange Time to Query Correlation.

6.2.3 Task Completion as a Success Measure

We next look at the number of task incompletions (not all code found) by search engine as a measure of unsuccessful searches and present the results in Table 35. We found that the iterative search approaches had less incomplete tasks than the others, with CodeLikeThis having the least number of incomplete tasks (4), baseline having 6, and CodeExchange and Google each with 7. However, we did not find these results statistically significant using $\chi 2$. Further, we found that the "Find 4" tasks had higher incompletion rates (19/96) than the "Find 1" tasks (5/96), and that the "Algorithms/Data Structure" tasks had a higher incompletion rate (16/96) than the "No Specific Role" tasks (8/96). As such, both broader tasks (Find 4) and more focused tasks (Algorithms /Data Structure) can prove more difficult to complete.

Similar to our previous analysis examining the correlation of experience and time, we now examine experience by task completion. In particular, we are interested in how success impacts experience because, in some sense, if not finishing a task does not lead to a worse

Table 35. Task Completion by Task and Search Engine.

	T1	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**Total**
CE	2	0	0	1	2	0	2	0	7
CLT	1	0	0	1	0	0	1	1	4
BL	0	0	1	1	1	0	2	1	6
G	2	0	1	0	3	0	1	0	7
Total	5	0	2	3	6	0	6	2	24

experience then there is cause to question using user experience as a criterion for measuring search engine performance. Figure 89 presents the experience scores as points (some overlap) for each of the incomplete tasks and labels each point with the search engine that was used for the task. We find the median of all the experience scores to be 3 which is a negative experience score, which suggests not completing a search task maps to a negative

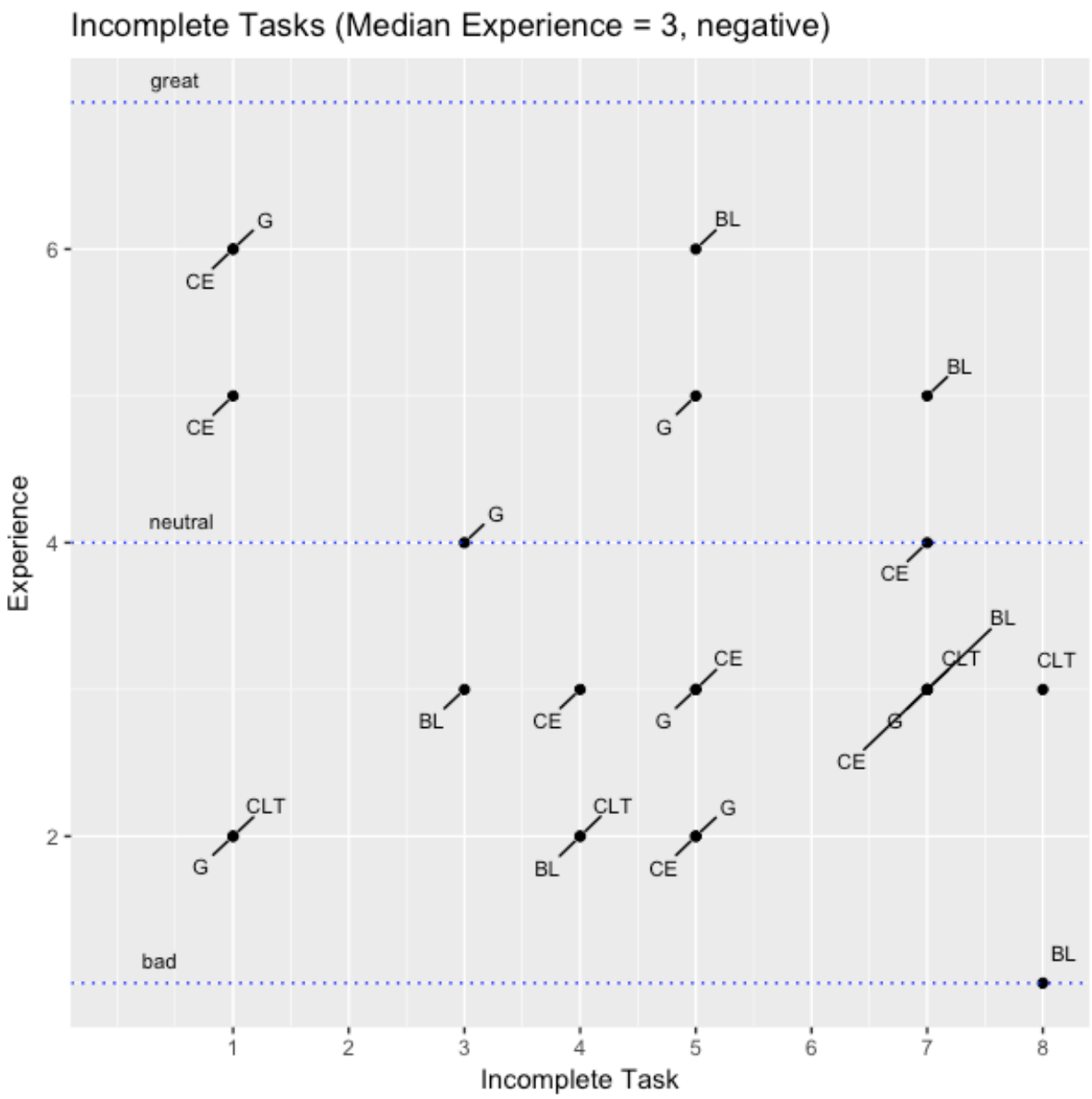


Figure 88. Experience by Incomplete Tasks.

search experience. We do find 6 out of 23 scores are positive. These scores come from the Find 4 tasks. In these instances, all the participants were able to find multiple snippets (one single snippet, three two snippets, and two three snippets) of code, but not all four snippets. One possible explanation for more positive experiences in these cases is that the participants may have had a positive experience finding the code they did find, which influenced their experience scores. Regardless, though, we find the central tendency is to have a negative experience when the task is not complete, which suggests negative user experience reflects unsuccessful searches, amongst other negative issues (e.g., taking too long to search in more focused tasks, as we saw in previous analyses).

6.2.4 Feature Usage

We next look at how the features of the search engines were used and some participant explanations on the usefulness of the features in order to "dive a level deeper" and explain some of the performance differences we saw above. We report what we found from the usage logs that were recorded during the experiments with CodeExchange, CodeLikeThis, Google, and the baseline.

6.2.4.1 CodeExchange Feature Usage

To understand what features of CodeExchange helped and may have caused the participants to search more (as we saw above), we looked at how often features were used, when feature usages were used to find code that was copied, when feature usage lead to positive/negative user experience, and looked at what the participants had to say during our interview. Table

Table 36. CodeExchange Search Behavior.

	T1	T2	T3	T4	T5	T6	T7	T8	T
R Frequency	9	2	10	2	4	7	9	3	46
C Frequency	3	0	1	1	4	1	4	2	16
LC Frequency	6	5	5	0	14	8	17	6	61
QP Frequency	13	18	25	14	41	19	36	10	176
Iterative Frequency	31	25	41	17	63	35	66	21	299
K Frequency	26	22	31	42	54	22	32	13	242
AS Frequency	1	1	23	0	7	1	2	1	36
H Frequency	3	6	0	0	0	0	2	2	13
Copies some time after only Keywords	10 (.38)	5 (.22)	16 (.51)	5 (.11)	17 (.31)	6 (.27)	8 (.25)	3 (.23)	70 (.28)
Copies some time after R/C/LC	14 (.77)	2 (.28)	15 (.93)	1 (.33)	8 (.36)	5 (.31)	15 (.5)	3 (.27)	63 (.51)
Copies some time after QP	3 (.23)	1 (.05)	6 (.24)	4 (.28)	7 (.17)	5 (.26)	11 (.30)	5 (.5)	42 (.23)
Copies some time after AS	0 (0)	0 (0)	4 (.17)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	4 (.1)

36 presents the usage frequency counts of features in CodeExchange by task. The key for Table 36 is **R** for recommendations, **C** for critiques, **LC** for language constructs, **QP** for query parts, **Iterative** for all iterative features, **K** for keywords, **AS** for advanced search, and **H** for history. We find that the features that support iteration were used most often (50.6%), followed by keyword text box (41.0%), advanced search (6.0%), and history (2.2%). Further, we found that the iterative features were used significantly more often than keywords ($p<0.1$, where $p=0.01$ with $\chi 2$ on a 2x1 contingency table on frequencies). This suggests that the iterative features of CodeExchange had a positive impact in completing search tasks and a reason for issuing more queries than the baseline.

To take another perspective on the iterative features' impact on completing a task, we looked at how many copies happened after only keywords versus after queries that include a recommendation, critique, or a language construct. We separate counting queries created using query parts from the other iterative features in this part of the analysis because counting query part usages would count copies after a deactivation even if the query leading to a copy is composed only of keywords, which we are trying to separate out in this analysis. Further, we look at the copy/keyword and copy/iterative-feature ratios (these ratios measure how many copies happen per keyword query and how many copies happen per query created with an iterative feature). The ratios appear in parenthesis after the copy counts in the table. We found that when the iterative features (R/C/LC) were used to refine a query, they lead to higher number of copies on average (.51) than when only keywords were used (.28). This data further suggests that the iterative features of CodeExchange played a substantial role in completing the search tasks.

Next we look at the relationship between using CodeExchange's iterative features and the participants' experience score. Figure 89 maps completed tasks to experience scores by the frequency iterative features were used. Each point represents a task completed by a participant and maps the frequency the participant used iterative features to complete the task to their experience score. Since there is sometimes overlap at points, we use size to show how many points occupy a location. For example, the one task that was completed with 18 usages of the iterative features and experience score of six is represented as a small circle. The four tasks with two usages and an experience score of four, on the other hand, are represented as a larger circle. We outline the task completions occurring in the positive,

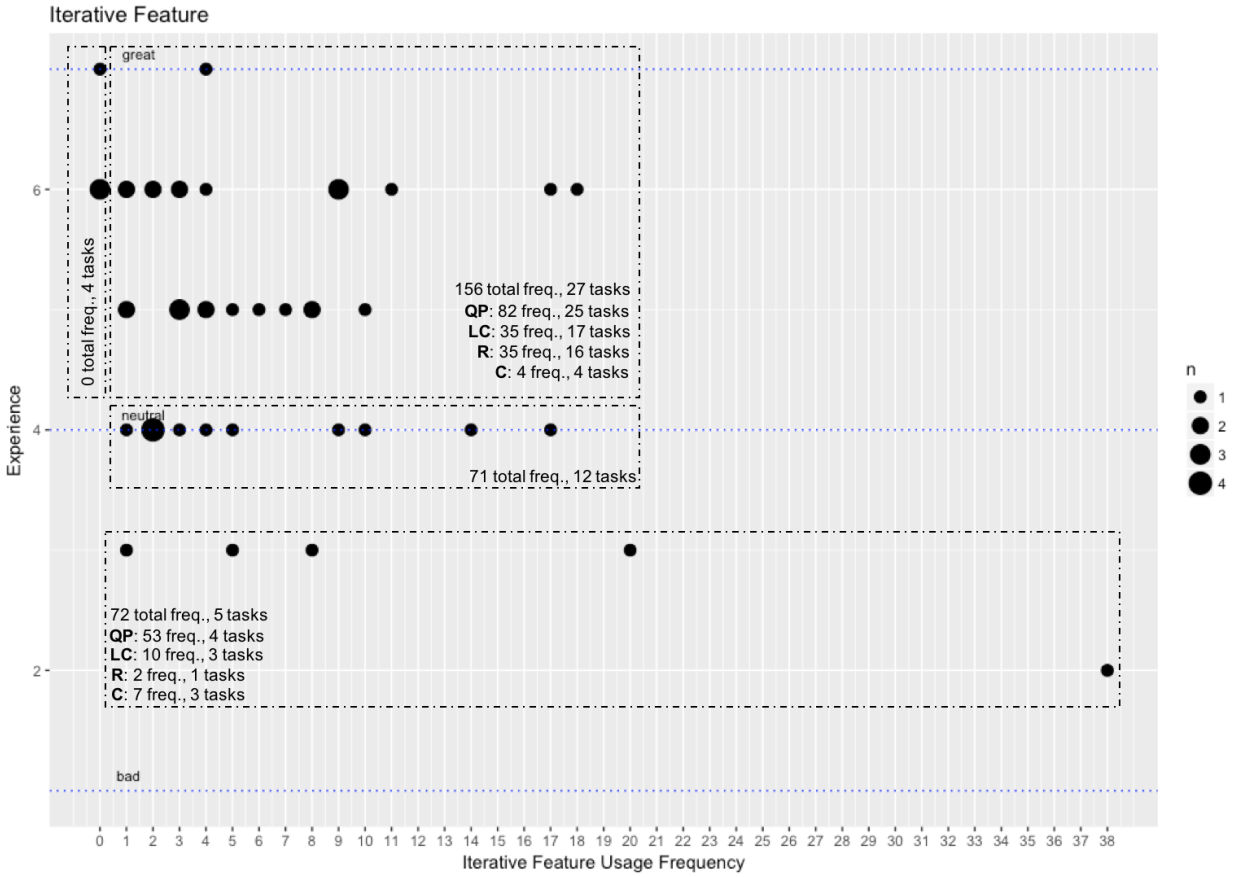


Figure 89. Mapping of Iterative Feature Usage Frequency to Experience.

neutral, and negative experience regions and annotate the outline with the total usage counts and total tasks completed. For example, we find that there are 12 tasks completed that received a neutral experience score, where the iterative features were used 71 times in total.

Our results show that in 27 tasks completed using an iterative feature one or more times (156 times in total), the participants reported a positive experience (the top outline). In 5 tasks completed that used an iterative feature one or more times (72 times total), the participants reported a negative experience in completing them (the bottom outline). We found more tasks completed with iterative features are associated with positive experiences than negative (27 to 5), and find more instances of using iterative features are associated

Table 37. Chi Squared Tests of Iterative Features and Experience.

Tasks Using Iterative Features	**Actual**			
	Postive	***Neutral***	***Negative***	***Total***
	27	12	5	44
	Expected			
	Postive	***Neutral***	***Negative***	***Total***
	14.66666667	14.66666667	14.66666667	44
			χ^2	
			p=	0.000181612
Usages of Iterative Features	**Actual**			
	Postive	***Neutral***	***Negative***	***Total***
	156	71	72	299
	Expected			
	Postive	***Neutral***	***Negative***	***Total***
	99.66666667	99.66666667	99.66666667	299
			χ^2	
			p=	4.2E-11

with positive experience scores than negative (156 to 72). Further, we find the differences in both cases significant (shown in Table 37). In four task completions, we find zero instances of using an iterative feature. These results suggest that the positive experience scores assigned to CodeExchange are associated with using the iterative features. This goes to explain why CodeExchange had a higher experience than the baseline. The high frequency of using iterative features, as shown above, explains why participants searched longer with CodeExchange (given the correlation we found between number of queries and time in Figure 87) and also explains the fact that they searched longer than the baseline and still had a higher experience. This means that searching more with the iterative features outweighed the spending of some extra time.

Thus far, we have seen results suggesting that CodeExchange's iterative features were used often, led to a higher rate of copies when used, and are used frequently and used in most tasks when the participants had a positive experience. Now we look at the impact of the frequency of using iterative features on the time to complete the task. Figure 90 plots time to complete a task to usage frequency of iterative features for each user (some points overlap). We find a moderate correlation between using iterative features and the impact it has on time to complete a task. This suggest that using iterative features do somewhat impact the time to complete a task, which we expected since they were used frequently and we found previously more queries is correlated to spending more time searching. We look next to see if this holds true by task type. Figure 91 shows the correlation analysis by task type, where the first row shows the correlations by broader tasks and the second row shows correlations by more focused tasks. For the Find 4, No Specific Role, and Algorithm/Data

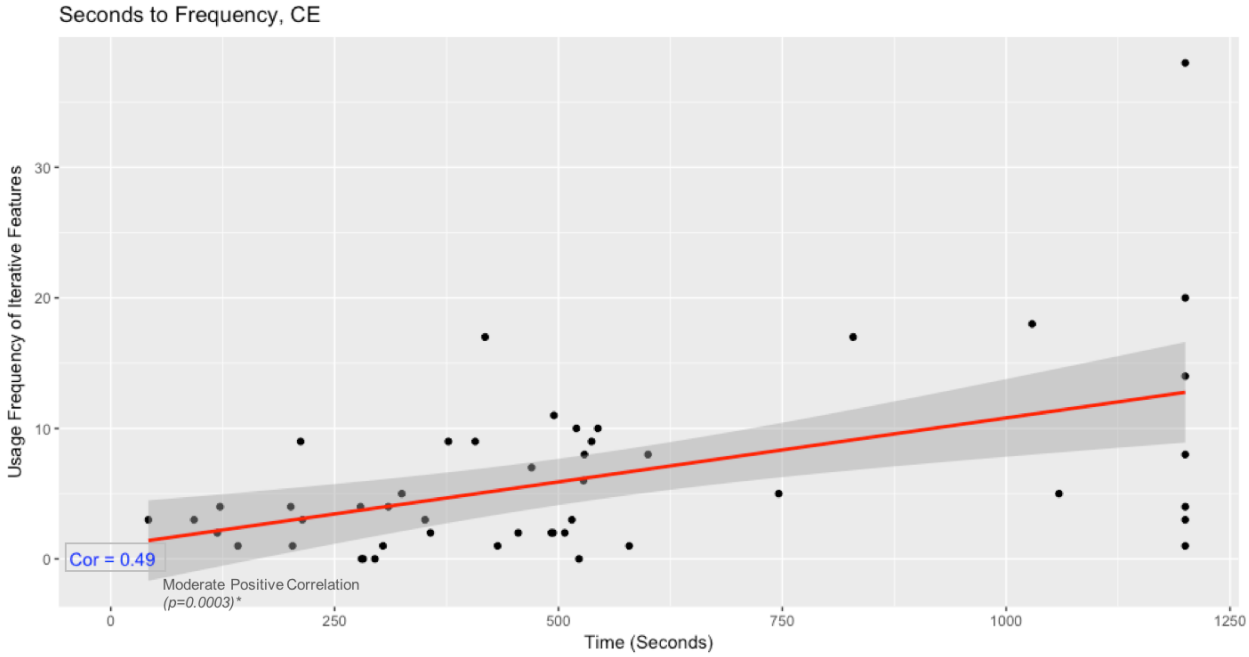


Figure 90. Uses of Iterative Features to Time Correlation Across All Tasks.

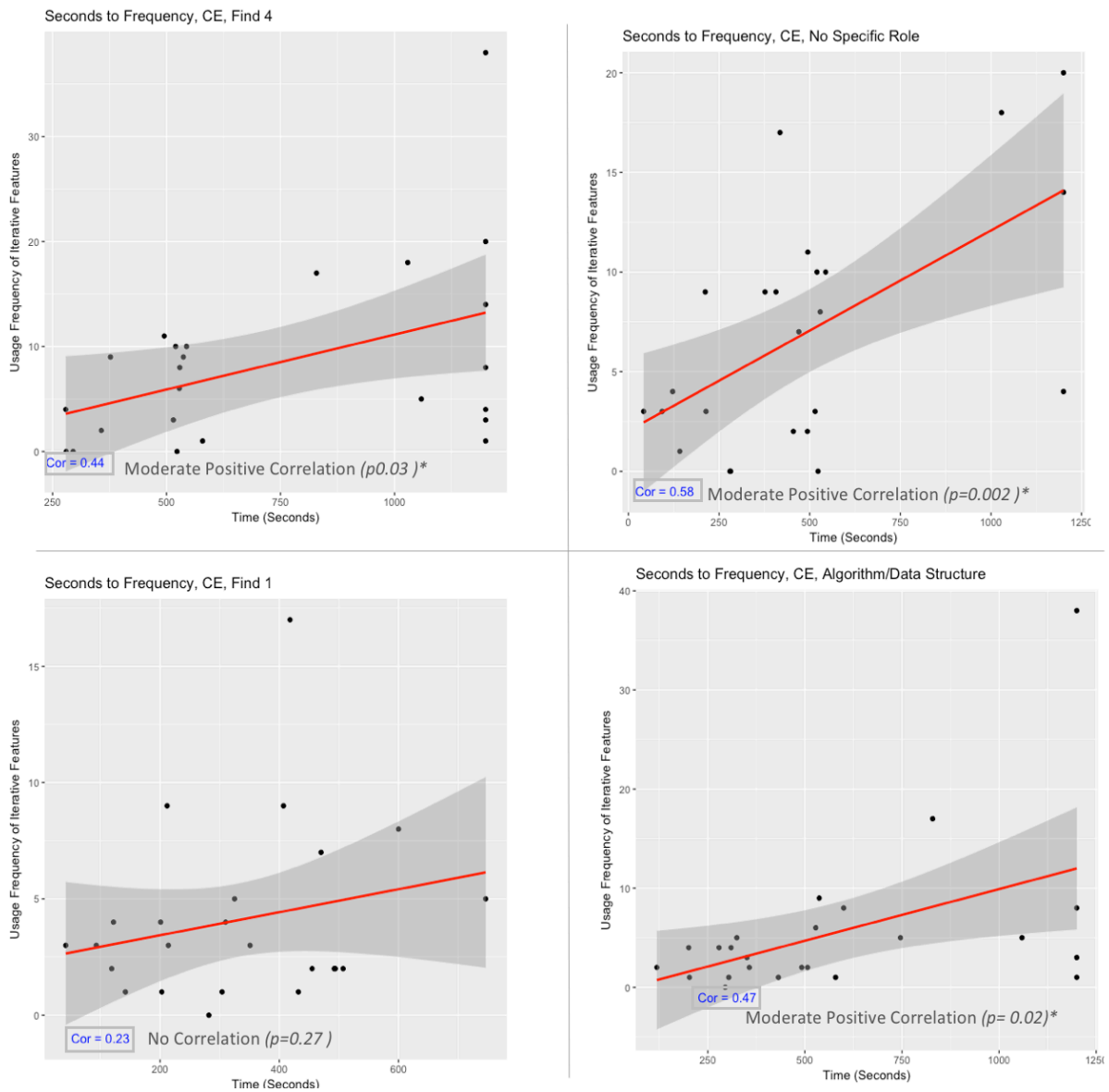


Figure 91. Uses of Iterative Features to Time Correlation by Task Type.

Structure tasks, we find significant positive correlations. While for Find 4 tasks, using more iterative features correlates with spending more time, which tends to decrease the experience (as shown in Figure 82), we also found the experience does not become negative until after 600 seconds (shown in Figure 82). Since, the majority of tasks completed with CodeExchange are finished in less than 600 seconds, CodeExchange tended to have a positive experiences for Find 4 tasks (also higher than other approaches). However, it does raise the

question, how much usage of iterative features is too much and too little? We answer that in our next analysis. Similarly, we find that, while increasing the usage of iterative features increases the time to complete Algorithm/Data Structure and No Specific Role tasks, which tends to decrease the experience (as shown in Figure 84 and Figure 86), we found the experience does not become negative until after around 500 seconds (as shown in Figure 84 and Figure 86). Since the majority of tasks completed in this task type are completed by CodeExchange in less than that time, CodeExchange tended to have a positive experience for No Specific Role and Algorithm/Data Structure tasks.

We now look at relationship between the average usage of iterative features and the participants' median experience. Figure 92 presents the relationship between the average number of iterative features used for a task and the median experience for that task. We found that the relationship is very close to a cubic curve using linear regression (adjusted R-squared value of .95 and a p value of 0.001 suggest a very close fit), and with no linear correlation using Pearson's test. The curve suggests that lower amounts of usage (close to 3) and higher amounts of usage (close to 11) are related to lower experience scores (lower but still positive), but moderate amounts of usage (around 5 to 7) are related to the highest experience scores. This relationship suggests that, while using iterative features are related to higher and positive experience scores, there might exist an ideal amount of usage to achieve the best search experience when searching with aspects/qualities of the results (as implemented in CodeExchange). We discuss more of the possible explanations in the Discussion chapter.

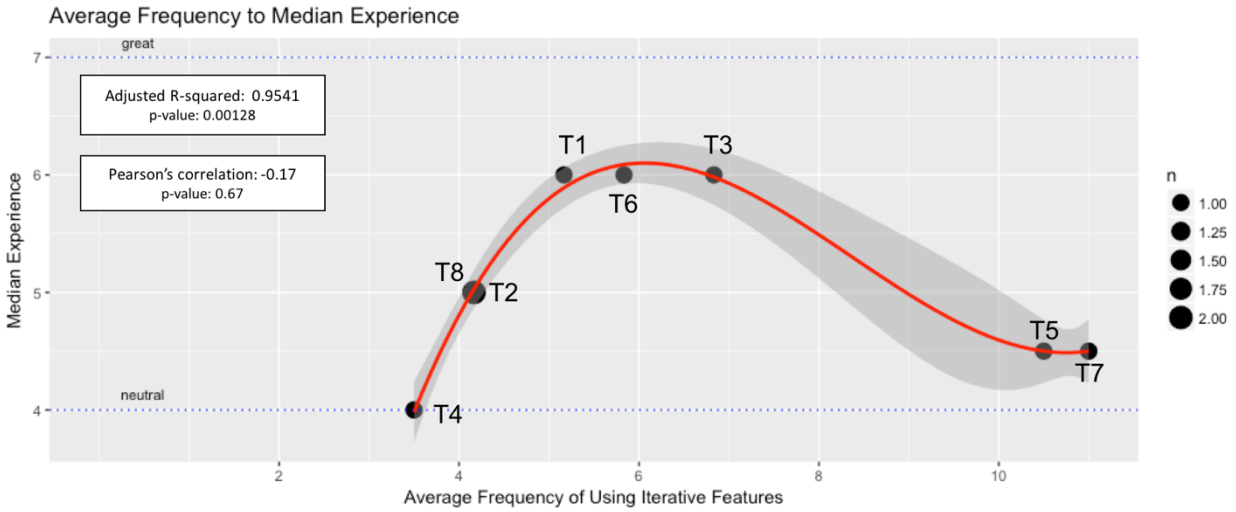


Figure 92. Average Number of Iterative Features Used to Median Experience for CodeExchange.

To get some idea of the kind of roles supported by CodeExchange's iterative features, we gathered what our participants had to say after their experiment as it related to CodeExchange's features. There was no structure to this interview, rather we just encouraged them to talk about the experiment and search engines they used. Our participants told us that, at a high level, *CodeExchange was better for drilling down…CodeExchange helped me go in a particular direction, where CodeLikeThis did not tell me.* They found language constructs useful, saying *I liked clicking the import…If I found a project that seemed to do what I needed... I could just click...and my search is in the project*. They felt query parts helped, saying *query parts helped explore*. They reported that refinement recommendations gave them ideas and helped them remember, saying *initially I typed in SMTP and got suggestion for a mail validator, then I added import* and *it guided me to think of ideas* and *I typed just "draw" in here, and it recommended AWT... I was like "Oh yeah,*

it was AWT". Of course, not all that was said during the interview was positive about CodeExchange, which we discuss in the interviews about the other search engines.

6.2.4.2 CodeLikeThis Feature Usage

Feature usage of CodeLikeThis was examined by looking at how often features were used and when using a feature was used to copy code. This data was derived from the logs for CodeLikeThis, which recorded like-this queries issued, keywords issued, back/forward buttons pressed, and copies that came after keywords or after a like-this query. Table 38 shows the usage and copy analysis by task.

We found that keywords were used twice as much as the like-this queries ($p<.1$, where $p=.0000001$ with $\chi 2$ on a 2x1 contingency table), which suggests that keywords played more of a role during search than the like-this queries. However, we looked at how many copies happened after keywords versus after like-this queries and the copy/keyword and copy/like-this ratios. These ratios appear in parenthesis after the copy counts. We found that keywords and like-this queries lead to an equal number of copies on average when they were used (.42). This suggests that keywords and like-this queries both may have an equally important impact in searching for code.

Table 38. CodeLikeThis Feature Usage.

	T1	T2	T3	T4	T5	T6	T7	T8	T
Back Frequency	25	6	12	5	11	1	21	12	93
Forward Frequency	0	0	2	0	0	0	0	0	2
More Frequency	14	5	15	4	14	4	10	8	74
Somewhat Frequency	5	1	3	3	3	1	12	2	30
Less Frequency	1	0	2	1	2	0	3	0	9
Iterative Frequency	20	6	19	8	19	5	25	10	113
Keywords Frequency	33	20	29	22	40	10	36	18	208
Copies after Keywords	17 (.51)	11 (.55)	15 (0.5)	3 (.13)	13 (.32)	6 (0.6)	15 (.41)	8 (.44)	88 (.42)
Copies after a Like-This	5 (.25)	2 (.33)	12 (0.6)	3 (.37)	12 (.63)	3 (0.6)	9 (.36)	2 (0.2)	48 (.42)

Next we look at the relationship between using CodeLikeThis' iterative features and the participants' experience score. Figure 93 maps completed tasks to experience scores by the number of times iterative features were used. Each point represents a task completed by a participant and maps the frequency the participant used iterative features to complete the task to their experience score. Since there is sometimes overlap at points, we use size to show how many points occupy a location. For example, there is one task completed where iterative features were used seven times to complete it and received an experience score of five; it is shown as a small circle. As another example, seven tasks involved one usage of an iterative feature and received an experience score of six; these are shown with a larger circle.

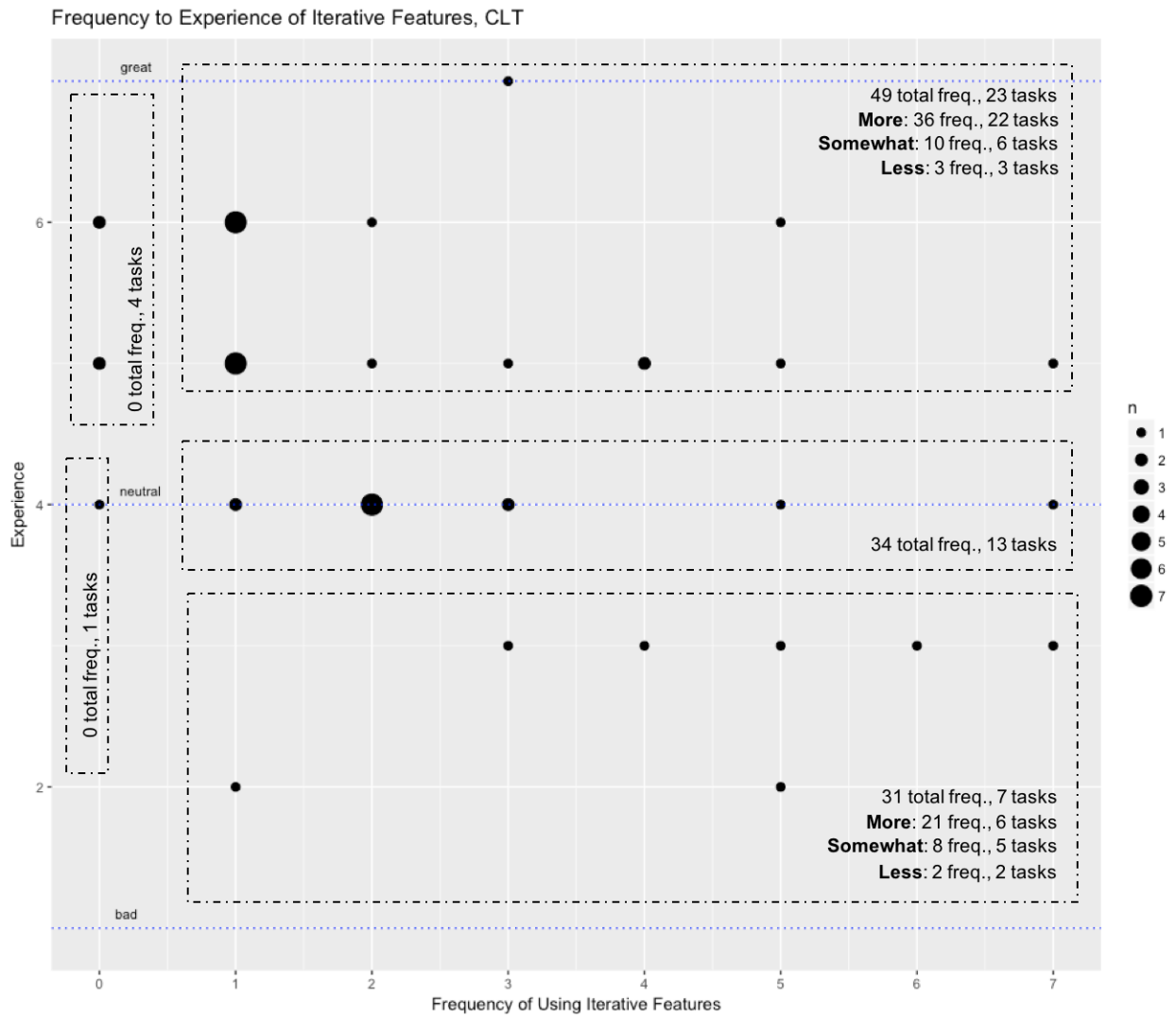


Figure 93. Mapping of Iterative Feature Usage to Experience.

We outline the tasks completed occurring in the positive, neutral, and negative experience regions and annotate the outline with the total usage counts and total tasks completed. For example, we find that there are 13 tasks completed that received a neutral experience score, where the iterative features were used 31 times in total.

Our results show that, in 23 tasks completed using an iterative feature one or more times (49 times in total), the participants reported a positive experience. In 7 tasks completed that used an iterative feature one or more times (31 times total), the participants reported a

negative experience in completing them. We find more tasks completed with iterative features are associated with positive experiences than negative (23 to 7), and find more instances of using iterative features are associated with positive experience scores than negative (49 to 31). Further, we find the differences in both cases significant (shown in Table 39). In five task completions, we find zero instances of using an iterative feature. These results suggest that the positive experience scores assigned to CodeLikeThis are associated with using the iterative features.

Table 39. Iterative Feature Usage by Experience Contingency Tables.

Tasks Using Iterative Features in CLT	**Actual**			
	Postive	***Neutral***	***Negative***	***Total***
	23	13	7	43
	Expected			
	Postive	***Neutral***	***Negative***	***Total***
	14.3333333	14.3333333	14.3333333	43
			χ^2 p=	0.01048154
Usages of Iterative Features in CLT	**Actual**			
	Postive	***Neutral***	***Negative***	***Total***
	49	34	31	114
	Expected			
	Postive	***Neutral***	***Negative***	***Total***
	38	38	38	114
			χ^2 p=	0.08652097

Thus far, we have seen results suggesting that CodeLikeThis iterative features lead to an equal rate of copies compared to keywords, are used frequently, and are used in most tasks when the participants had a positive experience. Now we look at what impact using iterative features has on the time to complete the task. Figure 94 plots time to complete a task to usage frequency of iterative features for each user (some points overlap). We find a weak correlation between using iterative features and the impact it has on time to complete a task. This suggests that using CodeLikeThis' iterative features may not impact the time to complete a task much. We look next to see if this holds true by task type. Figure 95 shows

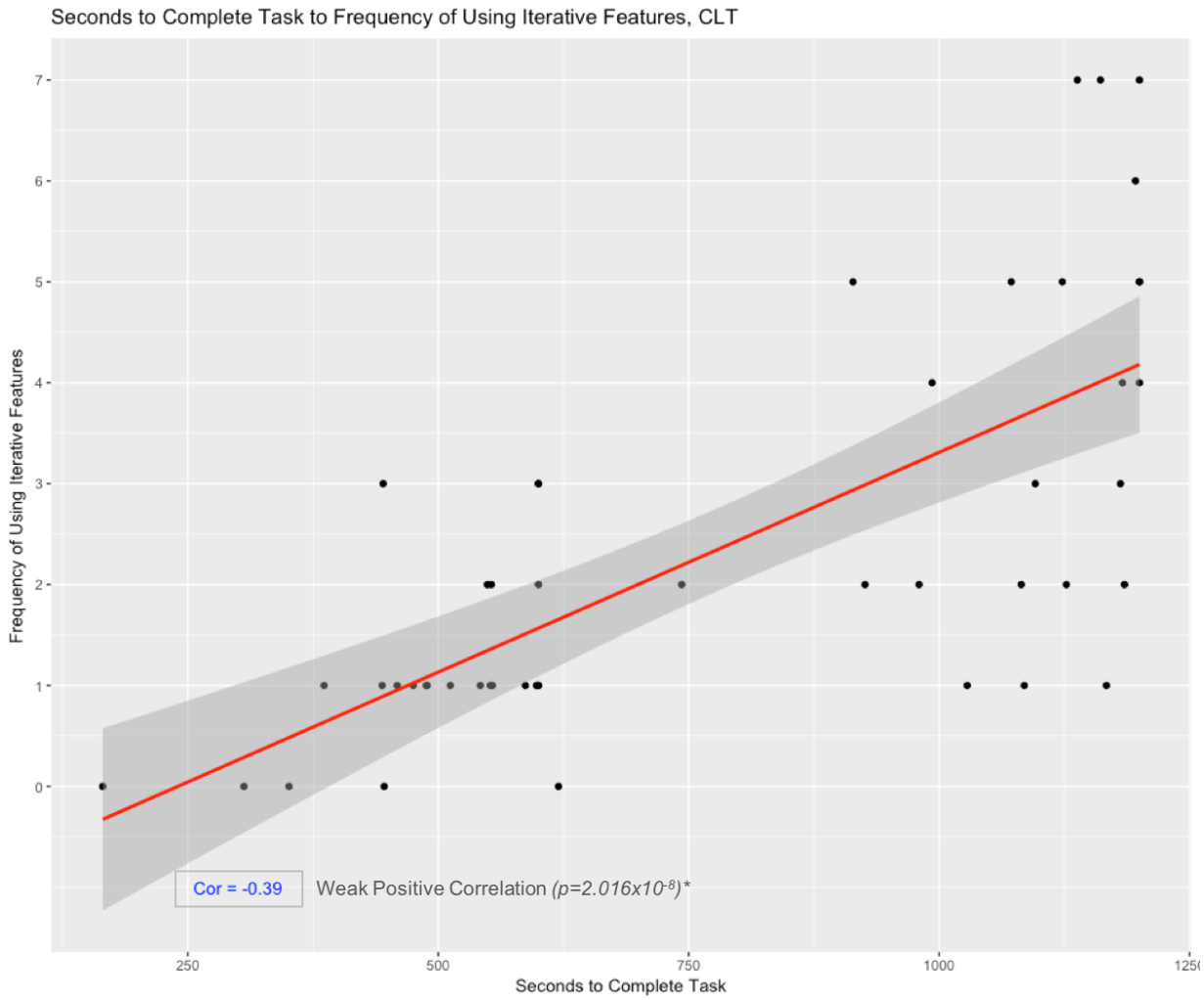


Figure 94. Time to Uses of Iterative Features Correlation Across All Tasks.

the correlation analysis by task type, where the first row shows the correlations by broader tasks and the second row shows correlations by more focused tasks. We find only two significant correlations. For the Find 4 and the No Specific Role tasks we find significant positive correlations, while for the other tasks we find no correlations. However, as shown in Figure 82, Figure 83, and Figure 84, we found that time is not correlated with experience for the border tasks (Find 4 and No Specific Role) and so taking more time to use iterative

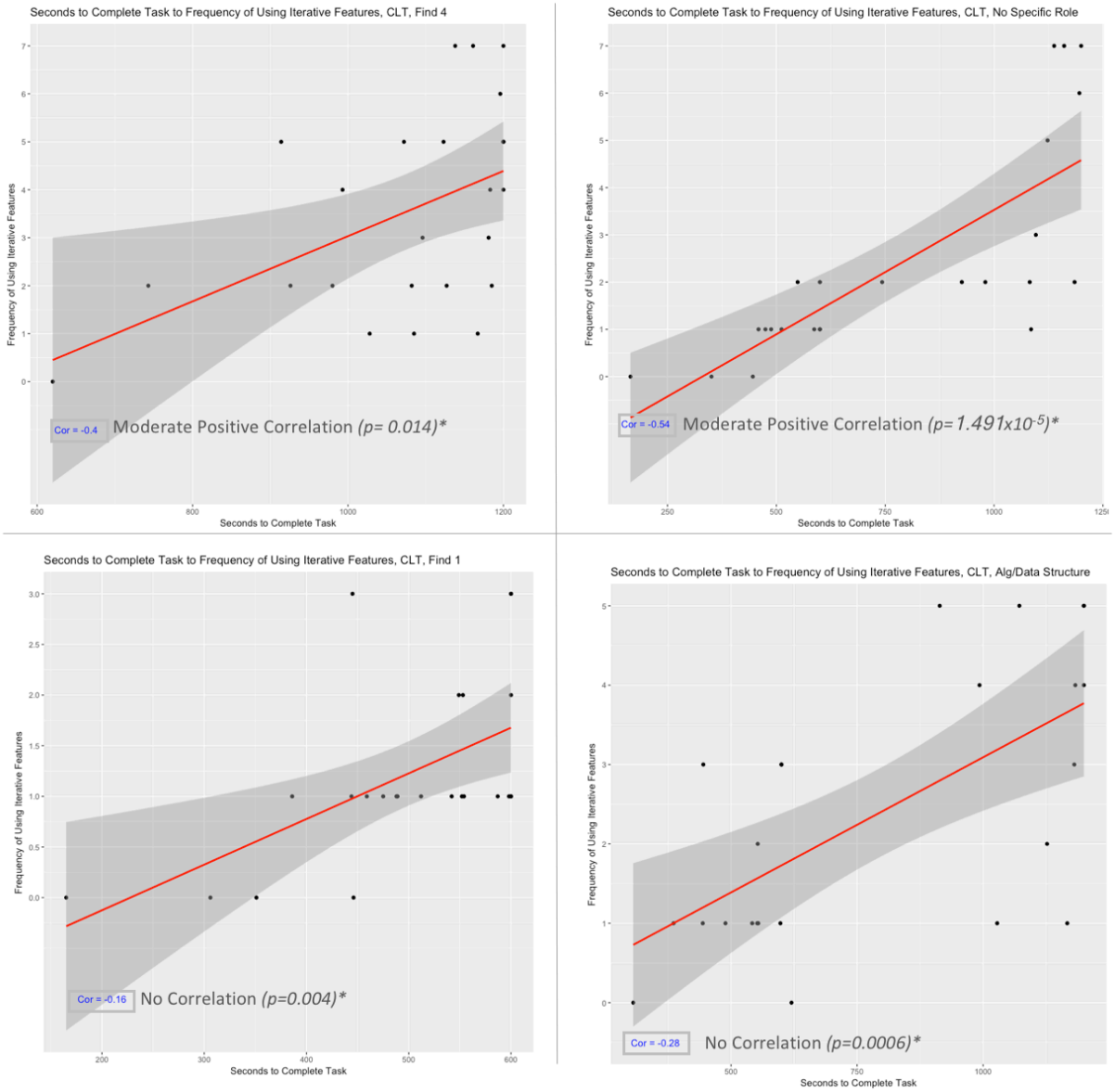


Figure 95. Uses of Iterative Features to Time Correlation by Task Type.

features may not impact experience much. However, it does raise the question, how much usage of like-this queries is too much and too little? We answer that in our next analysis.

Figure 96 presents the relationship between the average number of iterative features used for a task and the median experience for that task. We found that the relationship is best fitted by a linear curve, yet weakly fitted (adjusted R-squared value of .46 and a p value of 0.03 suggest a weak fit), but find a strong negative linear correlation with Pearson's test. The curve suggests that lower amounts of usage (close to 1 or 2) are related to the highest experience scores and higher amounts (close to 4) are related to relatively lower experience scores. This relationship suggests that, while using iterative features are related to higher and positive experience scores, that like-this queries might be best used 1 or 2 times to achieve higher experience scores. This contrasts with CodeExchange, where we found that

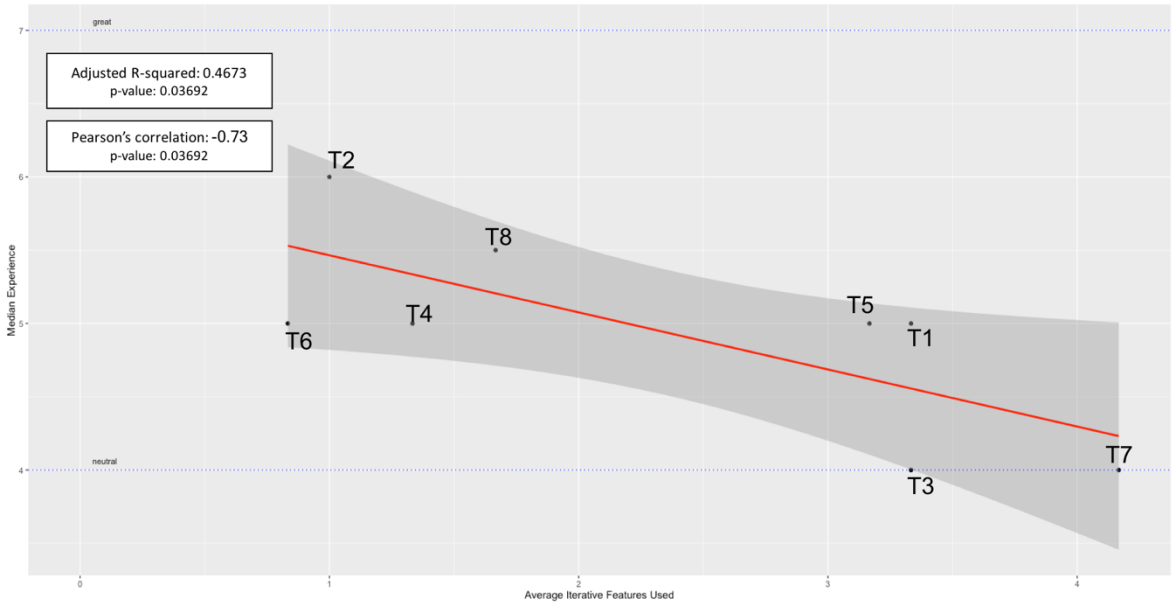


Figure 96. Average Iterative Features Used to Median Experience for CodeLikeThis.

the highest experience scores are associated to more moderate levels of usage of the iterative features (around 5 to 7 times). We discuss possible explanations in the next chapter.

At a high level, participants explained like-this queries helped them get a new perspective. They said with CodeLikeThis *you get to see different codes and different ideas*…and *it gave me a new perspective*… and *it was sort of a way of exploring*. They explained the like-this queries were *helpful because I wasn't sure exactly what I was looking for*… and that *it works well for queries that are common and precise: quicksort, hash table, game loop or game examples for instance.* However, it is clear that like-this queries do not always work, one participant explained *more like this, is pretty much what you expect, but the other two doesn't really follow the semantics in my mind* and *another saying I felt a little lost.*

6.2.4.3 Google Feature Usage

For Google, we recorded keyword queries, advanced keyword queries (e.g., restricting search to a site with the "site" qualification or using Google's advanced search form), and domains visited by clicking hyperlinks. Our first analysis is presented in Table 40 where we compared the frequency in which participants used keywords versus advanced queries by task. We found that keyword queries were used about five times as much advanced queries (84.7% to 15.2%) with $p<0.1$ ($p=1.4x10^{-28}$) with $\chi2$ on a 2x1 contingency table. This suggests either keywords sufficed or advanced search was less helpful or harder to use for the participants.

To get an idea of the impact of Google's index of web pages on code search, we next looked at the number of domains that were visited by clicking a hyperlink and how often domains were visited by clicking a hyperlink across participants (we do not double count a visit when one participant clicks the same link multiple times). We present the results of our analysis in Table 41 and Table 42 .In these tables, all domains visited occur under the column called "Domain" and are listed in descending order by the total number of times they were visited by participants, which is listed under the "Total" column. The frequency of visits by task number are labeled under the columns 1 though 8.

We found that there were 126 different domains visited 529 times, but the frequency at which they are visited followed a long tail distribution with github.com (131/529 visits) and stackoverflow.com (105/529 visits) visited most often and then a sharp drop in number of visits to other domains afterward. However, while individually the other 124 domains are visited much

Table 40. Google Keywords and Advanced Query Usage.

	Keywords	**Advanced**	**Total**
Task 1	33	5	38
Task 2	16	3	19
Task 3	27	7	34
Task 4	24	10	34
Task 5	59	9	68
Task 6	13	4	17
Task 7	34	1	35
Task 8	10	0	10
Total	216	39	255

Table 41. Domains Visited by Task - Part 1.

	Task										Task								
Domain	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*Total*	**Domain**	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*Total*
github.com	21	5	1	0	34	14	56	0	131	math.nist.gov	0	0	0	0	2	0	0	0	2
stackoverflow.com	5	8	17	26	28	3	14	4	105	planet-source-code.com	1	1	0	0	0	0	0	0	2
docs.oracle.com	1	0	16	0	1	3	4	0	25	processingjs.nihongoresources.com	2	0	0	0	0	0	0	0	2
java.net	0	0	0	0	19	0	0	0	19	programmers.stackexchange.com	0	0	0	0	2	0	0	0	2
commons.apache.org	0	0	0	0	9	0	0	0	9	raw.githubusercontent.com	0	0	0	0	1	1	0	0	2
codereview.stackexchange.com	1	0	0	3	1	0	3	0	8	statr.me	0	0	0	0	2	0	0	0	2
jcaif.sourceforge.net	0	0	0	0	0	0	8	0	8	abeautifulsite.net	1	0	0	0	0	0	0	0	1
jdesurvey.codeplex.com	0	0	0	0	0	0	8	0	8	android.googlesource.com	0	0	1	0	0	0	0	0	1
math.hws.edu	1	0	3	0	0	0	0	3	7	baeldung.com	0	0	0	1	0	0	0	0	1
tutorialspoint.com	0	0	1	0	0	2	2	2	7	bluej.org	0	0	1	0	0	0	0	0	1
alvinalexander.com	1	1	0	0	0	1	2	1	6	careerbless.com	0	0	0	0	1	0	0	0	1
introcs.cs.princeton.edu	0	0	0	1	5	0	0	0	6	code.arc.cmu.edu	0	0	1	0	0	0	0	0	1
coderanch.com	0	0	0	0	1	0	4	0	5	code.google.com	0	0	0	0	1	0	0	0	1
dreamincode.net	2	0	1	0	1	0	0	1	5	code.org	0	0	0	0	1	0	0	0	1
java2s.com	0	3	2	0	0	0	0	0	5	codereview.com	0	0	0	0	0	0	1	0	1
sanfoundry.com	0	0	0	1	4	0	0	0	5	codereview.stackexchagne.com	0	0	0	1	0	0	0	0	1
services.brics.dk	0	0	0	0	5	0	0	0	5	cs.cmu.edu	0	0	0	0	1	0	0	0	1
google.com	1	0	1	1	0	1	0	0	4	cs.elte.hu	0	0	0	0	1	0	0	0	1
quora.com	2	0	0	0	0	1	0	1	4	cs.ucf.edu	0	0	0	0	0	0	0	1	1
sourceforge.net	1	0	0	0	0	0	3	0	4	cs.umd.edu	0	0	1	0	0	0	0	0	1
zetcode.com	4	0	0	0	0	0	0	0	4	csee.umbc.edu	0	0	0	0	1	0	0	0	1
avc.com	0	0	0	0	3	0	0	0	3	dba.stackexchange.com	0	0	0	0	0	0	1	0	1
beginwithjava.blogspot.com	1	0	2	0	0	0	0	0	3	developers.google.com	0	1	0	0	0	0	0	0	1
codejava.net	0	0	0	0	0	2	1	0	3	developers.slashdot.org	1	0	0	0	0	0	0	0	1
cs.lmu.edu	0	0	3	0	0	0	0	0	3	docjar.com	0	0	0	1	0	0	0	0	1
gamedev.stackexchange.com	2	0	0	0	1	0	0	0	3	eclipse.org	0	0	0	0	0	0	1	0	1
java-gaming.org	3	0	0	0	0	0	0	0	3	example-code.com	0	1	0	0	0	0	0	0	1
javaprogrammingforums.com	1	0	0	0	0	1	1	0	3	forum.codecall.net	1	0	0	0	0	0	0	0	1
neiljohan.com	3	0	0	0	0	0	0	0	3	freecode.com	0	0	0	0	1	0	0	0	1
oracle.com	0	1	2	0	0	0	0	0	3	functionx.com	0	0	0	0	1	0	0	0	1
singularsys.com	0	0	0	0	3	0	0	0	3	gamedevelopment.tutsplus.com	1	0	0	0	0	0	0	0	1
bootstrapworld.org	0	0	0	0	2	0	0	0	2	ibm.com	0	0	0	0	0	0	1	0	1
codeproject.com	1	0	0	0	0	0	0	1	2	imhasib.wordpress.com	1	0	0	0	0	0	0	0	1
coderance.com	1	0	0	0	0	0	1	0	2	inmath.com	0	0	0	0	1	0	0	0	1
developer.com	0	2	0	0	0	0	0	0	2	instructables.com	1	0	0	0	0	0	0	0	1
dynamicgeometry.com	0	0	2	0	0	0	0	0	2	intelligentjava.wordpress.com	0	0	0	1	0	0	0	0	1
dzone.com	0	0	0	0	0	0	2	0	2	intmath.com	0	0	0	0	1	0	0	0	1
en.wikipedia.org	0	0	0	0	2	0	0	0	2	java-source.net	0	1	0	0	0	0	0	0	1
examples.javacodegeeks.com	0	0	1	0	0	0	1	0	2	javacooperation.gmxhome.de	1	0	0	0	0	0	0	0	1
freesourcecode.net	0	2	0	0	0	0	0	0	2	javaguicodexample.com	0	0	0	0	0	0	1	0	1
gamedev.net	2	0	0	0	0	0	0	0	2	javamarioplatformer.codeplex.com	1	0	0	0	0	0	0	0	1
gist.github.com	0	0	0	0	1	0	1	0	2	jdsoft.com	0	0	0	0	0	0	1	0	1
ihsn.org	0	0	0	0	0	0	2	0	2	jspsurveylib.sourceforge.net	0	0	0	0	0	0	1	0	1
java-forums.org	0	0	0	0	0	0	1	1	2	kaviddiss.com	0	0	0	0	0	0	1	0	1
java-tips.org	0	2	0	0	0	0	0	0	2	la4j.org	0	0	0	0	1	0	0	0	1
javagraphics.java.net	0	0	0	0	2	0	0	0	2	martin-thoma.com	1	0	0	0	0	0	0	0	1
javamail.java.net	0	2	0	0	0	0	0	0	2	mathandcoding.org	0	0	0	0	1	0	0	0	1
javaworld.com	1	0	0	0	0	0	1	0	2	mathematik.uni-kl.de	0	0	0	0	1	0	0	0	1
journaldev.com	0	0	0	0	0	0	1	1	2	mkyong.com	0	0	0	0	0	1	0	0	1
krum.rz.uni-mannheim.de	0	0	0	0	2	0	0	0	2	msdn.microsoft.com	0	0	0	0	0	1	0	0	1

Table 42. Domains Visited by Task - Part 2.

	Task								
Domain	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*Total*
mychess.com	1	0	0	0	0	0	0	0	1
ncbi.nim.nih.gov	0	0	0	0	0	1	0	0	1
netbeans.org	0	0	0	0	0	1	0	0	1
news.ycombinator.com	0	0	0	0	1	0	0	0	1
ntu.edu.sg	0	0	1	0	0	0	0	0	1
playsudoku.sourceforge.net	1	0	0	0	0	0	0	0	1
popkade.ir	0	0	0	0	0	0	1	0	1
programcreek.com	0	1	0	0	0	0	0	0	1
programmingsimplified.com	0	0	0	0	1	0	0	0	1
progressivejava.net	1	0	0	0	0	0	0	0	1
projects.congrace.de	0	0	0	0	1	0	0	0	1
raywenderlich.com	1	0	0	0	0	0	0	0	1
scicomp.stackexchange.com	0	0	0	0	1	0	0	0	1
smallbusiness.chron.com	0	0	0	0	0	1	0	0	1
sourcecodesworld.com	0	0	1	0	0	0	0	0	1
suberic.net	0	1	0	0	0	0	0	0	1
surjey.cvs.sourceforge.net	0	0	0	0	0	0	1	0	1
survey.codeplex.com	0	0	0	0	0	0	1	0	1
surveymonkey.com	0	0	0	0	0	0	1	0	1
surveysystem.com	0	0	0	0	0	0	1	0	1
techoverflow.net	1	0	0	0	0	0	0	0	1
twili.com	0	0	0	0	0	0	1	0	1
twillio.com	0	0	0	0	0	0	1	0	1
vertabelo.com	0	0	0	0	0	0	1	0	1
wphooper.com	0	0	1	0	0	0	0	0	1

less compared to github.com or stackoverflow.com, collectively they are visited (293/529) more frequently than github.com and stackoverflow.com. This suggests that the size and variety of Google's index was helpful for participants in their search.

Table 43. Google Search Behavior.

	T1	T2	T3	T4	T5	T6	T7	T8
Queries per User	6.33	3	5.83	5.67	11.3	3	5.83	1.67
Terms per Query	5.92	5.52	4.23	5.24	4.07	5.93	4.8	4.5
Terms Deleted per Query	2.43	1	1.47	1.61	1.42	1.57	2.03	0.67
Terms Added per Query	2.57	1.05	1.53	1.88	1.45	1.79	1.94	1

We looked at the query behavior of search with Google and present the results in Table 43. Each row presents on average how many queries were issued per user, the average number of terms in the keyword queries, and the average number of terms deleted and added when modifying a query. The results support the hypothesis that code search on the Internet, even with a large-scale web search engine, is iterative and can have a vast range of queries (1.67 to 11.3). These results suggest Google could benefit from features that explicitly support iterative code search. Further, analyzing queries by task type supports the same conclusion that Google could benefit from features that explicitly support iterative code search, where the typical range of queries span from 3.33 to 7.33 as shown in Table 44.

Interestingly, an even deeper analysis suggests that Google could especially benefit from features that explicitly support using the results from the previous query in creating the next query. Out of 198 queries, 105 (53%) queries contained words occurring in the text on web pages visited from the previous query in the same task. Among the 198 queries considered,

Table 44. Google Search Behavior by Task Type.

	Find 4	Find 1	No Specific Role	Algorithm/Data Structure
Queries per User	7.33	3.33	4.42	6.25
Terms per Query	4.65	5.35	4.92	4.83
Terms Deleted per Query	1.73	1.26	1.52	1.63
Terms Added per Query	1.76	1.46	1.55	1.75

we do not include first queries for a task because no previous websites can be visited. When counting words, we exclude stop words that are not in the Java language and words occurring in the first query that came directly from the participant and not a web page. These results suggest the participants were "borrowing" content from a web page in creating their next keywords. Further, we counted words from many web pages. Specifically, we counted words from StackOverflow 27% of the time, GitHub 14% of the time, and other pages that included tutorial and documentation pages 47% of the time (note, if a word in a query occurred in more than one web page visited in the previous results, they both counted as the source). We note that GitHub might have a somewhat lower percentage, because some of the visits appear to be only for navigational purposes (e.g., walking a file hierarchy) only.

We find that the number of terms removed from a query is about equal to the number that is added. Further, the number of terms added, deleted, and added and deleted are on average less than the number of terms per query. This suggests that the participants often kept terms from the previous query in their next query, making their query modifications more as incremental adjustments. This suggests support for incrementally modifying a query to iteratively search needs to be supported.

While users iteratively searched with Google using keyword queries, we wanted to see if the number of keyword queries issued correlated with the user's experience. Figure 97 presents the relationship between the average number of keyword queries used for a task and the median experience for that task. We found that the relationship is best fitted by a linear curve, yet moderately fitted (adjusted R-squared value of .63 and a p value of 0.01 suggest a

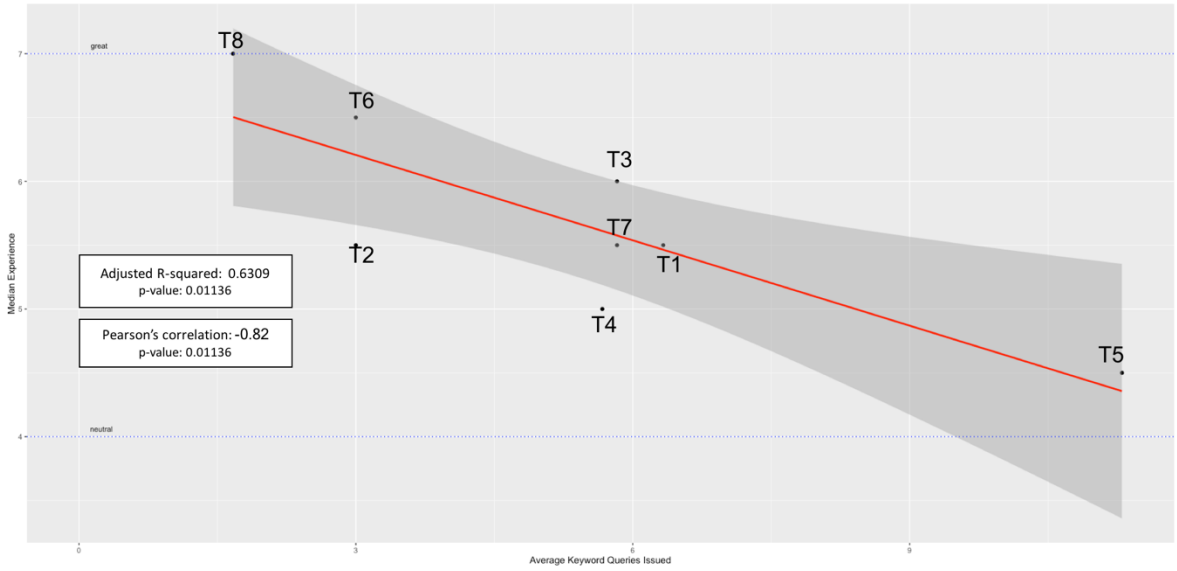


Figure 97. Average Iterative Features Used to Median Experience for Google.

moderate fit), but we find a strong negative linear correlation with Pearson's test. The curve suggests that lower amounts of keyword query usage (close to 2 or 3) are related to the highest experience scores and higher amounts (closer to 11) are related to, relatively, lower experience scores. This relationship suggests that there seems to be a relationship between higher numbers of iterations with keywords and a, relatively, lower user experience.

Participants told us they appreciated Google for the context and comments they would find on web pages. Some said *sometimes I was like, oh man I wish I could use Google at this point, just to get some context so I can understand what I need to search in CodeExchange*… and *I use Google to find best practices* and *Google had comments*. However, there are times when Google did not help. Our participants told us sometimes they asked *Why am I getting this?* and *Google is lacking in digging down*… and *for simple straight forward [questions] I felt CodeLikeThis was better*.

6.2.4.4 Baseline Feature Usage

While the baseline is limited in features, we were able to log some statistics about the queries issued using the baseline search engine and present the results in Table 45. Each row presents on average how many queries were issued per user, the average number of terms in the keyword queries, and the average number of terms deleted and added when modifying a query. The results are similar to what we see with Google, but with important differences. Similar to Google, we find that code search on the Internet is iterative with a vast range of number of queries on average (3.17 to 9). Also similar to Google, Table 46 shows that the average queries by task type is somewhere between 4 and 7. However, unlike Google, the baseline had fewer terms per query and fewer terms added/deleted. This difference cannot easily be explained. It could be that Google searches require more terms to effectively search with them, Google's autocomplete recommends more useful keywords, or web pages give the participants more ideas for keywords, to just name a few.

Figure 98 presents the relationship between the average number of keyword queries used for a task and the median experience for that task. We found that the relationship is best fitted by a polynomial to the 4th degree, yet moderately fitted (adjusted R-squared value of

Table 45. Baseline Search Behavior.

	T1	T2	T3	T4	T5	T6	T7	T8
Queries per User	7.17	3.83	4.67	3.5	9	3.17	7.17	6
Terms per Query	2.05	2.7	1.39	2.57	1.37	2.95	1.44	2.22
Terms Deleted per Query	1.05	0.86	0.89	1.4	1.02	0.78	0.9	1
Terms Added per Query	1.05	0.82	1	1.2	1.04	1.17	0.98	1.17

Table 46. Baseline Search Behavior by Task Type.

	Find 4	Find 1	No Specific Role	Algorithm/Data Structure
Queries per User	7.00	4.13	4.71	6.42
Terms per Query	1.57	2.55	1.94	1.92
Terms Deleted per Query	0.95	0.97	0.84	1.05
Terms Added per Query	0.99	1.05	0.95	1.06

.81 and a p value of 0.051 suggest a moderate fit), and we find a no correlation with Pearson's test. From the graph and correlation test, we find no clear pattern of number of keyword queries issued with the baseline on experience. That is, sometimes more queries are better and sometimes it is worse. This contrasts with what we found in Google, where fewer queries lead to better experiences than more. A possible explanation is that sometimes seeing more results in the baseline lead to an improved experience, whereas seeing more results in Google did not. However, an improved experience for the baseline might still be, categorically speaking, negative or neutral.

6.2.5 Why Code Was Chosen for Tasks

Now we turn our attention to why code was selected for the tasks. We are particularly interested in the motivations behind selecting code and how they might explain, in part, the iterative search behavior observed in our experiments. To understand why code was chosen, we examined the 463 explanations given by our participants that reported why they chose code (one explanation for each snippet they found) and used them to create categories of reasons for selecting code. To create these categories, three graduate students (the author and two other software engineering graduate students) participated in an affinity

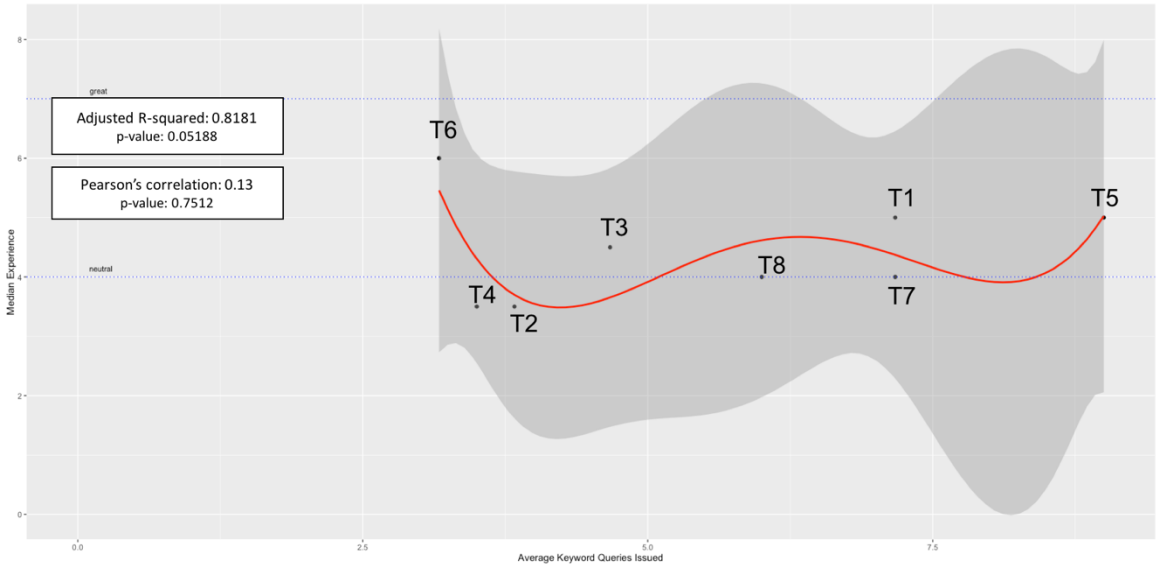


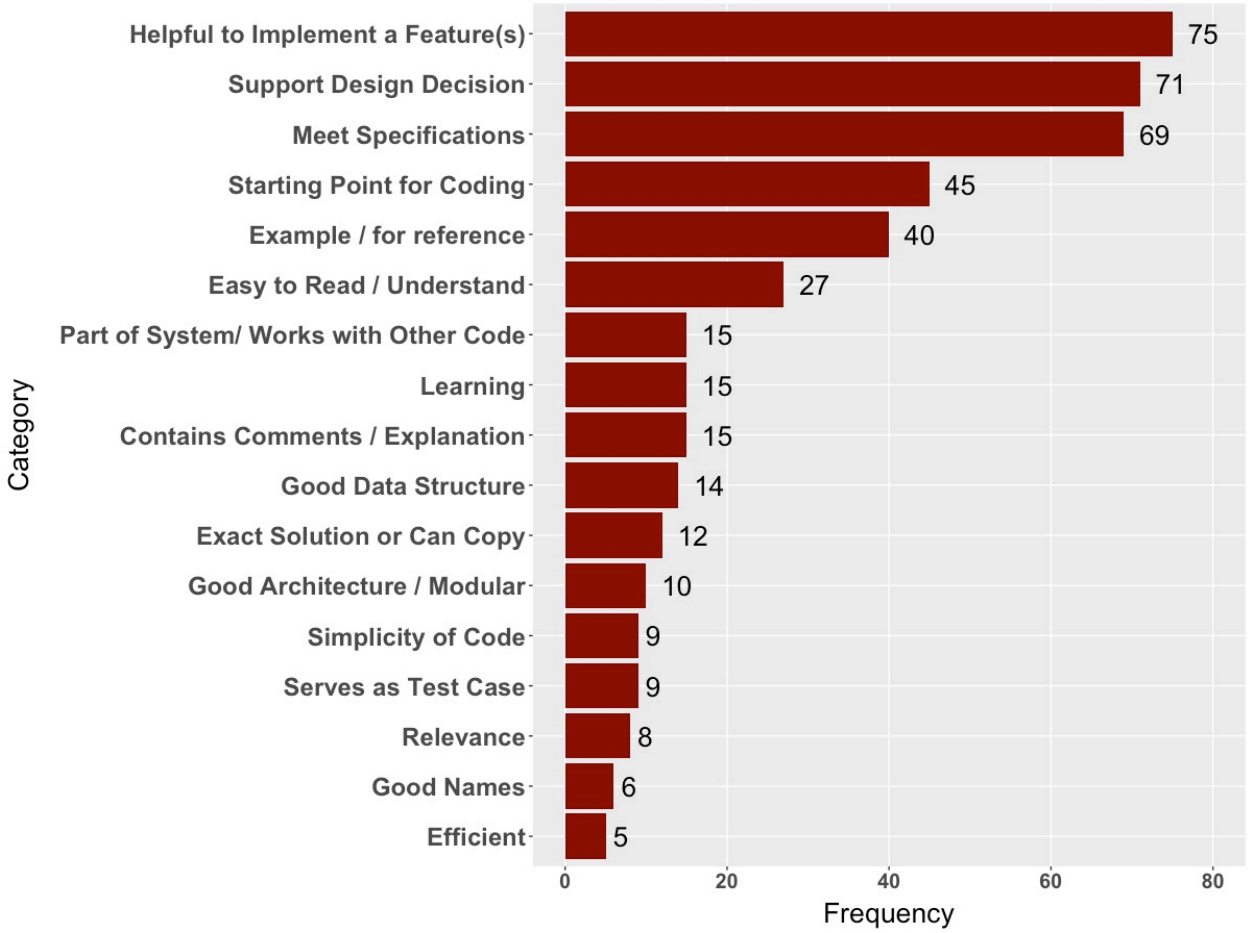
Figure 98. Average Iterative Features Used to Median Experience for Baseline.

diagramming session to cluster the "why" explanations. Each graduate student took an explanation and either assigned it a category (if one had already been created by the group) or created a new category to assign it to. The group of students were free to discuss names of categories and move explanations if needed. From the 463 "why" explanations, 28 categories emerged. Most reasons fell into five clusters; we show the top 17 clusters in Figure 99 (the other 11 clusters had only 1 to 3 explanations in them).

We found that our participants often selected code because it:

- helped implement a feature the participant had in mind (e.g., *A user would want to save their sketch, so we need…save and load…*)
- supported a design decision the programmer made about what the code should do (e.g., *I want to support 3D too.*)
- met the problem specification (e.g., *…performs the job really well and hence I choose it*)

Figure 99. Reasons Why Participants Selected Code.



- served as a starting point to solve the programming problem (e.g., *…an ideal starting point for building connect four…contains the correct sequence of gameplay…*)
- served as an example or a reference for how to solve the programming problem (e.g., *…demonstrates how calculators work…*)

The top two reasons for selecting code suggest that when the code being searched for is not completely specified (as in our tasks), that the programmer will make decisions on what to search for as they search. Often, they make decisions related to design (features they think are needed or other design decisions). These reasons might, in part, explain some of the

iterative search behavior we observed in our experiments and that were observed by others as well [7], [13], [48], [50], [97], [106], because making design decisions is often argued to be an iterative process [2,46]. However, often programmers found code simply because they felt it satisfied requirements. Interestingly, we also observed that many participants reported that the code they selected would serve as a useful starting place to write code, which contrasts with selecting code that will simply complete their task (as with strict copy-paste coding). This suggest that programmers sometimes are looking for a starting place to begin transforming for their own needs rather than the final solution.

6.2.6 Within Subjects Sub-Analysis

Our previous analyses have been between-subjects analysis, in which we measured the differences of experience, time, and success of iterative approaches across participants for each task. This allowed us to examine the impact of iterative approaches separately from who is searching and measure impact on participants in general rather than any particular participant. However, analyzing the differences in performance by individual participants lets us control for individual differences in our analysis and see if the effects we saw above also hold when the participant makes or is responsible for the comparison. Such an analysis is a within-subjects analysis. There are risks associated with such an analysis (e.g., learning effects, ordering effects, carry over effects from previous tasks), but we present a within-subjects analysis to see if the high-level story above is reflected here as well.

Typically, a within-subjects analysis might take two treatments and measure a dependent variable after each treatment for each participant. For example, a typical within-subjects analysis in our case would be to measure the time difference between completing task one with Google and then completing task one with CodeExchange for each participant. However, recall our experiment design is a mixed design and no participants repeats a task or uses all treatments, so we cannot conduct the typical within-subjects analysis. However, given our even task distribution among search engines, we can compare participant performance between two search engines on tasks that belong to the same sub-category (i.e., Find 4-No Specific Role, Find 4-Algorithm/Data Structure, Find 1- No Specific Role, and Find 1-Algorithm/Data Structure). For example, participants 1,7,13, and 19, used CodeExchange and Google to complete their tasks. Since task one and task five both are Find 4 and No Specific Role tasks, we can compare how each of those participants performed when trying to complete a Find 4-No Specific Role task with CodeExchange and with Google. For example, participant 1 gave an experience score of 6 for completing task 1 with CodeExchange and an experience score of 5 completing task 5 with Google. Since both task 1 and 5 belong to the same sub-category, our within-subject analysis says participant 1 had a better experience completing Find 4-No Specific Role tasks with CodeExchange than Google. We acknowledge there is a huge threat in doing this, because the tasks being different could cause the observed difference and likely to have an impact. Further, our within analysis is limited by the number of participants comparing any pair of treatments. In our case, only four participants compare a pair of treatments for each sub-category, giving us 16 data points to work with to conduct a within analysis for each pair of search engines. However, in the

Table 47. Within-Subjects CodeExchange Analysis.

CE > BL	CE == BL	CE < BL	CE > G	CE == G	CE < G
8	5	3	7	1	8
*p=0.10**			*p=0.93*		
CLT > BL	CLT == BL	CLT < BL	CLT > G	CLT == G	CLT < G
6	5	5	2	6	8
p=0.91			*p=0.02**		

interest of examining our result from a different perspective, we present the limited within-subject analysis here.

Table 47 presents our within analysis between each iterative approach and each non-iterative approach for experience. The header identifies how often an individual participant thought that CodeExchange or CodeLikeThis provided a better, equal, or worse experience in a sub-category. We found CodeExchange exchange was thought to provide a better experience than the base line and an equal experience to Google. CodeLikeThis fared worse, with being about equal to the base line and providing a worse experience than Google. The statistical significance was computed with a $\chi 2$ test on a 2X3 contingency table. These results mirror, to some extent, our high level story that Iterative approaches can significantly provide a better user experience than non-iterative approaches or an experience comparable to Google. However, the results also appear to "flip" somewhat from the between-subject analysis. CodeExchange performs comparably with Google in the within-subject analysis, but, in the between-subject analysis, Google performed significantly better. Further, Google performs significantly better than CodeLikeThis in the with-subject analysis,

Table 48. ANOVA Within Results on Time.

	Find 4 - No Specific Role	**Find 4 - Algorithm/Data Structure**	**Find 1 - No Specific Role**	**Find 1 - Algorithm/Data Structure**
CE	596.5 – G 397 – BL	1009.5 – G 690 – BL	256.25 – G 296 – BL	436.25 – G 323.5 – BL
CLT	802.25 – G 705.25 – BL	499.75 – G 877.75 – BL	274.25 – G 313.75 – BL	376 – G 246.5 – BL
G	810 – CE 585.75 – CLT	713.5 – CE 567.75 – CLT	239.25 – CE 223.25 – CLT	389.75 – CE 220.75 – CLT
BL	543.25 – CE 726.5 – CLT	356 – CE 875.5 – CLT	148.5 – CE 244.75 – CLT	415.25 – CE 301 – CLT
ANOVA	*p=0.677*	*p=0.715*	*p=0.329*	*p=0.922*

but, in the between-subject analysis, Google performed better but not significantly. These results suggest that when the participant is responsible for the experience comparison, thus controlling for individual differences, the experience results are different, but, overall, they tell a similar high level story.

We also performed a similar analysis as above, but focusing on comparing the times to paste all the required snippets. To do this analysis, we conducted a repeated measures ANOVA statistical test to see if there were any significant differences by sub-category. Table 48 presents the significance results from the ANOVA test for repeated measures, were we also present the mean times per treatment pairs. For example, CodeExchange under the Find 4 – No Specific Role sub-category, had a mean time of 596.5 seconds against Google, and a mean time of 397 seconds against the baseline. Similar to the results above, we find not statistical difference in time in a within-subject analysis.

6.2.7 End of Experiment Survey

Each participant ended their experiment by filling out a survey composed of eight questions about the two search engines they used to complete all their tasks. The goal of the survey was to capture some overall impressions of the search engines they used as a tool they might be able to use today in their current development practice. Each participant answered four questions about each of their search engines (making eight total questions) and the questions are presented in Table 49[4]. The questions themselves were taken from another user experiment comparing code search engines by Henninger [48], and were designed to evaluate if the search engines used during the experiment would help the participant in their current development practice.

Table 49. End of Experiment Survey.

ID	Statement	Agreement Score *(1=strongly disagree, 4=neutral, 7=strongly agree)*
1	Using ***<search engine name>*** would improve my performance in developing software.	1, 2, 3, 4, 5, 6, 7
2	I would find ***<search engine name>*** easy to use.	1, 2, 3, 4, 5, 6, 7
3	Using ***<search engine name>*** would enhance my effectiveness in developing software.	1, 2, 3, 4, 5, 6, 7
4	I would find ***<search engine name>*** a useful support tool for developing software.	1, 2, 3, 4, 5, 6, 7

[4] Note, the survey presented "strongly disagree" as "bad" and "strongly agree" as "great", but the participants were told to treat them as how much they agree with the presented statements.

Figure 100. Box Plot Summary of End of Experiment Survey.

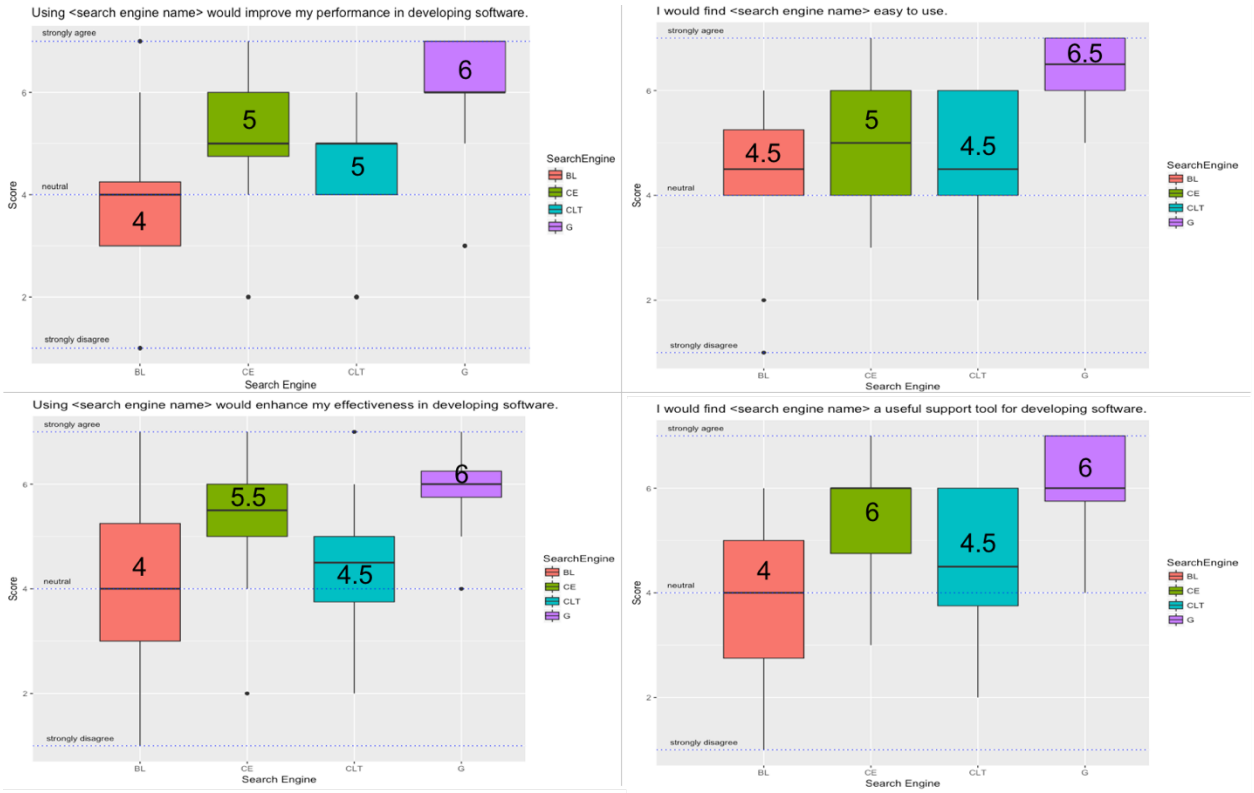


Figure 100 presents the box plot summary of the scores from all the participants answering each of the four questions. Each graph is titled with the question answered, where each box plot presents the spread of the scores for each search engine (identified by color and by the y-axis) and is labeled with the median score. We find that the iterative approaches always have a median positive response, with CodeExchange achieving a higher median than the baseline in all questions and CodeLikeThis achieving a higher median in three of four questions and equal in one. These responses suggest that the participants think that integrating iterative features into traditional code search engines would improve the performance of developing software, make the search engine easier to use, make programmers more effective in developing software, and make code search engines a more useful support tool.

Not surprisingly, Google scores high in many questions (highest median in three questions and tying for the highest median in one question). Google is heavily used today by programmers in their day to day work and can be used to search a vast amount of code indexed on the Internet (as we showed in Table 44 and Table 42). As such, our participants' answers to our survey questions about if Google would improve performance, is easy to use, would make the programmer more effective, and would be a useful support tool is more of a response to the reality of today rather than a hypothetical future. However, what we do find surprising is CodeExchange tying Google as a useful support tool for developing software. Recall, the major difference between CodeExchange and the baseline search engine are the iterative features (the index and the ranking algorithm are the same). This means the major difference, the design of the iterative features, made the participants score CodeExchange and Google equally as would be support tools for developing software.

The survey results also address the question of if code search engines could be useful at all. The participants' responses suggest that indeed they can be. While the non-iterative code search engine received overall neutral responses, the iterative approaches received positive responses. Further, the participants felt CodeExchange would serve as a useful support tool as strongly as they felt Google would serve as a useful support tool. This suggests that CodeExchange might be closer to something a programmer can use today in their day to day practice and that the performance of CodeExchange in our lab study might extend out into real world environments.

6.3 Threats to Validity

Several possible threats to validity exist with our study. First, while we put in our best effort to make this laboratory study realistic, it still lacks the realism one would find in a field study with each of the search engines. As such, further studies need to be performed to examine whether our results hold in real-world environments.

Second, as all of our participants have used Google for years, and Google indexes more code and different information than our prototypes, our experiment inherently is unbalanced. However, we felt it was important to tolerate this imbalance, since Google is so ubiquitous and represents a 'gold standard' for how developers search. It is not surprising to us that Google performed "better" on a number of cases, even though CodeExchange and CodeLikeThis offer the same ability to search with keywords only. What is more important, however, is that CodeExchange and CodeLikeThis did outperform Google on a number of cases to show the promise of dedicated support for iteration in search.

Third, the search tasks we used by no means cover all possible types of search tasks. We intentionally focused on broader search tasks, given the goal of this thesis of addressing code search when developers do not know exactly what they want and search around more exploratorily. However, even within this narrower focus, we could have chosen to use other search tasks. While we attempted to ameliorate this issue by modeling our tasks on real searches in real code search engines, a longitudinal study with CodeExchange and

CodeLikeThis in practice is needed to examine if their features apply beyond the eight search tasks we used.

Fourth, it is possible that participants did not seriously attempt the tasks. The author and two graduate students each individually inspected all snippets and explanations, assessing if they represented genuine attempts. In 97.3% of the cases, unanimous agreement was that they were genuine attempts (1 result was ranked not genuine by all 3 people, 6 results by 2, and 6 results by 1; these were spread across search engines and participants). This gives us confidence that most of our results represent genuine attempts by our participants.

Chapter 7

Discussion

In this chapter, we take a step back and look at the broad implications and "big picture" of the results in Chapter 6. We first return to the research question and discuss, at a higher level than Chapter 6, what the results mean for the design of code search engines. We then return to our initial perspective taken on code search and reflect what the results suggest is the bigger picture.

7.1 Answer to Research Question

Our research question concerned the impact on the experience, time, and success of the code search process of explicitly supporting software developers in searching iteratively. In the following subsections, we discuss what our results suggest might be the answers to this question through the articulation of eight lessons learned. Further, we examine some of the

Table 50. Search Engines by Components.

	Specificity Ranking Algorithm	RankBrain Ranking Algorithm	ST-Hybrid Ranking Algorithm	Code Index	Google Web Index	Iterative Support on Aspects	Iterative Support on Entire Result
CodeExchange	▨			▨		▨	
Baseline	▨			▨			
CodeLikeThis			▨	▨			▨
Google		▨			▨		

answers by the components that largely define each of the search engines as shown in Table 50, where each search engine is mapped to its defining components by the cells that are shaded. For example, CodeExchange is defined by the Specificity ranking algorithm, the code index it searches over, and the iterative features to use aspects/qualities of the results in creating the next query.

Lesson 1: All major components being equal, adding support to iteratively search with aspects/qualities of the results can significantly increase the user's experience compared to a traditional code search engine.

We found that support for using the results from one query in creating the next can significantly improve the user experience compared to a traditional code search engine (as exemplified by our baseline). Specifically, when we kept major components the same between our baseline and CodeExchange (i.e., index and ranking algorithm) except for the iterative features in CodeExchange, we saw a statistically significant improvement in the participants' experience in completing code search tasks with CodeExchange (both in the between and within analysis). These findings suggest that adding iterative features

supporting using aspects/qualities to a code search engine will significantly improve the user's experience. However, contrasting with the comparison between the baseline and CodeExchange, are our results when comparing the baseline with CodeLikeThis. Specifically, when we altered the ranking algorithm to the ST-Hybrid ranking algorithm and offered only the like-this query method to iteratively search, we found that CodeLikeThis only performed marginally better than the baseline search engine (both in the between and within analysis).

Lesson 2: Supporting using aspects/qualities of the results in creating the next query provides a better experience for broader search tasks and supporting using an entire result in creating the next query provides a better experience for more focused tasks.

When we compared experience scores between CodeExchange and CodeLikeThis, we found that they exactly complemented each other in the kinds of search tasks they support. In particular, we found that CodeExchange provided a better experience than CodeLikeThis for the broader tasks and CodeLikeThis provided a better experience for the more focused tasks. Since the iterative features provided by both search engines were often used by the participants, these results suggest that the best way to iteratively search depends on the search task. Specifically, how much the user knows about what they are looking for might determine if they should use like-this queries (for more focused tasks) or queries using aspects/qualities of the results (for broader tasks). However, in order to use like-this queries effectively for more focused tasks, the ST-Hybrid ranking algorithm has to be used, as we do

not know from our experiments whether other ranking algorithms would lead to similar outcomes. Such a comparison will need to be performed as part of future research.

Lesson 3: All major components being unequal, adding iterative features may not improve the search experience compared to Google.

When we compared CodeExchange and CodeLikeThis with Google, we found less impact of the iterative approaches. The between analysis found Google performed significantly better than CodeExchange (Google higher five times, equal two times, and lower once) and better than CodeLikeThis (Google higher five times, equal once times, and lower twice), though in this case not significantly. In the within analysis, Google and CodeExchange performed about equally (Google higher eight times, equal once, and lower seven times), but Google provided a significantly better experience than CodeLikeThis (Google higher eight times, equal six times, and lower two times).

The results suggest that using Google's RankBrain algorithm and very large web index can provide a better experience than code search engines that have a different ranking algorithm and a much smaller code index, even though the latter provides support for iteratively searching for code. Given the participants' past experiences with Google and the bias that might cause, the results are somewhat hard to interpret. However, the results do suggest that just adding iterative features to any ranking algorithm and/or index is not sufficient to create the best search experience.

Lesson 4: CodeExchange and CodeLikeThis, together, are comparable to the state of the art in search engines used for code search.

Given that CodeLikeThis and CodeExchange complement each other, we looked again at our results and asked if either iterative approach can provide a higher experience score than the baseline and Google. Combined, we found that the two iterative approaches provided a significantly better experience compared to the baseline (six higher and two ties), but only slightly better than just CodeExchange alone (six higher, one tie, and one lower). We also found that the iterative approaches provided a comparable experience to Google (three higher, two ties, and three lower). These results suggest that, combined, either as one or used together, our iterative approaches could potentially outperform the baseline and perform comparably to Google. Yet, the differences in Google's index and the index used by the iterative approaches is a confounding factor and means that Google's performance is, in part, a consequence of the difference in indexes. We, of course, do note that combining CodeExchange and CodeLikeThis into one tool (since this is a hypothetical at this point) may not do as well, since adding more features to a search engine may confuse users. Nonetheless, if both methods of iterative search can be elegantly combined, we believe that the potential is to be as good as Google, even with a smaller index. This clearly is an experiment to be performed, as is running the combination on the same index used by Google.

Lesson 5: Time spent searching is often not correlated to the user's experience for broader search tasks.

In general, we find very few correlations (by task, category, or search engine) between time spent searching and experience for the broader search tasks. Interestingly, we did find time was negatively correlated with experience for broader tasks with CodeExchange, yet it appears to not have impacted the overall positive experience the participants had with it. While CodeExchange had two to three times more queries issued with it than the other search engines, requiring more time to search, CodeExchange's iterative features were found to significantly improve the user's search experience. This suggests that the positive user experience gained by using iterative features may have made the extra time worth the cost. We illustrate what this means in Figure 101 , which mirrors ours results in the previous chapter. The y-axis is experience and the x-axis is time, and the line represents the negative correlation between time and experience for CodeExchange. While more time using CodeExchange appeared to degrade the user experience, the support offered by CodeExchange kept the regression line mostly in the positive experience region.

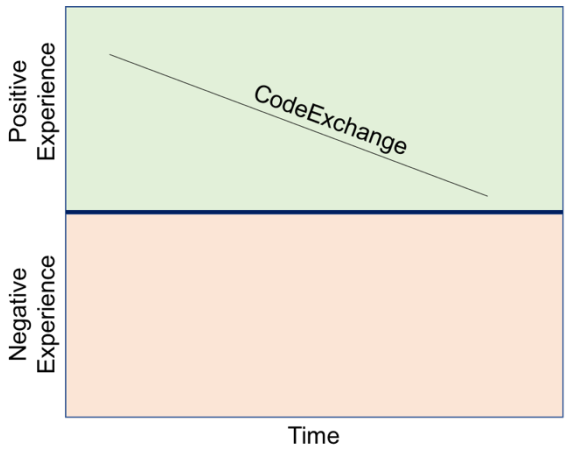


Figure 101. Illustration of Negative Correlation but Positive Experience.

Lesson 6: Time spent searching is often negatively correlated to the user's experience for more focused tasks.

In contrast with broader tasks, we found several negative linear correlations (by task, category, and search engine) between time and experience for more focused tasks. This suggests that the time spent searching for more focused tasks is more critical for the user's experience. In fact, regardless of search engine, our linear regression models predicted that after about eight minutes of searching, the search experience becomes negative for the user. While using iterative features was associated with higher experience scores, using them too much (beyond eight minutes) for more focused tasks is a sign the search did not go as well.

Lesson 7: The ideal number of times to use iterative features looks different for CodeExchange and CodeLikeThis.

We saw that our participants used the iterative features of CodeExchange and CodeLikeThis, and we asked what is the ideal amount of usage? To answer this question, we examined the relationship between the number of usages of the iterative features and experience. While we found answers for an ideal number of times to use iterative features, the possible reasons for these numbers are not necessarily obvious and appear to be context dependent.

We found that with CodeExchange and using the iterative features a moderate amount (5 to 7 times on average) was associated with the best experience scores of CodeExchange. Lower (3 on average) or higher (10 on average) usage was associated with lower experience scores. In contrast, we found that CodeLikeThis provided the best experiences when the users issued 1 or 2 queries, on average, with the iterative features and provided worse experiences when issuing more queries (3 or 4 queries on average) with the iterative features.

A better amount of usage being a moderate amount for CodeExchange might be a result of the interplay between the kinds of tasks best supported by CodeExchange and the degrading effect time can have on the user's experience. The relatively lower number of queries issued with iterative features happened for the more focused tasks, when less queries were issued on average and CodeExchange appeared to offer a less positive user experience in comparison to the broader tasks. The moderate and higher usage happened mostly for the

broader tasks, where a higher number of queries were issued on average and CodeExchange appeared to offer a more positive user experience in comparison to the more focused tasks. This explains why more usage was associated with higher experience scores. However, given that there is a negative linear correlation between time and experience for boarder tasks with CodeExchange, the highest usages are not associated with the highest experience scores because of the cost of spending more time issuing more queries.

The better amount of usage being a lower amount for CodeLikeThis might be a result of the kinds of tasks CodeLikeThis best supports. Specifically, we found the lower usages of the iterative features happened for the more focused tasks where fewer queries were issued on average and CodeLikeThis appeared to offer a more positive user experience in comparison to the broader tasks. The higher usage of the iterative features happened for the broader tasks were more queries were issued on average and CodeLikeThis appeared to offer a less positive user experience. However, because there was no correlation between time and experience for broader tasks with CodeLikeThis, the worse experience found for higher usages is not because of more time spent issuing queries, but because the like-this queries were less effective for broader search tasks.

Lesson 8: Inability to complete a search task often results in a negative search experience.

Our results suggest that providing explicit support in iteratively searching for code can improve success, but not significantly. Specifically, we measured success as being able to complete a task or not, and we found that participants could complete more tasks with CodeLikeThis, but not significantly more. However, completing tasks is very import to our participants.

We found that the median experience participants had when they did not finish a task was negative. Interestingly, this means that the impact of not completing a task on experience acts a category changer. Regardless whether the experience begins positive for the user while searching, not finishing a search tasks changes the experience to negative. This contrasts with the negative linear correlation we found between time and experience, which showed time acts more as a gradually degrader of experience. That is, the time taken to finish a task does not necessary make the experience negative, but rather can make the experience less positive.

We also found that certain kinds of search tasks are more difficult than others. We found higher occurrences of incompletion in the Find 4 tasks (a broader task type) and the Algorithms/Data Structure tasks (a more focused task type). This suggests that both broader and more focused tasks can be difficult to complete.

7.2 A New Perspective on Code Search

We take yet another step back and re-examine the perspective of code search taken in this thesis by reflecting on what the results suggest is the bigger picture and what it might mean for new code search engine designs. As we discussed in Chapters 1 and 2, the dominant perspective of code search reduces search to a query and returning the "best" results for that query. With this picture of search, it might not be surprising that the main advances in previous research have been on the central entities involved in this picture – the query and the ranking algorithm. Much of the previous research made advances in expressive queries, ranking algorithms, query formation support, and code result quality. Further, evaluation of these advances is often done by a group of researchers that predetermine what code should or should not be returned for a query and evaluate a ranking algorithm's ability to match that set of predetermined code. However, this picture of search is incomplete — the user is missing — which is particularly crucial since the original intention behind all information retrieval tools is to satisfy the user's "information needs" [153].

In this thesis, we investigated a new perspective that includes the user by seeing what happens to the design of a code search engine when the user's needs shape the requirements. By needs we do not necessarily mean the user's opinion of what the design should be, but we go deeper and mean the cognitive processes of the user that shape what the search engine should be like. Cognitive processes of the user (e.g., learning, remembering, idea generation, pattern recognition) are natural processes long evolved over millions of years that shape the user's behavior.

In Chapter 2, we discussed some of the reasons developers have for searching for code (e.g., remembering, idea generation, and learning) and that, when searching, they do so iteratively. Based on this, we took the perspective that the search process might be inherently iterative because many the reasons why the developers were searching are cognitive in nature and might require iteration. With this perspective, we evaluated what happens when searching iteratively is explicitly supported.

Our results paint a picture of code search that in some ways match our original guess that search is iterative and needs support, but in other ways offer a much richer and nuanced reality. Our results suggest that the iterative behavior observed in developers is due to their needs rather than a fault of the search engine. Specifically, our finding that iterative features can significantly improve the user experience over a traditional code search engine and it appeared that users, when using Google, were often taking words from web pages and using them in their next queries implies that iteratively searching is what users need to do (perhaps unknowingly). Additionally, our finding that it appeared participants, when using Google, often took words from web pages and used them as a keyword in their next query, also supports the idea that iteratively searching with the results is what users want to do. Further, our results suggest that this iterative behavior is nuanced and a reflection of a more complex search process than previously thought. Specifically, how developers iteratively search might reflect how much they know and require different support. We found that when developers have a more complete idea of what they are searching for, they have a more positive experience using an entire result as the next query. On the other hand, when

developers have a partial idea of what they were searching for, they have a more positive experience using aspects/qualities of the results in creating the next query.

In Chapter 5, we also saw that the cognitive needs of developers can change the traditional requirements of ranking algorithms. Specifically, we saw that developers often prefer result sets that are diverse. Preferring diverse results is somewhat startling when looked at from the traditional perspective of search, because it means preferring a set of results that are not the most on topic results and, thus, preferring results that might be at lower ranks because they are different from the other results. However, diversity of results begins to make sense when the user's needs are added to the old perspective. When users search for code, they often are not certain what they want (e.g., when generating ideas or learning), so when they issue a query, that query is not a reflection of a well-formed idea of what is being searched for, but a reflection of a "fuzzy" or broad sense of what is needed. In this new way of looking at search, giving the user results all matching the query in a variety of ways, rather than in one way, makes sense, because it informs the user of what exists and, thus, informs their next query.

Our results show that this new picture does not exclude the old way of looking at code search, but rather expands it. We saw that Google performed very well by providing many positive experiences for our participants while searching. Further, our results suggest that simply adding iterative features to a search engine with a much smaller index than Google is not sufficient to create a much better experience. Keywords also remain critical to search. We saw they were still heavily used in all search engines and serve as a simple way to begin

searching. In our new perspective on code search, then, it appears as if the best performing search engine looks like one with a large index like Google's, with a high-performance ranking algorithm that diversifies the results when needed, and supports iteratively searching with features that differ and are tailored to different scenarios as to what the user may be searching for, how much knowledge they already have, and what kind of iterations on the results may be needed.

Chapter 8

Conclusion

Programmers frequently use search engines to search for code as part of their practice [97], [108] and, as such, software engineering researchers are investigating how to improve code search engines. While many different approaches to improving code search exist, these approaches are generally similar in one very visible design decision: they are non-iterative approaches in expecting a query and optimizing on returning the best matching results for that query.

While a search engine that returns the results the programmer is looking for after the first query appears ideal, many times the programmer is not sure what they are looking for and does not search for code with a single query. Instead, they issue multiple queries [7], [13],

[50], [106], [115], where, after receiving results, the programmer modifies their query by removing keywords, adding keywords, or some combination of both, and repeats this process multiple times [7], [50], [106]. That is, search looks like an iterative process where programmers often submit a query, get results, reflect on and learn from the results, submit a modified query in response to the results, get new results, and so on, until the programmer stops searching.

This dissertation investigated what happens when programmers are explicitly supported in searching iteratively for code. It particularly answered the following research question:

> *What is the impact of explicitly supporting software developers in searching iteratively on the experience, time, and success of the code search process?*

We addressed this research questions by developing new approaches for explicitly supporting iteratively searching for code and evaluating them. Doing so resulted in six contributions.

8.1 Contribution 1 — Using Aspects of Results

Our first approach, implemented in CodeExchange, to support iteratively searching was specifically designed to aid the developer when, initially, the programmer is uncertain of exactly what they are searching for and is engaged in a more exploratory search involving the submission of multiple queries through which to explore what examples may be

available. In such a search scenario, the insight behind CodeExchange is that the next query tends to be relative to the results, and often to specific aspects of the results of the previous query. CodeExchange supports the user with four specific features for creating a next query: language constructs, critiques, query refinement recommendations, and query parts.

8.1.1 Language Constructs

Language constructs support the developer in selecting structural characteristics of a result (e.g., method calls, interfaces implemented, or code imported) to bring those characteristics into the query. Using a language construct yields a query that is a mix between keywords, if they were a part of the query before selecting a language construct, and characteristics of results. Unlike keywords, that may or may not retrieve code matching a topic described by the keywords, a language construct constrains the query to retrieve code exactly matching the characteristics specified by that language construct.

8.1.2 Critiques

The second feature, critiques, supports the developer in selecting the value of different technical qualities (complexity, size, number of imports) of a result as a lower or upper bound to bring that bound into the query to constrain the next set of results. In this way, if the developer feels a code result is lacking (e.g., too long or not complex enough), they can bound the next set of code results to attempt to avoid that quality.

8.1.3 Query Refinement Recommendations

In contrast to modifying the query relative to a specific result, query refinement recommendations present the user with common aspects (imports, parent classes, or interfaces) or domain related terms across all the results to add to the query. The recommendations help make visible to the programmer common aspects of results that are difficult to infer just from the top results that are actually visible. After adding a recommendation to the query and getting the results, the recommendations are updated again using the newly returned results. In this way, the programmer can iteratively search by continually selecting query refinement recommendations.

8.1.3 Query Parts

Query parts modularize the programmer's query each time a programmer adds to a query. Whether by providing one or more new keywords or using one of the new features of CodeExchange, the addition is separately identified by CodeExchange in its interface. Each query, then, consists of a set of separate parts that, together, form the actual query issued, but to the programmer remain individual components. Query parts leverage this by enabling a programmer to turn off / turn back on each of these parts separately. In this way, after a programmer gets new results, they can respond by trying different combinations of their query parts to search in different "directions".

8.2 Contribution 2 — Using an Entire Result

Our second approach, implemented in CodeLikeThis, is also specifically designed to aid the developer when, initially, the programmer is uncertain of exactly what they are searching for and is engaged in a more exploratory search involving the submission of multiple queries through which to explore what examples may be available. In such a scenario, the insight behind CodeLikeThis is that the next query tends to be relative to the results and sometimes specific to an entire result rather than any specific aspect of a result (unlike the scenarios supported by CodeExchange). As such, CodeLikeThis supports the developer in forming the next query out of the entire result simply by selecting if they want code that is similar to that result. In this way, CodeLikeThis changes how the next query is constructed, from choosing and entering keywords, to creating a query simply by choosing a result to see code that is similar to it. CodeLikeThis supports the user with two specific features for creating a next query: a diversity ranking algorithm and like-this queries.

8.2.1 Diversity Ranking Algorithm

While, typically, the results after the first keyword query are optimized to be the most topically related to the keywords, CodeLikeThis also needs to address giving the user results to select from to begin iteratively searching. If CodeLikeThis only returns the most topically related results, then the results themselves may all be very similar to each other and hide examples that are different, but still on topic, that the programmer could recognize as similar to what they are searching for. CodeLikeThis addresses this issue with its diversity ranking algorithm that first gives the user a diverse set of code matching the initial keywords but

different from each other. Each code result, then, can equally serve as a point of comparison that the programmer can use to find other similar or dissimilar code.

8.2.1 Like-This Queries

Once the programmer discovers a result they might want to use for finding similar code, the programmer can use like-this queries. Specially, like-this queries support the programmer in retrieving code in the entire search engine by how similar they are to the result the programmer has chosen. Like-this queries do not act as modifications to previous queries (as was the behavior in CodeExchange), but rather replace the previous query with a like-this query. Moreover, three types of like-this queries are supported by CodeLikeThis. If the programmer wants additional results that are very similar to the result they have chosen, then they can issue a *more-like-this* query to retrieve the code examples, from the search engine's index, closest in similarity. If the programmer wants code that is less like a result, they can issue a less-like-this query to retrieve code that is more different than similar, but still having exhibiting similarity, to the chosen result. Finally, if the programmer wants code that is like a result but feels they also are looking for different code, they can issue a *somewhat-like-this* query, which returns code examples that are not as similar as those returned from the more-like-this query but also not as different as those returned from the less-like-this query.

8.3 Contribution 3 — Laboratory Experiment

To answer our research question, we conducted a laboratory experiment involving 24 developers measuring the experience, time, and success of each participant in searching for code with our iterative approaches as well as two non-iterative approaches, a baseline and Google. The participants sat alone in a room completing eight different and independent search tasks that were designed to cover a space of tasks that range from broad to more focused.

We found that explicitly supporting using the results from one query in creating the next can significantly improve the user experience compared to a traditional code search engine (our baseline). Specifically, when we kept major components equivalent between our baseline and CodeExchange (i.e., index and ranking algorithm), we saw a statistically significant improvement in the participants' experience in completing code search tasks with CodeExchange. These results suggest that adding CodeExchange's iterative features to a search engine will significantly increase the search experience. However, we saw only a marginal improvement in user experience with CodeLikeThis.

We found that CodeExchange and CodeLikeThis were complementary in the kinds of tasks they supported. That is, CodeExchange provided a better user experience for the broader tasks and CodeLikeThis provided a better user experience for the more focused tasks. Our results, then, suggest that, when the search task is broader, and the user has less of an idea of what they are looking for, then iteratively searching with aspects of the results is better.

On the other hand, when the search task is more focused, and the user has more of an idea of what they are looking for, then iteratively searching by similarity of a result is better.

We found that adding iterative features is insufficient to provide the best experience, however. When compared to Google, CodeExchange only provided a better experience once and equal twice, and CodeLikeThis only provided a better experience twice and equal once. Google, then, provided a better experience five times when compared to both individually. In the within analysis we found CodeExchange and Google were comparable in user experience even though Google often provided a better experience than CodeLikeThis. However, given that CodeExchange and CodeLikeThis are complementary, we compared when either iterative approach provided a better experience than Google (evaluating iterative design approaches more generally but also evaluating a hypothetical search engine when both approaches are available in one interface). We found that the approaches became comparable, where an iterative approach provided a better experience than Google three times, equal two times, and both lower three times. These results suggest that two iterative approaches exist that together (e.g., in a hypothetical search engine with access to both approaches) are comparable in user experience to Google. These results, however, also suggest iterative features are not enough to support the best experience for code search. Given that we found, when all things are equal, adding iterative features improves experience, then the best performance might be obtained by putting CodeExchange or both CodeExchange and CodeLikeThis on Google's index and/or ranking algorithm.

8.4 Contribution 4 — New Findings

With our iterative approaches and laboratory experiment we were able to answer our research question and find nuances in the answer. However, the data we collected revealed additional observations. Specifically, we found that, while time is important for our participants, it is not the deciding factor for their experience. However, not being able to complete a task usually always results in a negative experience. We also found new reasons for why programmers select code to use off the Internet.

For more focused tasks, we found that time usually acts a gradual degrader of experience and that usually about after eight minutes of search the experience turns negative. On the other hand, taking more time to search by issuing more queries does not mean a worse experience than taking less time to search with fewer queries. In particular, we found that users issued two to three times more queries with CodeExchange, resulting in more time spent searching. Yet, CodeExchange often provided a better user experience than the baseline. For broader tasks, we found time was usually not a factor for experience, so the differences we found in experience might be more related to the effectiveness of the features of the search engine.

Unlike time, a gradual degrader of experience, task success seems to act more as a categorical changer of experience. We found that, when participants in our laboratory experiment were unable to finish a task, they typically had a negative experience. While more time taken might

mean a lower positive experience, not finishing a task often means a negative experience and, thus, has a more severe consequence on the user.

While previous research has shown that programmers search to learn, get ideas, remember, or simply copy and paste, our results speak to why programmers select code they find. Specifically, when the code being searched for is not completely specified (as in our tasks), the programmer will make decisions on what to search for as they search. Often, they make decisions related to design (features they think are needed or other design decisions). Other times, programmers selected code simply because they felt it satisfied requirements or would serve as a useful starting place to write code that would satisfy requirements.

8.5 Contribution 5 — Implementations

With this thesis, we also contributed working implementations of our iterative approaches that are publicly available. Implementing the iterative approaches was necessary, not only to approximate Internet-scale search engines for the lab experiment, but also to determine the feasibility of the approaches. Particularly in the early stages of this research, it was necessary to explore whether the approaches could be reified as search engines with reasonable response times (on the order of seconds) and with usable interfaces.

We presented the architectures of each approach and demonstrated each implementation searching over indexes of Java classes about 10M in size. Further, we detailed how the CodeLikeThis architecture can leverage the fact there are only 30 possible next queries by

prefetching all possible next results while the user is looking at the current results – making the response time appear faster.

8.6 Contribution 6 — Code Search Index

To create implementations of our approaches that approximate Internet-scale search engines, we needed to create a relatively large index of code to search over. To create such an index, we mined about 10M classes off github.com from about 300,000 repositories written in Java. The classes we mined were indexed under a variety of technical and social properties extracted from the classes' abstract syntax trees and meta data extracted from the downloaded repository. While we used this index for code search, it can be used for a variety of purposes (e.g., running statistics on the evolution of code or running a natural language analysis on commit messages), and, as such we have made it publicly available. However, code often is removed or changed on GitHub, so we also have made available the complete list of repositories that we mined, as well as the software used to mine them, so that indexes can be recreated or created at various sizes and with more up-to-date code/information from GitHub.

8.7 Future Work

Our results suggest several avenues of future investigation. The most immediate future work is to evaluate what the evaluation results suggest is an even better code search engine. That is, we wish to explore the design and evaluation of a search engine where both CodeExchange and CodeLikeThis are integrated and search over Google's search index and/or use Google's

ranking algorithm. This might mean simply putting CodeExchange and CodeLikeThis on Google's search index and using them side by side, or designing a new interface that integrates CodeExchange and CodeLikeThis functionally. Another approach might be to extend Google's interface to support iterative features modeled after those of CodeExchange and CodeLikeThis.

Our results also suggest that a much deeper investigation needs to be performed on how developers think and the impact it has on search. Not only did we find that developers search iteratively, but that different ways of thinking, depending on knowledge, might imply different requirements for tool support. We found that, when tasks were more focused, using like-this queries provided a better experience than using aspects/qualities of the results. When the task was broader, on the other hand, we found that using aspects/qualities of the results provided a better experience. This could be a consequence of the cognitive process involved in searching. That is, developers are better able to recognize a result as similar to what they are looking for if they have a more complete idea of what they are looking for (as in more focused tasks). On the other hand, developers are worse at recognizing a result as similar to what they are looking for if they have less of an idea of what they actually want (as in broader tasks. Still, it appears that aspects/qualities of the results can match the partial knowledge they do have in that case.

While CodeLikeThis generalizes like-this queries so that it is possible to issue more, somewhat, and less like-this queries, it must apply them to an entire result. However, we found that sometimes when our participants issued a like-this query, the results did not

match what they had in mind. As such, we wish to explore alternative approaches to issuing like-this queries that more closely match what the user is thinking. One approach to explore is supporting the user in highlighting a portion of a code result in which they are interested and to use that highlighted portion to find other code with contents similar to what the user highlighted. In this way, the search engine avoids matching against portions of the code the user did not have in mind. Further, it might be possible to borrow query parts from CodeExchange and to make each highlight a query part, so that each highlight acts as a refinement. The collective set of highlights, then, is used to find other similar code — matching code that is partially similar to various past results. In this way, we could support the user in issuing a like-this query for results closer to what they had in mind, but also support incrementally searching with aspects (i.e., highlights) of the results when the user is engaged in a broader search task.

Bibliography

[1] 262588213843476, "10 Million Repositories," *GitHub*, 22:56:22 UTC. [Online]. Available: https://github.com/blog/1724-10-million-repositories. [Accessed: 02-Nov-2016].

[2] D. Ahl, Ed., *The Best of Creative Computing: Volume 2*. Creative Computing Press, 1977.

[3] A. Bacchelli, L. Ponzanelli, and M. Lanza, "Harnessing Stack Overflow for the IDE," in *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*, 2012, pp. 26–30.

[4] S. Bajracharya *et al.*, "Sourcerer: A Search Engine for Open Source Code Supporting Structure-Based Search," in *In Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06*, 2006, pp. 25–26.

[5] S. K. Bajracharya, "Facilitating Internet-Scale Code Retrieval," Ph.D., University of California, Irvine, United States -- California, 2010.

[6] S. K. Bajracharya, *Facilitating Internet-Scale Code Retrieval*. Irvine, Calif: University of California, Irvine, 2010.

[7] S. K. Bajracharya and C. V. Lopes, "Analyzing and Mining a Code Search Engine Usage Log," *Empir. Softw. Eng.*, vol. 17, no. 4–5, pp. 424–466, Aug. 2012.

[8] S. Bajracharya, J. Ossher, and C. Lopes, "Searching API Usage Examples in Code Repositories with Sourcerer API Search," in *Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*, New York, NY, USA, 2010, pp. 5–8.

[9] O. Barzilay, C. Treude, and A. Zagalsky, "Facilitating Crowd Sourced Software Engineering via Stack Overflow," in *Finding Source Code on the Web for Remix and Reuse*, S. E. Sim and R. E. Gallardo-Valencia, Eds. Springer New York, 2013, pp. 289–308.

[10] W. Beaton, "Eclipse Corner Article: Abstract Syntax Tree." [Online]. Available: https://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html. [Accessed: 04-Apr-2017].

[11] D. M. Blei, A. Y. Ng, M. I. Jordan, and J. Lafferty, "Latent Dirichlet Allocation," *J. Mach. Learn. Res.*, vol. 3, p. 2003, 2003.

[12] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-Centric Programming: Integrating Web Search into the Development Environment," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2010, pp. 513–522.

[13] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2009, pp. 1589–1598.

[14] M. Bruch, M. Monperrus, and M. Mezini, "Learning from Examples to Improve Code Completion Systems," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, New York, NY, USA, 2009, pp. 213–222.

[15] M. Bruch, T. Schäfer, and M. Mezini, "FrUiT: IDE Support for Framework Understanding," in *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, New York, NY, USA, 2006, pp. 55–59.

[16] R. P. L. Buse and W. Weimer, "Synthesizing API Usage Examples," in *Proceedings of the 34th International Conference on Software Engineering*, Piscataway, NJ, USA, 2012, pp. 782–792.

[17] J. Carbonell and J. Goldstein, "The Use of MMR, Diversity-Based Reranking for Reordering Documents and Producing Summaries," in *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, New York, NY, USA, 1998, pp. 335–336.

[18] S. Chatterjee, S. Juvekar, and K. Sen, "SNIFF: A Search Engine for Java Using Free-Form Queries," in *Fundamental Approaches to Software Engineering*, M. Chechik and M. Wirsing, Eds. Springer Berlin Heidelberg, 2009, pp. 385–400.

[19] C. L. A. Clarke *et al.*, "Novelty and Diversity in Information Retrieval Evaluation," in *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, New York, NY, USA, 2008, pp. 659–666.

[20] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*, 2 edition. Hillsdale, N.J: Routledge, 1988.

[21] N. Craswell and D. Hawking, "Overview of the TREC 2004 Web Track," in *Thirteenth Text Retrieval Conference*, Gaithersburg, Maryland, 2004, vol. Special Publication 500-261.

[22] D. R. Cutting, D. R. Karger, J. O. Pedersen, and J. W. Tukey, "Scatter/Gather: A Cluster-based Approach to Browsing Large Document Collections," in *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, New York, NY, USA, 1992, pp. 318–329.

[23] B. Dagenais and H. Ossher, "Automatically Locating Framework Extension Examples," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, New York, NY, USA, 2008, pp. 203–213.

[24] J. E. Davidson and R. J. S. PhD, Eds., *The Psychology of Problem Solving*. Cambridge, UK ; New York: Cambridge University Press, 2003.

[25] F. Deissenboeck and M. Pizka, "Concise and Consistent Naming," *Softw. Qual. J.*, vol. 14, no. 3, pp. 261–282, Sep. 2006.

[26] A. Deshpande and D. Riehle, "The Total Growth of Open Source," in *Open Source Development, Communities and Quality*, Springer US, 2008, pp. 197–209.

[27] F. Détienne, *Software Design—Cognitive Aspects*. New York, NY, USA: Springer-Verlag New York, Inc., 2002.

[28] F. A. Durão, T. A. Vanderlei, E. S. Almeida, and S. R. de L. Meira, "Applying a Semantic Layer in a Source Code Search Tool," in *Proceedings of the 2008 ACM Symposium on Applied Computing*, New York, NY, USA, 2008, pp. 1151–1157.

[29] A. E. Eiben and S. K. Smit, "Parameter Tuning for Configuring and Analyzing Evolutionary Algorithms," *Swarm Evol. Comput.*, vol. 1, no. 1, pp. 19–31, Mar. 2011.

[30] S. S. Epp, *Discrete Mathematics with Applications*, 3 edition. Belmont, CA: Brooks Cole, 2003.

[31] E. Fast, D. Steffee, L. Wang, J. R. Brandt, and M. S. Bernstein, "Emergent, Crowd-scale Programming Practice in the IDE," in *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, New York, NY, USA, 2014, pp. 2491–2500.

[32] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, *How to Design Programs: An Introduction to Programming and Computing*, 1st edition. Cambridge, Mass: The MIT Press, 2001.

[33] G. Fischer, S. Henninger, and D. Redmiles, "Cognitive Tools for Locating and Comprehending Software Objects for Reuse," in *[1991 Proceedings] 13th International Conference on Software Engineering*, 1991, pp. 318–328.

[34] S. D. Fleming *et al.*, "An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 2, p. 14:1–14:41, Mar. 2013.

[35] W. B. Frakes and B. A. Nejmeh, "Software Reuse Through Information Retrieval," *SIGIR Forum*, vol. 21, no. 1–2, pp. 30–36, Sep. 1986.

[36] R. E. Gallardo-Valencia and S. Elliott Sim, "Internet-Scale Code Search," in *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, Washington, DC, USA, 2009, pp. 49–52.

[37] R. E. Gallardo-Valencia and S. E. Sim, "What Kinds of Development Problems can be Solved by Searching the Web?: A Field Study," in *Proceedings of the 3rd International*

Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation, New York, NY, USA, 2011, pp. 41–44.

[38] X. Ge, D. Shepherd, K. Damevski, and E. Murphy-Hill, "How Developers Use Multi-Recommendation System in Local Code Search," in *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2014, pp. 69–76.

[39] G. Glass and K. Ables, *UNIX for Programmers and Users*, 3 edition. Upper Saddle River, N.J: Pearson, 2003.

[40] M. Goldman and R. C. Miller, "Codetrail: Connecting Source Code and Web Resources," *J. Vis. Lang. Comput.*, vol. 20, no. 4, pp. 223–235, Aug. 2009.

[41] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A Search Engine for Finding Highly Relevant Applications," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, New York, NY, USA, 2010, pp. 475–484.

[42] S. Guo and S. Sanner, "Probabilistic Latent Maximal Marginal Relevance," in *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, New York, NY, USA, 2010, pp. 833–834.

[43] F. S. Gysin and A. Kuhn, "A Trustability Metric for Code Search Based on Developer Karma," in *Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*, New York, NY, USA, 2010, pp. 41–44.

[44] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, "Automatic Query Reformulations for Text Retrieval in Software Engineering," in *Proceedings of the 2013 International Conference on Software Engineering*, Piscataway, NJ, USA, 2013, pp. 842–851.

[45] J. A. Hanley and B. J. McNeil, "The Meaning and Use of the Area Under a Receiver Operating Characteristic (ROC) Curve," *Radiology*, vol. 143, no. 1, pp. 29–36, Apr. 1982.

[46] M. A. Hearst and J. O. Pedersen, "Reexamining the Cluster Hypothesis: Scatter/Gather on Retrieval Results," in *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, New York, NY, USA, 1996, pp. 76–84.

[47] S. Henninger, "Using Iterative Refinement to Find Reusable Software," *IEEE Softw.*, vol. 11, no. 5, pp. 48–59, Sep. 1994.

[48] S. R. Henninger, "Locating Relevant Examples for Example-Based Software Design," Ph.D., University of Colorado at Boulder, United States -- Colorado, 1993.

[49] R. Hoffmann, J. Fogarty, and D. S. Weld, "Assieme: Finding and Leveraging Implicit References in a Web Search Interface for Programmers," in *Proceedings of the 20th*

annual ACM symposium on User interface software and technology, New York, NY, USA, 2007, pp. 13–22.

[50] R. Holmes, "Do Developers Search for Source Code Examples Using Multiple Facts?," in *ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2009. SUITE '09*, 2009, pp. 13–16.

[51] R. Holmes and G. C. Murphy, "Using Structural Context to Recommend Source Code Examples," in *27th International Conference on Software Engineering, 2005. ICSE 2005. Proceedings*, 2005, pp. 117–125.

[52] R. Holmes and R. J. Walker, "Task-Specific Source Code Dependency Investigation," in *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2007, pp. 100–107.

[53] M. Hucka and M. J. Graham, "Software search is not a science, even among scientists," *ArXiv160502265 Cs*, May 2016.

[54] O. Hummel, W. Janjic, and C. Atkinson, "Code Conjurer: Pulling Reusable Software out of Thin Air," *IEEE Softw*, vol. 25, no. 5, pp. 45–52, Sep. 2008.

[55] W. S. Humphrey, *Introduction to the Team Software Process(sm)*, 1 edition. Addison-Wesley Professional, 1999.

[56] D. Jiang, J. Pei, and H. Li, "Mining Search and Browse Logs for Web Search: A Survey," *ACM Trans. Intell. Syst. Technol.*, vol. 4, no. 4, p. 57:1–57:37, Oct. 2013.

[57] J. Johnson, *Designing with the Mind in Mind, Second Edition: Simple Guide to Understanding User Interface Design Guidelines*, 2 edition. Amsterdam ; Boston: Morgan Kaufmann, 2014.

[58] Jordan and T. M. Mitchell, "Machine Learning: Trends, Perspectives, and Prospects.," *Science*, vol. 349, no. 6245, pp. 255–260, 2015.

[59] C. K. G. and O. Andrew, "The Size and Growth Rate of the Internet," Center for Discrete Mathematics & Theoretical Computer Science, 1999.

[60] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting Working Code Examples," in *Proceedings of the 36th International Conference on Software Engineering*, New York, NY, USA, 2014, pp. 664–675.

[61] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks," *IEEE Trans. Softw. Eng.*, vol. 32, no. 12, pp. 971–987, Dec. 2006.

[62] W. Kruskal, T. S. Ferguson, J. W. Tukey, E. J. Gumbel, and F. J. Anscombe, "Discussion of the Papers of Messrs. Anscombe and Daniel," *Technometrics*, vol. 2, no. 2, p. 157, May 1960.

[63] S. Kullback and R. A. Leibler, "On Information and Sufficiency," *Ann. Math. Stat.*, vol. 22, no. 1, pp. 79–86, Mar. 1951.

[64] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a Name? A Study of Identifiers," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*, 2006, pp. 3–12.

[65] O. A. L. Lemos, S. K. Bajracharya, and J. Ossher, "CodeGenie: A Tool for Test-driven Source Code Search," in *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, New York, NY, USA, 2007, pp. 917–918.

[66] O. A. L. Lemos, A. C. de Paula, F. C. Zanichelli, and C. V. Lopes, "Thesaurus-Based Automatic Query Expansion for Interface-driven Code Search," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, New York, NY, USA, 2014, pp. 212–221.

[67] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, "Sourcerer: Mining and Searching Internet-Scale Software Repositories," *Data Min. Knowl. Discov.*, vol. 18, no. 2, pp. 300–336, Oct. 2008.

[68] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi, "UCI Source Code Data Sets." University of California, Irvine, Bren School of Information and Computer Sciences, 2010.

[69] C. V. Lopes, *Exercises in Programming Style*, 1 edition. Boca Raton: Chapman and Hall/CRC, 2014.

[70] G. F. Luger, P. Johnson, C. Stern, J. E. Newman, and R. Yeo, *Cognitive Science: The Science of Intelligent Systems*, 1 edition. San Diego: Academic Press, 1994.

[71] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid Mining: Helping to Navigate the API Jungle," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2005, pp. 48–61.

[72] N. Mangano, A. Baker, M. Dempsey, E. Navarro, and A. van der Hoek, "Software Design Sketching with Calico," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, New York, NY, USA, 2010, pp. 23–32.

[73] L. Martie and A. van der Hoek, "Sameness: An Experiment in Code Search," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR)*, 2015, pp. 76–87.

[74] L. Martie, T. Kwak, and A. van der Hoek, "Understanding the Impact of Support for Iteration on Code Search," in *Proceedings of the 2017 25th International Symposium on Foundations of Software Engineering*, 2017, p. (to appear).

[75] L. Martie, T. LaToza, and A. van der Hoek, "CodeExchange: Supporting Reformulation of Internet-Scale Code Queries in Context," presented at the Proceedings of the 30th International Conference on Automated Software Engineering, 2015, pp. 24–35.

[76] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding Relevant Functions and Their Usage," in *Proceedings of the 33rd International Conference on Software Engineering*, New York, NY, USA, 2011, pp. 111–120.

[77] A. Michail, "Data Mining Library Reuse Patterns Using Generalized Association Rules," in *Proceedings of the 22Nd International Conference on Software Engineering*, New York, NY, USA, 2000, pp. 167–176.

[78] M. L. Minsky, *Computation: Finite and Infinite Machines*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1967.

[79] G. Moody, *Rebel Code: Linux and the Open Source Revolution*. Basic Books, 2009.

[80] M. Mooty, A. Faulring, J. Stylos, and B. A. Myers, "Calcite: Completing Code Completion for Constructors Using Crowds," in *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2010, pp. 15–22.

[81] A. Newell and H. A. Simon, *Human Problem Solving*. Prentice-Hall, 1972.

[82] A. T. Nguyen *et al.*, "Graph-Based Pattern-Oriented, Context-Sensitive Source Code Completion," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 69–79.

[83] S. Oney and J. Brandt, "Codelets: Linking Interactive Documentation and Example Code in the Editor," in *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems*, New York, NY, USA, 2012, pp. 2697–2706.

[84] J. Ossher, S. Bajracharya, and C. Lopes, "Automated Dependency Resolution for Open Source Software," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, 2010, pp. 130–140.

[85] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web.," 11-Nov-1999. [Online]. Available: http://ilpubs.stanford.edu:8090/422/. [Accessed: 17-Dec-2012].

[86] W. Pender, D. Saddler, J. Shea, and D. Ward, *Cambridge 2 Unit Mathematics Year 11 Enhanced Version PDF Textbook*. Cambridge University Press, 2011.

[87] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Seahawk: Stack Overflow in the IDE," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 1295–1298.

[88] D. Poshyvanyk, A. Marcus, and Y. Dong, "JIRiSS - An Eclipse Plug-In for Source Code Exploration," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*, 2006, pp. 252–255.

[89] pubmeddev, "Home - PubMed - NCBI." [Online]. Available: https://www.ncbi.nlm.nih.gov/pubmed/. [Accessed: 16-Dec-2016].

[90] E. S. Raymond, *The Cathedral and the Bazaar*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1999.

[91] S. P. Reiss, "Semantics-Based Code Search," in *Proceedings of the 31st International Conference on Software Engineering*, Washington, DC, USA, 2009, pp. 243–253.

[92] M. P. Robillard, W. Coelho, and G. C. Murphy, "How Effective Developers Investigate Source Code: An Exploratory Study," *IEEE Trans. Softw. Eng.*, vol. 30, no. 12, pp. 889–903, Dec. 2004.

[93] S. R. Rosales and P. K. Mehrotra, "MES: An Expert System for Reusing Models of Transmission Equipment," 1988, pp. 109–113.

[94] D. E. Rose and D. Levinson, "Understanding User Goals in Web Search," in *Proceedings of the 13th International Conference on World Wide Web*, New York, NY, USA, 2004, pp. 13–19.

[95] H. Rubenstein and J. B. Goodenough, "Contextual Correlates of Synonymy," *Commun ACM*, vol. 8, no. 10, pp. 627–633, Oct. 1965.

[96] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3 edition. Upper Saddle River: Pearson, 2009.

[97] C. Sadowski, K. T. Stolee, and S. Elbaum, "How Developers Search for Code: A Case Study," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, New York, NY, USA, 2015, pp. 191–201.

[98] N. Sahavechaphan and K. Claypool, "XSnippet: Mining For Sample Code," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, New York, NY, USA, 2006, pp. 413–430.

[99] H. Sanchez and J. Whitehead, "Source Code Curation on StackOverflow: The Vesperin System," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, vol. 2, pp. 661–664.

[100] H. Sanchez, J. Whitehead, and M. Schäf, "Multistaging to understand: Distilling the essence of java code examples," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 2016, pp. 1–10.

[101] J. Sauro and J. S. Dumas, "Comparison of Three One-question, Post-task Usability Questionnaires," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2009, pp. 1599–1608.

[102] J. Sauro and J. R. Lewis, "Correlations Among Prototypical Usability Metrics: Evidence for the Construct of Usability," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2009, pp. 1609–1618.

[103] W. Scacchi, J. Feller, B. Fitzgerald, S. Hissam, and K. Lakhani, "Understanding Free/Open Source Software Development Processes," *Softw. Process Improv. Pract.*, vol. 11, no. 2, pp. 95–105, Mar. 2006.

[104] J. Scott, *BBS: The Documentary*. Bovine Ignition Systems, 2005.

[105] R. C. Seacord, S. A. Hissam, and K. C. Wallnau, "AGORA: A Search Engine for Software Components," *IEEE Internet Comput.*, vol. 2, no. 6, p. 62-, Nov. 1998.

[106] S. E. Sim, M. Agarwala, and M. Umarji, "A Controlled Experiment on the Process Used by Developers During Internet-Scale Code Search," in *Finding Source Code on the Web for Remix and Reuse*, S. E. Sim and R. E. Gallardo-Valencia, Eds. Springer New York, 2013, pp. 53–77.

[107] S. E. Sim, C. L. A. Clarke, and R. C. Holt, "Archetypal Source Code Searches: A Survey of Software Developers and Maintainers," in , *6th International Workshop on Program Comprehension, 1998. IWPC '98. Proceedings*, 1998, pp. 180–187.

[108] S. E. Sim, M. Umarji, S. Ratanotayanon, and C. V. Lopes, "How Well Do Search Engines Support Code Retrieval on the Web?," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 1, p. 4:1–4:25, Dec. 2011.

[109] H. A. Simon and A. Newell, "Human Problem Solving: The State of the Theory in 1970," *Am. Psychol.*, vol. 26, no. 2, pp. 145–159, 1971.

[110] R. Sindhgatta, "Using an Information Retrieval System to Retrieve Source Code Samples," in *Proceedings of the 28th International Conference on Software Engineering*, New York, NY, USA, 2006, pp. 905–908.

[111] J. Singer, T. C. Lethbridge, C. Gauthier, W. M. Gentleman, H. Johnson, and J. Sayyad, "What's so Great About `Grep'? Implications for Program Comprehension Tools," Tech. rep., National Research Council, Canada., 1997.

[112] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An Examination of Software Engineering Work Practices," in *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, Toronto, Ontario, Canada, 1997, p. 21–.

[113] L. B. Smith, "Similarity and Analogical Reasoning," S. Vosniadou and A. Ortony, Eds. New York, NY, USA: Cambridge University Press, 1989, pp. 146–178.

[114] M. D. Smucker and J. Allan, "Find-Similar: Similarity Browsing As a Search Tool," in *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, New York, NY, USA, 2006, pp. 461–468.

[115] J. Starke, C. Luce, and J. Sillito, "Working with Search Results," in *ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2009. SUITE '09*, 2009, pp. 53–56.

[116] K. T. Stolee, S. Elbaum, and D. Dobos, "Solving the Search for Source Code," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 3, p. 26:1–26:45, Jun. 2014.

[117] K. T. Stolee, S. Elbaum, and D. Dobos, "Solving the Search for Source Code," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 3, p. 26:1–26:45, Jun. 2014.

[118] A. P. Street and D. J. Street, *Combinatorics of Experimental Design*. New York, NY, USA: Oxford University Press, Inc., 1986.

[119] J. Stylos and B. A. Myers, "Mica: A Web-Search Tool for Finding API Components and Examples," in *IEEE Symposium on Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006*, 2006, pp. 195–202.

[120] A. G. Sutcliffe and N. Maiden, "Cognitive Studies in Software Engineering," in *Proceedings of 5th European Conference on Cognitive Ergonomics*, Urbino, Italy, 1990.

[121] R. N. Taylor and A. van der Hoek, "Software Design and Architecture The Once and Future Focus of Software Engineering," in *2007 Future of Software Engineering*, Washington, DC, USA, 2007, pp. 226–243.

[122] S. Thummalapenta and T. Xie, "Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, New York, NY, USA, 2007, pp. 204–213.

[123] S. Thummalapenta and T. Xie, "SpotWeb: Detecting Framework Hotspots via Mining Open Source Repositories on the Web," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, New York, NY, USA, 2008, pp. 109–112.

[124] L. Torvalds and D. Diamond, *Just for Fun: The Story of an Accidental Revolutionary*, Reprint edition. New York, NY: HarperBusiness, 2002.

[125] M. Umarji, S. E. Sim, and C. Lopes, "Archetypal Internet-Scale Source Code Searching," presented at the OSS, 2008, pp. 257–263.

[126] T. A. Vanderlei, F. A. Durão, A. C. Martins, V. C. Garcia, E. S. Almeida, and S. R. de L. Meira, "A Cooperative Classification Mechanism for Search and Retrieval Software Components," in *Proceedings of the 2007 ACM Symposium on Applied Computing*, New York, NY, USA, 2007, pp. 866–871.

[127] L. Vogel, *Eclipse IDE: Eclipse IDE based on Eclipse 4.2 and 4.3*. Lars Vogel, 2013.

[128] D. Wightman, Z. Ye, J. Brandt, and R. Vertegaal, "SnipMatch: Using Source Code Context to Enhance Snippet Retrieval and Parameterization," in *Proceedings of the 25th annual ACM symposium on User interface software and technology*, New York, NY, USA, 2012, pp. 219–228.

[129] K. Williams, J. Wu, and C. L. Giles, "SimSeerX: A Similar Document Search Engine," in *Proceedings of the 2014 ACM Symposium on Document Engineering*, New York, NY, USA, 2014, pp. 143–146.

[130] L. A. Williams and R. R. Kessler, "Experiments with Industry's 'Pair-Programming' Model in the Computer Science Classroom," *Comput. Sci. Educ.*, vol. 11, no. 1, pp. 7–20, Jan. 2001.

[131] Y. Ye and G. Fischer, "Supporting Reuse by Delivering Task-relevant and Personalized Information," in *Proceedings of the 24th International Conference on Software Engineering*, New York, NY, USA, 2002, pp. 513–523.

[132] Y. Ye, Y. Yamamoto, K. Nakakoji, Y. Nishinaka, and M. Asada, "Searching the Library and Asking the Peers: Learning to Use Java APIs on Demand," in *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, New York, NY, USA, 2007, pp. 41–50.

[133] A. Zagalsky, O. Barzilay, and A. Yehudai, "Example Overflow: Using Social Media for Code Recommendation," in *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*, 2012, pp. 38–42.

[134] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and Recommending API Usage Patterns," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Berlin, Heidelberg, 2009, pp. 318–343.

[135] "About WordNet - WordNet - About WordNet." [Online]. Available: https://wordnet.princeton.edu/. [Accessed: 19-Jan-2017].

[136] "Alpha, Significance Level of Test," in *Encyclopedia of Survey Research Methods*, 2455 Teller Road, Thousand Oaks California 91320 United States of America: Sage Publications, Inc., 2008.

[137] "Apache Lucene - Apache Lucene Core." [Online]. Available: http://lucene.apache.org/core/. [Accessed: 20-Jan-2017].

[138] "BBS Archives - Door Developer Kits." [Online]. Available: http://archives.thebbs.org/ra102a.htm. [Accessed: 02-Nov-2016].

[139] "Bitbucket | The Git Solution for Professional Teams." [Online]. Available: https://bitbucket.org/product. [Accessed: 02-Nov-2016].

[140] "Black Duck Open Hub Code Search." [Online]. Available: http://code.openhub.net/. [Accessed: 24-Apr-2017].

[141] *Compute! Magazine Issue 002*. 1980.

[142] "Eclipse Code Recommenders." [Online]. Available: http://www.eclipse.org/recommenders/. [Accessed: 16-Dec-2012].

[143] "Eclipse IDE Plug-in Development: Plug-ins, Features, Update Sites and IDE Extensions." [Online]. Available: http://www.vogella.com/tutorials/EclipsePlugin/article.html. [Accessed: 14-Nov-2016].

[144] "GitHub · Build Software Better, Together." [Online]. Available: https://github.com/. [Accessed: 16-Dec-2012].

[145] "Google." [Online]. Available: https://www.google.com/. [Accessed: 17-Dec-2012].

[146] "Google Code Search." [Online]. Available: http://www.google.com/codesearch. [Accessed: 16-Dec-2012].

[147] "Google Turning Its Lucrative Web Search Over to AI Machines," *Bloomberg.com*.

[148] "Hacker News." [Online]. Available: https://news.ycombinator.com/. [Accessed: 02-Feb-2017].

[149] "Home | Krugle - Software development Productivity." [Online]. Available: http://www.krugle.com/. [Accessed: 16-Dec-2012].

[150] "Institute for Software Research." [Online]. Available: http://isr.uci.edu/. [Accessed: 27-Aug-2016].

[151] "Introduction to A* Pathfinding," *Ray Wenderlich*. .

[152] *Introduction to Information Retrieval*, 1 edition. New York: Cambridge University Press, 2008.

[153] "Introduction to Information Retrieval." [Online]. Available: http://nlp.stanford.edu/IR-book/. [Accessed: 14-Dec-2016].

[154] "Java News/Tech/Discussion/etc. • /r/java," *reddit*. [Online]. Available: https://www.reddit.com/r/java/. [Accessed: 02-Feb-2017].

[155] "JavaNCSS - A Source Measurement Suite for Java." [Online]. Available: http://www.kclee.de/clemens/java/javancss/. [Accessed: 04-Apr-2017].

[156] "JavaRanch - A Friendly Place for Java Greenhorns." [Online]. Available: https://javaranch.com/. [Accessed: 02-Feb-2017].

[157] "JWI 2.4.0." [Online]. Available: http://projects.csail.mit.edu/jwi/. [Accessed: 22-May-2017].

[158] "Ohloh Code Search." [Online]. Available: http://code.ohloh.net/. [Accessed: 16-Dec-2012].

[159] "Polya, G.: Mathematics and Plausible Reasoning: Induction and Analogy in Mathematics. (Paperback)." [Online]. Available: http://press.princeton.edu/titles/1735.html. [Accessed: 24-Nov-2016].

[160] "ProjectLocker: Subversion Hosting, Git Hosting, SVN Hosting." [Online]. Available: http://projectlocker.com/. [Accessed: 12-Nov-2016].

[161] "Search Engine Market Share." [Online]. Available: https://www.netmarketshare.com/search-engine-market-share.aspx?qprid=4&qpcustomd=0. [Accessed: 27-Aug-2016].

[162] "SourceForge - Download, Develop and Publish Free Open Source Software." [Online]. Available: http://sourceforge.net/. [Accessed: 16-Dec-2012].

[163] "Stack Overflow." [Online]. Available: http://stackoverflow.com/. [Accessed: 16-Dec-2012].

[164] "SwitchYard - JBoss Community." [Online]. Available: http://switchyard.jboss.org/. [Accessed: 21-Mar-2017].

[165] "TFIDFSimilarity (Lucene 4.0.0 API)." [Online]. Available: https://lucene.apache.org/core/4_0_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html. [Accessed: 02-Feb-2017].

[166] "The SAGE Encyclopedia of Qualitative Research Methods | SAGE Publications Inc." [Online]. Available: https://us.sagepub.com/en-us/nam/the-sage-encyclopedia-of-qualitative-research-methods/book229805. [Accessed: 19-Mar-2017].

[167] "The SourceForge Story - Datamation," 04-Aug-2011. [Online]. Available: https://web.archive.org/web/20110804024950/http://itmanagement.earthweb.com/cnews/article.php/3705731. [Accessed: 15-Oct-2016].

[168] “Welcome [Savannah].” [Online]. Available: http://savannah.gnu.org/. [Accessed: 12-Nov-2016].