Title
Source Code Curation Tooling for the Code Forager

Permalink
https://escholarship.org/uc/item/4p05m519

Author
Sanchez, Huascar Antonio

Publication Date
2015

License
CC BY-NC-SA 4.0

Peer reviewed|Thesis/dissertation
UNIVERSITY OF CALIFORNIA
SANTA CRUZ

SOURCE CODE CURATION TOOLING FOR THE CODE FORAGER

A dissertation submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Huascar Antonio Sanchez

December 2015

The Dissertation of Huascar Antonio Sanchez
is approved:

Prof. Jim Whitehead, Chair

Prof. Luca de Alfaro

Prof. Cormac Flanagan

Tyrus Miller
Vice Provost and Dean of Graduate Studies
Table of Contents

List of Figures vi
List of Tables ix
Abstract x
Dedication xi
Acknowledgments xii

1 Introduction 1
  1.1 Motivation & Problem ................................. 1
  1.2 Curation equals Quality minus Junk .................... 4
  1.3 Source Code Curation ................................ 5
  1.4 Thesis Statement ...................................... 5
  1.5 Research Questions .................................... 6
  1.6 Definition of Terms .................................... 8
    1.6.1 Code Foraging ..................................... 8
    1.6.2 Source Code Curation .............................. 9
    1.6.3 Program Understanding ............................. 9
    1.6.4 Refactoring ....................................... 11
  1.7 Contributions .......................................... 11
    1.7.1 Curating code with the Vesperin System .......... 12
    1.7.2 Codepacking ...................................... 12
    1.7.3 Multistaging to Understand ....................... 13
  1.8 Outline .................................................. 14

2 Related Work 16
  2.1 Using Code Examples .................................. 17
  2.2 Inspecting Java Programs .............................. 19
### 3 The Vesperin System

- **3.1 Vesperin on StackOverflow** .................................................. 22
- **3.2 Vesperin Use Model** ................................................................. 23
  - 3.2.1 Scratch Space and Drafts ..................................................... 24
  - 3.2.2 Other Elements ................................................................. 25
  - 3.2.3 Vesperin Actions ............................................................... 25
  - 3.2.4 D.R.Y.I.N.G Source Code Curation ...................................... 27
- **3.3 Vesperin's Architecture** ..................................................... 27
  - 3.3.1 Violette .............................................................................. 28
  - 3.3.2 Kiwi ................................................................................... 33
- **3.4 Putting It All Together** ....................................................... 36
- **3.5 Conclusion** ........................................................................... 47

### 4 Evaluating Vesperin

- **4.1 Method** ................................................................................. 50
  - 4.1.1 Participants and Design ...................................................... 50
  - 4.1.2 Used materials .................................................................... 51
  - 4.1.3 Procedure and manipulations ............................................. 56
- **4.2 Pilot Sessions** ......................................................................... 58
- **4.3 Participant Background** ....................................................... 62
- **4.4 Results** ................................................................................ 63
  - 4.4.1 How will developers use these source code curation interfaces? ........................................... 63
  - 4.4.2 Will the set of provided facilities be sufficient for curating code examples? .................. 67
  - 4.4.3 Will developers be able to better understand the unfamiliar code examples via curation? ............... 68
- **4.5 Conclusion** ........................................................................... 70

### 5 Codepacking

- **5.1 Our approach** ......................................................................... 72
- **5.2 Codepacking session** ............................................................. 74
  - 5.2.1 Setup: Incremental parsing ............................................... 75
  - 5.2.2 Resolving missing dependencies ...................................... 76
  - 5.2.3 Reusing information from signature and its top scoring construct .................................. 80
- **5.3 Codepacking** ........................................................................ 82
  - 5.3.1 The PackingSpace ............................................................... 83
  - 5.3.2 The Explanatory Power of Results .................................... 85
  - 5.3.3 Mapping Signatures to Java constructs ............................ 85
  - 5.3.4 Execution Plans ................................................................. 86
  - 5.3.5 Codepacking Architecture ................................................. 87
- **5.4 Evaluation** ............................................................................ 88
  - 5.4.1 Sample population ............................................................. 88
  - 5.4.2 Evaluation approach ......................................................... 90
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Vesperin’s Use Model and the connection of a Vesperin page to a StackOverflow page. A Vesperin page is pictured as a layer placed on top of a Q&amp;A page containing Java code examples. On that layer, code foragers can curate the enclosed examples.</td>
</tr>
<tr>
<td>3.2</td>
<td>Vesperin’s architecture. Dimmed elements (e.g., codepacking) in the architecture are features that we’ll unpack in Chapter 5 and Chapter 6 respectively.</td>
</tr>
<tr>
<td>3.3</td>
<td>Violette’s User Interface. This interface is designed to make it easy for code foragers to examine (via curation) the online source code they encounter. The stage visual component, along with its functionality, is described in Chapter 6.</td>
</tr>
<tr>
<td>3.4</td>
<td>Kiwi’s Rename Member Command. This command contains information of what member to rename, the new name, the offset range or location of target variable, and the actual source code. The packing variable is related to Codepacking, which is described in Chapter 5. The rest of commands are described in <a href="http://www.vesperin.com/kiwi/help">http://www.vesperin.com/kiwi/help</a>.</td>
</tr>
<tr>
<td>3.5</td>
<td>Searching for a code example that shows how to parallelize the QuickSort algorithm.</td>
</tr>
<tr>
<td>3.6</td>
<td>Selecting one of the many results generated by the search engine.</td>
</tr>
<tr>
<td>3.7</td>
<td>Hovering over the code block (with the on-screen pointer) of a StackOverflow Q&amp;A’s accepted answer to show the Edit button.</td>
</tr>
<tr>
<td>3.8</td>
<td>The code forager clicks the Edit button. A new scratch space is created.</td>
</tr>
<tr>
<td>3.9</td>
<td>The code forager clicks the Cleanup button. This function works as described in Section 3.3.1.</td>
</tr>
<tr>
<td>3.10</td>
<td>The code forager has marked a new draft and has found that the source code lacks some external references.</td>
</tr>
<tr>
<td>3.11</td>
<td>The code forager has added the missing references and then re-inspected the source code. Violette has come back with new errors.</td>
</tr>
<tr>
<td>3.12</td>
<td>The code forager implements the missing methods and then re-inspects the source code. Violette finds no more errors.</td>
</tr>
<tr>
<td>3.13</td>
<td>Chunking a code fragment into a method.</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>3.14</td>
<td>Renaming <code>selectArrayPivot</code> to <code>partition</code></td>
</tr>
<tr>
<td>3.15</td>
<td>Chunking the next code fragment in the <code>quicksort</code> method.</td>
</tr>
<tr>
<td>3.16</td>
<td>Displaying the Edit Tracker or Draft’s History.</td>
</tr>
<tr>
<td>3.17</td>
<td>The code forager ends the curation session by downloading the curated source code.</td>
</tr>
<tr>
<td>4.1</td>
<td>Search Cue, a StackOverflow Search Application.</td>
</tr>
<tr>
<td>4.2</td>
<td>Background questionnaire given to participants at the beginning of the study.</td>
</tr>
<tr>
<td>4.3</td>
<td>Monitoring participants' interactions with <code>Vesperin</code> via an interaction event model.</td>
</tr>
<tr>
<td>4.4</td>
<td>Summary of participants' background information.</td>
</tr>
<tr>
<td>4.5</td>
<td>All participants' editing activity in a 20 minutes time window. Each participant is represented by a colored area.</td>
</tr>
<tr>
<td>4.6</td>
<td>Distribution of participants' expectations before (Pretest), and experiences after (Posttest) using <code>Vesperin</code>. Horizontal axes: 5-point Likert scale, ranging from strongly disagree (“- -”) to strongly agree (“++”). Vertical axes: number of participants.</td>
</tr>
<tr>
<td>4.7</td>
<td>Distribution of participants' expectations before (Pretest), and experiences after (Posttest) using <code>Vesperin</code>. Outlier values are included. Horizontal axes: 5-point Likert scale, ranging from strongly disagree (“- -”) to strongly agree (“++”). Vertical axes: number of participants.</td>
</tr>
<tr>
<td>5.1</td>
<td>Incomplete code example.</td>
</tr>
<tr>
<td>5.2</td>
<td>Simplified Codepacking architecture.</td>
</tr>
<tr>
<td>5.3</td>
<td>Java code with an ambiguous reference.</td>
</tr>
<tr>
<td>5.4</td>
<td>Signature creation and usage.</td>
</tr>
<tr>
<td>5.5</td>
<td>Results evaluation (i.e., explanations of a signature).</td>
</tr>
<tr>
<td>5.6</td>
<td>Generating an execution plan.</td>
</tr>
<tr>
<td>5.7</td>
<td>Java code after resolving an ambiguous reference.</td>
</tr>
<tr>
<td>5.8</td>
<td>Codepacking architecture.</td>
</tr>
<tr>
<td>6.1</td>
<td>Searching for a code example that shows how to find the smallest number in an array without using sorting.</td>
</tr>
<tr>
<td>6.2</td>
<td>Generated results by Google search engine. Programmer selects first one.</td>
</tr>
<tr>
<td>6.3</td>
<td>Finds the smallest number in an array without using sorting.</td>
</tr>
<tr>
<td>6.4</td>
<td>Generated code stages (green buttons).</td>
</tr>
<tr>
<td>6.5</td>
<td>Multistaging to Understand session. Unfolded code blocks are highlighted.</td>
</tr>
<tr>
<td>6.6</td>
<td>Pseudocode for <code>MethodSlicing</code>.</td>
</tr>
<tr>
<td>6.7</td>
<td>One application of <code>MethodSlicing</code> against the <code>SmallestNum</code> code example.</td>
</tr>
<tr>
<td>6.8</td>
<td>Pseudocode for <code>GetAllAstNodeBindings</code>.</td>
</tr>
<tr>
<td>6.9</td>
<td>Pseudocode for <code>ReconstructSourceCode</code> subroutine.</td>
</tr>
</tbody>
</table>
6.10 Pseudocode for updated \texttt{ReconstructSourceCode}. This subroutine returns a tuple comprising the reconstructed source code and the code elements to hide.

6.11 Reduced large code stages. Knapsack capacity is 15 (LOC). Uninformative elements are hidden using code folding. The \ldots{} symbol represents hidden areas.


7.1 Comprehension questions for task 1.
7.2 Comprehension questions for task 2.
7.3 Comprehension questions for task 3.
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Supported HTTP Endpoints by Kiwi. All of the endpoints are preceded by vesperin.com host.</td>
<td>34</td>
</tr>
<tr>
<td>4.1</td>
<td>Pretest questionnaire.</td>
<td>54</td>
</tr>
<tr>
<td>4.2</td>
<td>Posttest questionnaire.</td>
<td>55</td>
</tr>
<tr>
<td>4.3</td>
<td>Single program understanding task</td>
<td>57</td>
</tr>
<tr>
<td>4.4</td>
<td>Interactions with Vesperin</td>
<td>64</td>
</tr>
<tr>
<td>4.5</td>
<td>Participants' top 3 features. Each column represents a participant. The 1's, 2's, and 3's represent a participant’s first, second, and third choice respectively.</td>
<td>67</td>
</tr>
<tr>
<td>5.1</td>
<td>PackingSpace numbers</td>
<td>88</td>
</tr>
<tr>
<td>5.2</td>
<td>Evaluation sample</td>
<td>89</td>
</tr>
<tr>
<td>5.3</td>
<td>Recovery of body declarations evaluation</td>
<td>91</td>
</tr>
<tr>
<td>5.4</td>
<td>Resolution of missing dependencies evaluation</td>
<td>91</td>
</tr>
<tr>
<td>7.1</td>
<td>Crossed Factorial Design</td>
<td>117</td>
</tr>
<tr>
<td>7.2</td>
<td>Closed-ended program comprehension questions for evaluating abstractions</td>
<td>118</td>
</tr>
<tr>
<td>7.3</td>
<td>Open-ended program comprehension questions for evaluating abstractions</td>
<td>119</td>
</tr>
<tr>
<td>7.4</td>
<td>Guidelines for the Multistaging to Understand experiment</td>
<td>125</td>
</tr>
<tr>
<td>7.5</td>
<td>Guidelines for the Reading to Understand experiment</td>
<td>126</td>
</tr>
<tr>
<td>7.6</td>
<td>Average Response Accuracy</td>
<td>127</td>
</tr>
<tr>
<td>7.7</td>
<td>Rejection of $H_0,accuracy,X,Y$</td>
<td>128</td>
</tr>
<tr>
<td>7.8</td>
<td>Average Reviewing Time</td>
<td>129</td>
</tr>
</tbody>
</table>
Abstract

Source Code Curation Tooling for the Code Forager

by

Huascar Antonio Sanchez

The Web has changed the dynamics of programming. We are in an era where reusing code from the Web is frequently the norm, not the exception. In this era, programmers use question and answer (Q&A) sites like StackOverflow to pose coding questions and receive coding answers. Programmers using these sites often engage in a complex code foraging process of understanding and adapting the code snippets they encounter to determine their fitness for use. While search still dominates modern code retrieval, search alone offers little support for validating search results for fitness of use. This is, in large part, due to the inherently questionable quality of online source code. Most online source code is not guaranteed to be good, to work properly, or to be trustworthy. This dissertation focuses on this challenge by introducing Source Code Curation, along with a set of tools that implements it. Source Code Curation is a blend of filtering, refinement, and validation activities. Source Code Curation can help programmers determine what source code is more likely to be useful, and what’s not. Specifically, it can help them both address the inherently questionable quality of online source code upfront, and complete their source code understanding tasks quickly and accurately.
For Claudia, Isabella, & Camilla
Acknowledgments

First, I would like to thank my wife Claudia for her love, understanding, and support in the work I do and the time it takes me away from her and from our children. Second, I would also like to thank my parents and my grandmother for instilling in me my sense of intellectual curiosity and exploration. My children–Isabella and Camilla–who may not fully understand what motivated me to pursue higher education, but I hope that someday when they read this dissertation, they would understand what drove their father to pursue a Ph.D. in Computer Science. Last, but not the least, I would like to thank my mentor and friend, Jim Whitehead, who has influenced, supported, inspired, and collaborated with me throughout my years at UC Santa Cruz. Jim, you have inspired me and taught me so much over these years. Thank you. This dissertation is dedicated to all of you!
Chapter 1

Introduction

This dissertation explores the notion of Source Code Curation in the context of foraging for useful source on the Web to help solve a programming task, and the tools that implement this notion. This chapter starts to unpack some of these elements as well as provide some background information before diving into the core work.

1.1 Motivation & Problem

Over the past few years, the Web has changed the dynamics of programming. We are in an era where copying and pasting code from the Web is frequently the norm, not the exception. In this era, programmers use software question and answer (Q&A) sites like StackOverflow to pose detailed coding questions and receive detailed coding answers. Programmers using these sites engage in a complex code foraging process of understanding and adapting the source code (e.g., code examples) they encounter.

Such a process is mostly informal and ad-hoc, and ultimately can be a drain
on one of the most precious resources in software development: time. In fact, just like junkyard scavenging, this process is laborious and challenging. It is laborious, as it often involves multiple rounds of specific steps (e.g., searching, browsing, screening, filtering) to find the best source code for the required task (Marchionini, 2006). It is challenging, as it frequently involves dealing with source code with inherently questionable quality. Most source code on the Web is not guaranteed to be good, to work properly, or to be trustworthy (Gysin and Kuhn, 2010).

Existing research on search and adaptation of code examples (Barzilay, 2011; Brandt et al., 2010; Ponzanelli et al., 2013; Wightman et al., 2012) focuses more on addressing the laborious aspect of this process and less on the challenging aspect. This dissertation focuses more on the challenging aspect of code foraging and less on the laborious aspect, as the questionable quality of online source code has strong implications on the effectiveness of code foraging, and thus affects its laboriousness. In fact, despite the benefits of search technology reported by prior research, substantial time and effort can still be required to consume online source code. This is, in large part, due to its inherently questionable quality (Sanchez and Whitehead, 2015).

The perception of the quality of a snippet of online source code is highly dependent on its fitness for use, where fitness for use is the degree to which a snippet is suitable for an application or purpose. This fitness for use is relative to the specific task that a programmer has at hand. In other words, it is a perceptual and somewhat subjective attribute (Gryna and Juran, 2001). In this dissertation, when we write quality we mean fitness for use.
We characterize quality by a series of quality dimensions which represent a set of desirable characteristics for an information resource (Dandashi 2002; ISO/IEC 2001; Kan 2002). Examples of these dimensions, and the ones we consider relevant to code foraging, include:

- **Accuracy.** Is the found online source code syntactically correct? Incorrect source code has drastically reduced usefulness for the code foragers.

- **Adaptability.** Can the found online source be modified with ease? Not adaptable source code can introduce significant difficulty for code foragers attempting to retool them into their own code.

- **Completeness.** Are all the required code elements available to parse the source code? Does the found source code need to be cleansed from errors? Are there any omissions (required capabilities) from the source code? In some cases, missing capabilities can be irrelevant, but when the missing capability is critical to a specific programming need, then completeness can be an issue.

- **Understandability.** Is the meaning of the online source code clear to a user? Is the meaning of the source code ambiguous?

Given these dimensions, we can now characterize what we mean by a source code with a questionable quality. We characterize a source code with a questionable quality as a source code that lacks some of the above characteristics.

Uncertainty over the quality of online source code can have negative effects on effective code foraging, as it can directly affect programmers’ decision making process.
during a code adaption (Mackay, 1991). For instance, having to contend with poor quality source code can ultimately lead to poor decision making during code adaptation, and thus impact productivity. When making code modification decisions or interpreting code abstractions, programmers must be able to assess the quality of the source code they are adapting.

Consequently, if we are to be more effective at foraging for useful online source code, then we must also be ready to address upfront the challenges posed by inherently questionable code quality. It’s until then we will be able to make sense of them properly and then make informed decisions on whether to reuse them.

This dissertation focuses on such a proposition. It does it by tapping into an area that is well known for addressing issues of this kind. This area is called curation.

1.2 Curation equals Quality minus Junk

When people think of curation, they think of examining content to separate the meaningful from the meaningless, assembling them into formations of understanding that represent their own original ideas, and transforming them to improve quality, discovery, and reusability (Krysa, 2006). To that effect, curation is not much different than filtering, refinement, and validation (Stonebraker et al, 2013); all coalesced into an act that assures that the content at hand is high quality before use.

Curation represents a nice blend of three activities: filtering, refinement, and validation. As such, curation could help programmers during their code foraging activi-
ties. It could help them determine what source code is more likely to be useful, what’s junk, and what’s not. With that in mind, it seems logical to use curation to address the challenging nature of code foraging, as it can be applied to online source code. When curation is applied to source code it then becomes Source Code Curation.

1.3 Source Code Curation

Inspired by Ball (2009); Bourne and McEntyre (2006); Krysa (2006); Stone-braker et al. (2013), we introduce this notion to cover the act of discovering a source code snippet of interest, cleaning and transforming it (i.e., refining it), and then presenting it in a meaningful and organized way. Source Code Curation’s goal is to improve the quality of online source code before reuse, validate its fitness for use, and assist with code understanding.

Given that the questionable quality of online source code has strong implications on effective code foraging, and that the quality of online source code is described by a series of desirable characteristics, then we can use Source Code Curation to improve these qualities.

1.4 Thesis Statement

Specifically, our thesis is that by building tools that support the notion of Source Code Curation, programmers will be able (1) to deal with the inherently questionable quality of online source code upfront, (2) to better understand this source code,
and (3) to validate its fitness for use.

1.5 Research Questions

The question then becomes: how can this notion be implemented? One key insight that helps us answer this question is that programmers are curious. They want to exercise their problem solving abilities almost immediately (Brandt et al., 2009). As a result, any solutions for the challenging nature of code foraging should not impede programmers’ natural impulses, especially since these impulses are an integral part of their learning experience (Kuhn and DeLine, 2012).

Based on the above observations, this dissertation explores the development of Source Code Curation tools. These tools enable programmers to cope with online source code’s challenging nature upfront, before consumption, and to gain quicker and more accurate understanding of curated source code.

An essential part of knowing what kind of Source Code Curation tools could help programmers deal with the inherently questionable quality of online source code is knowing precisely what programmer behaviors, needs, and motivations are during code foraging sessions. We will examine these matters via the exploration of the following guiding questions:

RQ1: How should interfaces for curating source code work?

RQ2: What are the curation strategies used by programmers when using Source Code Curation tools? Are the provided facilities sufficient for dealing with online code’s
challenging nature?

RQ3: What are the effects of using Source Code Curation tools on code understanding accuracy and task efficiency? Will code foragers be able to comprehend some of the tradeoffs involved in using online code more efficiently and more accurately?

To answer these questions, we embark on a set of qualitative and quantitative studies involving professional software engineers who practice code foraging. This includes a thorough review of published research and recommendations for search-driven development. Particularly, we focus on reviewing research that embraces the latent opportunity of existing code examples on the Web. We report on studies conducted locally at a software company in the Bay Area, and also remotely via the Upwork crowdsourcing service.

Over a period of several months, we visited this software company sporadically, and also interacted with our recruited professionals at Upwork on an on-demand basis. We studied these professional software engineers through observation techniques, task analysis, and other feedback methodologies such as surveys, and focus groups. Particularly, we were interested in discovering what strategies these engineers use when engaging in code foraging activities, and if the used strategies were sufficient. By clustering these observed strategies, we could determine different usage patterns of varying difficulty that could suggest directions for the type of Source Code Curation tools currently needed.

We are excited by what we found. The act of code foraging has been, and is, greatly changing; yet, the means and methods programmers employ to forage online
source code are still informal and ad-hoc. For instance, upon finding source code online, engineers often begin adapting code before they fully understood it. Our findings are consistent with Brandt’s study of opportunistic programmers (Brandt et al., 2009), but provide additional insights into the challenges faced by engineers when applying the learn-by-doing approach. As one of the software engineers explains “To be honest, retooling online source code could be painful. Especially, if you didn’t comprehend in advance some of the tradeoffs involved in using such an online source code.”

1.6 Definition of Terms

The following definitions are provided to ensure uniformity and understanding of these terms throughout the dissertation.

1.6.1 Code Foraging

We define code foraging as the act of locating useful source code on the Web to help solve a software development task. We call programmers who practice this type of reuse code foragers. Code foraging is a type of reuse that is frequently practiced by programmers during their code examples search and adaptation (Brandt et al., 2010, 2009), and example embedding (Barzilay, 2011).

Practicing this type of reuse is different than practicing other forms of reuse (Krueger, 1992). It requires one to coalesce a set of distinctive abilities when foraging for useful online source code. For instance, it requires skillfulness to find the right source code for a task, technical knowledge to quickly understand the found code, and insight
to know how to adapt that found code into an existing codebase. If applied properly, code foraging can improve programming productivity.

### 1.6.2 Source Code Curation

Source Code Curation covers the act of discovering a source code snippet of interest, cleaning and transforming it, and then presenting it in a meaningful and organized way. This umbrella term aims at forging new relationships among activities such as code recommendation (Ponzanelli et al., 2014), code documentation generation (Parnin et al., 2012; Subramanian et al., 2014), and code retrieval (Wightman et al., 2012). They all play a role in curation: from recommending the best ranked source code candidates to allowing frequent documentation and code edits to keep source code fresh and relevant.

### 1.6.3 Program Understanding

Program understanding is an important software engineering activity. It is required when programmers maintain, reuse, migrate, re-engineer, or enhance a piece of software (Brooks, 1983; Lakhotia, 1993; Littman et al., 1987; von Mayrhauser and Vans, 1993). For example, during corrective maintenance, programmers are involved in activities undertaken to detect, isolate, and fix a bug, all without breaking anything. To facilitate success, programmers must understand the software well enough in order to analyze it.

The current literature on program understanding assumes that comprehension
is done top-down, bottom-up, or some combination of the two, and that a mental representation of the program is built at various abstraction levels (e.g., code, task) as as result of these strategies (von Mayrhauser et al. 1995).

As proposed by [Brooks (1983)], Top-down Program Comprehension is about reconstructing knowledge about the program domain and then mapping this knowledge to the source code. This form of comprehension occurs when the source code is familiar. The central strategy employed in top-down program comprehension is hypothesis formation and evaluation. Programmers start with a general hypothesis about the nature of the program, and then formulate subsidiary hypotheses to refine the general hypothesis in a depth-first manner. Hypotheses are verified or rejected against the source code by looking at beacons. Beacons are features that allow code elements to be bound to the hypothesis. For example, the presence of a partition method in some source code may confirm the hypothesis that the program implements the Quicksort algorithm.

As proposed by [Pennington (1987)], Bottom-up Program Comprehension is about understanding source code and the mapping of code fragments onto domain concepts. This form of comprehension occurs when the source code is unfamiliar. It requires programmers to start reading small code fragments and then mentally grouping (chunking) them into large aggregates. The purpose of these large aggregates is built from their parts. This grouping process will continue until the goal of the program is revealed. The central strategy employed in bottom-up program comprehension is chunking. Chunking is used to build two mental models: a program model and a situation model. The program model refers to a control flow abstraction of the program. The situation model
uses the program model to create data-flow and functional abstractions.

A hybrid program comprehension strategy reflects a combination of the previous two program comprehension strategies. Specifically, when using a hybrid program comprehension strategy, program understanding is formed by switching between top-down and bottom-up program comprehension strategies as needed (Pennington 1987).

1.6.4 Refactoring

Refactoring is about evolution (Opdyke 1992). It is about improving the design of source code; making it not only more reusable but also flexible to subsequent semantic changes. It is not about changing or improving software from a functional point of view.

Refactoring is seen as a representation of the processes most programmers instinctively do to restructure their code (Fowler 1999). It is a way of describing any behaviour-preserving program modification in terms of atomic procedures of renaming, moving, deleting, and introducing new code (Mens and Tourwé 2004). The atomicity of these procedures makes refactoring stand out among other program evolution and code restructuring approaches. Particularly, because it makes manual implementation practical and also facilitates the means for proving correctness of applied restructurings.

1.7 Contributions

This thesis makes three main contributions. With begin with the introduction of Source Code Curation, along with the system that implements this notion. Next, we introduce a technique that attempts to fill the gaps found in incomplete source code
found by code foragers. We end by introducing a technique that uses the divide and conquer principle to steer understanding of online source code found by code foragers.

1.7.1 Curating code with the Vesperin System

The first contribution is the introduction of the notion of Source Code Curation, along with the Vesperin system, which implements this notion. Vesperin is suited to the unique challenges of curating online source code written in Java. In particular, most online source code is not guaranteed to be good, work properly, or be trustworthy. This is a problem for code foragers attempting to write new code by retooling code they find online. The Vesperin system helps in multiple ways. First, it validates whether the found code is syntactically correct. Second, it allows code foragers to explore code modification ideas in the original Web page (in-place) to steer code understanding. By investing in such a process, code foragers can intuitively experiment with coding ideas hands-on, before consumption outside the Web browser, and thus verify their hypotheses about curated code in situ. At a high level, Vesperin consists of two main components:

1. A Chrome Extension for allowing code foragers to actually edit, change, and validate online source code in-place. This includes the ability to clip a snippet of curated source code and save it in a comfortable reading view for examining now or later on one’s computer.

2. A RESTful service for managing source code curation and parsing operations.
1.7.2 Codepacking

The second contribution is the introduction of Codepacking, a technique implemented in the Vesperin system. Codepacking is responsible for filling the gaps in incomplete online source code found by code foragers, whenever possible. Most online source code brought in to a code foragers’ environment are code fragments. By definition, code fragments are incomplete and often ambiguous. These fragments might not be embedded in methods, classes, or both (Subramanian et al., 2014). Moreover, they may also declare fields whose package reference is not included. As described in Section 1.6.2, this incompleteness is also a problem for code foragers who attempt to retool any found code into their own code. Codepacking helps by packing each online source code with missing code elements in order to give a good start with code foragers’ curation activities. In other words, it boosts curation even further by helping out with completeness.

1.7.3 Multistaging to Understand

The third contribution is the introduction of the Multistaging to Understand comprehension technique, along with Vesperin’s multistager component which implements it. The purpose of the multistager is to improve online source code understanding accuracy and task efficiency during code foraging tasks. It does this by automatically dividing a found online source into a series of discrete chunks of behavior, which can be accessed non-sequentially. Each chunk of behavior represents a cohesive subset of functionality from the found online source code. The main goal of the multistager is to
speed up the curation process. We find that the Multistaging to Understand technique is a valuable tool for understanding code examples where most of the code is non-localized, but has only minor benefits when code is partially or fully delocalized. The technique provides consistent speed improvements, irrespective of delocalization.

Multistaging shares similarities with code reading by stepwise abstraction. Code reading by stepwise abstraction \cite{linger1979} calls for identifying prime subprograms in the software, determines their function and uses them to determine a function for the entire program. The effectiveness and efficiency of this technique for structured programs was validated by \cite{basili1987}, and validated for object-oriented programs by \cite{dunsmore2003}. In contrast, we focus on algorithmically generating all the prime subsets of behavior based on the source code’s content.

All of the above contributions are motivated by a vision of where code foraging practices involving Source Code Curation can take us. The time is ripe to build Source Code Curation systems and run experiments to lay the groundwork for a less ad-hoc and more thoughtful code foraging. These systems will increase programming productivity, all by amplifying programmers’ ability to both select and combine the right snippets to craft solutions that will meet their needs.

1.8 Outline

This dissertation will progress as follows. In Chapter 2, we talk about related work, positioning this work in the greater context of search-driven development.
Chapter 3 introduces Vesperin, a system for curating Java code examples on StackOverflow. Chapter 4 presents some experiments to evaluate Vesperin’s usefulness. Chapter 5 presents Codepacking and its evaluation. Chapter 6 discusses the development of a technique for revealing segments of code in an easy to understand sequence, and how it can be integrated into the code foraging process. Chapter 7 presents an experimental validation of this comprehension technique. Chapter 8 coalesces our experience into a high-level discussion of the current state-of-the-art of search-driven development, outlining the challenges we face now, and suggesting directions for future work. Chapter 9 closes the thesis with a few brief concluding remarks.
Chapter 2

Related Work

The Web provides a rich source of online source code, shared by millions of people in code repositories, blogs, and technical forums. As an amazing treasure trove of code, the Web enables programmers to build applications opportunistically by iteratively searching for, modifying, and combining fragments of code (code examples) (Sim and Gallardo-Valencia 2013). Practicing this form of reuse is ubiquitous (Kim et al. 2004) and has been found to be intuitive to many programmers, novices and experts alike (Lahtinen et al. 2005). Nonetheless, it can be a time-consuming activity. Our work aids programmers in this matter, as described throughout this dissertation.

Our work builds on two primary areas of prior work; tools for using code examples, and tools for aiding with their inspection.
2.1 Using Code Examples

In various ways, research has been done to embrace the latent opportunity of existing code examples. Numerous tools have been developed to support a range of specific tasks, such as code authoring, understanding, integration, and documentation linking. For instance, some of these tools enable programmers to integrate pre-authored-and-refined (curated) code examples into development environments to assist programmers when authoring new code (Hartmann et al., 2010; Wightman et al., 2012). Others enable programmers to directly search the Web for code examples from within a development environment (Bhardwaj et al. 2011; Brandt et al. 2010, 2009. Barzilay (2011) takes these ideas further, proposing a comprehensive approach, called Example Embedding, which focuses on embedding example-related concerns inside the development process, tools, practices, etc. Yet, even with these advances in searching through online scraps of code (Barzilay 2011; Brandt et al. 2010; Ponzanelli et al. 2013; Wightman et al. 2012), programmers still struggle with consumption of the code results they find.

Programmers’ struggles are caused by online source code’s questionable quality (Sanchez and Whitehead 2015). Consequently, if we are to be more effective in locating useful scraps of code, and thus minimize these struggles, then we must still also address their questionable quality upfront. We address these challenges in a single coherent source code curation system, called Vesperin (Chapter 3).

Curating code examples requires access to the code example’s syntax and static semantics, which is difficult to obtain because code examples tend to be incomplete and
ambiguous (Dagenais and Robillard, 2012). The use of partial program analysis to aid with the generation of fully-resolved ASTs from partial programs (e.g., code examples) is not new. Dagenais and Hendren (2008) use fuzzy type inference on a partial program in order to fake its declarative completeness. A similar technique is PARSEWeb (Thum-malapenta and Xie, 2007). PARSEWeb does static analysis on partial programs by looking at input-output types, and then suggests method invocation sequences containing the receiver object of interest. Our Codepacking technique (described in Chapter 5) is a special case of these ideas. Consequently, Codepacking can be easily extended with the above techniques to address cases where it could not locate specific dependencies in some code example.

Additionally, there are also tools that try to bridge the gap between the Web browser and the editor to help re-establish context and improve understanding of code examples (Ginosar et al., 2013; Hartmann et al., 2011). Others author multistage code examples using either direct editing and editable code histories (Ginosar et al., 2013), record and replay of code-based tutorials (Kojouharov et al., 2004), or annotated code tours to highlight important code locations (Oezbek and Prechelt, 2007). While effective, these tools typically assume that a complete and correct set of code stages is not available (Ginosar et al., 2013). Consequently, they rely on humans to massage an existing source code (via direct editing and editable histories) and then turn it into a multistage code example. Our work is quite different (Chapter 6). We assume that code stages are available and they can be extracted based on an example’s existing source code. We algorithmically reveal the prime subsets of behavior in a code example via an easy to
understand sequence. In addition, our technique does not require a complete program to work. It can handle partial and non-compiling programs.

Like our work (described in Chapter 6), JTourBus provides a mechanism for incremental navigation of important source code locations (Oezbek and Prechelt 2007). It leads programmers directly to relevant details about the source code, but does not offer an automatic way for identifying these prime locations. In contrast, our MethodSlicing with Reduction technique focuses on carefully slicing the example into a series of cohesive chunks of functionality, reducing long chunks whenever possible.

2.2 Inspecting Java Programs

Multistaging to Understand (described in Chapter 6) shares similarities with code reading by stepwise abstraction (Linger et al. 1979). Code reading by stepwise abstraction calls for inspectors to identify prime subprograms in the software, determine their function and use them to determine a function for the entire program. The effectiveness and efficiency of this technique for structured programs was validated by Basili and Selby (1987), and validated for object-oriented programs by Dunsmore et al. (2003). In contrast, we focus on algorithmically generating all of the prime subsets of behavior in advance for the programmers. We can generate these prime subsets based entirely on the source code’s content.

Another similar approach involves using static program slicing approaches to aid with code inspection. For example, CodeSurfer (Anderson and Teitelbaum 2001)
is advertised as a great companion for code inspections. Tools of this sort tend to be inherently conservative. Consequently, they tend to produce large slices that are often too large to be of practical use. In contrast, we reduce this information-overloading problem by automatically reducing large generated slices (i.e., large code stages). A reduced code stage shows its most informative code elements (i.e., code elements with high usage score) and hides (i.e., folds) its less informative ones.
Chapter 3

The Vesperin System

This chapter introduces the Vesperin system, a system for curating online Java source code\footnote{Parts of this chapter are published in a conference paper at the International Conference on Software Engineering (ICSE) 2015 titled “Source Code Curation on StackOverflow: The Vesperin System.” (Sanchez and Whitehead 2015)}. The goal of Vesperin is to provide an abstraction that allows code foragers to curate online source code as a set of manual and semi-automatic edits, as well as extra documentation.

Vesperin’s operations are firmly grounded in a chain of premises: (1) the quality of online source code has implications for effective code foraging; (2) the quality of source code is represented by a set of desired characteristics; (3) these characteristics are described by a series of quality dimensions; such as accuracy, adaptability, completeness, and understandability; (4) Source Code Curation can be used to improve these dimensions.

Vesperin consists of two main components: a Chrome Extension (named Vi-
olette) for allowing developers to curate online source code in-place, and a RESTful service (named Kiwi) for managing source code curation and parsing operations. Together, they provide an interface designed to make it easy for code foragers to examine (via curation) the online source code they encounter.

The following sections unpack Vesperin’s first application, as well as its two main components. Specifically, we’ll describe Vesperin’s use model and architecture.

3.1 Vesperin on StackOverflow

Vesperin’s first application is StackOverflow. StackOverflow is a software question and answer site for programmers written by programmers, with a large and rapidly growing user base. Programmers using this site engage in a complex code foraging process of understanding and adapting the code examples they encounter. As an increasingly popular platform, and an amazing treasure trove of code examples, it seems natural for us to use Vesperin’s capabilities to address the challenging nature of code examples on StackOverflow, in particular, Java code examples.

To explore the extent and challenging nature of code examples on StackOverflow, we collected evidence from 50,000 Q&A pages accumulated over a 4 year span (i.e., [2011,2014]), and studied them in detail. Criteria for collecting pages are simple: (1) each answer must contain code examples; (2) it should target Java; and (3) it should be an accepted answer.

We found that 25,472 (51%) of the studied Q&A pages are described by single

\footnote{Our data set, and results are available at http://goo.gl/6IbfV1}
snippets, which are typically short, poorly structured, and incomplete. Of these, the median number of lines of code is 8 and 75% of them have less than 19 lines of code. Moreover, 99% of these single snippets are unparsable. The remaining 49% are described by multiple snippets. The average number of lines is 45. Similar to the single snippets group (51%), they are typically incomplete, and thus unparsable. These characteristics raise challenges for developers who are trying to digest unfamiliar code on StackOverflow. As a result, these characteristics are addressed by Vesperin.

3.2 Vesperin Use Model

Figure 3.1 shows the Vesperin’s use model and the connection of a Vesperin page to a Web page containing Java source code. This model is described in this section.

Figure 3.1: Vesperin’s Use Model and the connection of a Vesperin page to a StackOverflow page. A Vesperin page is pictured as a layer placed on top of a Q&A page containing Java code examples. On that layer, code foragers can curate the enclosed examples.
3.2.1 Scratch Space and Drafts

_Vesperin_ assures that code blocks containing Java code have their own scratch space. A scratch space is an augmented code editor that allows programmers to interact with Java code examples. It is the place where code foragers make all the edits; either manually via direct editing or semi-automatically via _Vesperin_'s built-in operations. An auxiliary interface controls the version structure of a code example affected by these edits.

When performing edits on the scratch space, code foragers can mark up major versions (or drafts) of code examples. Marking a draft is a form of checkpointing (Archer Jr et al. 1984), and saves a snapshot of the changed code example for future recoveries. A draft is a named set of edits. Edits include insertions or deletions of text. A marked draft is final, and cannot be deleted.

The purpose of marking drafts is twofold. First, it provides a view of how the code example has changed over time. Second, it adds error tolerance into the curation process. As such, marking drafts enables exploratory learning (Olsen 2009) during the curation process.

In summary, code foragers can evaluate previous changes to the code example and then try new code modification ideas without fear or commitment. Especially, since they can recover from any mistakes by restarting their actions from any marked point.
3.2.2 Other Elements

As code foragers curate code examples, there are times when they need more context. Code foragers can give this context via notes. A note is used to enter comments, tags, and could be used either to summarize some code or outline its intention. Ultimately, notes provide meaningful feedback to those trying to understand a code example. For all these reasons, Vesperin supports notes.

Using notes in the above manner may present a fundamental tension. Either we consider keeping notes connected to a specific global point in the text (which can change in future drafts), or we consider keeping notes local to their corresponding draft; thus making it unavailable to other drafts. The latter turned out to be simpler and is our preferred solution.

3.2.3 Vesperin Actions

Creating and using scratch spaces on a Q&A page are two of Vesperin’s main actions. Creating a scratch space on a rendered Q&A page is facilitated by having some abstract representation of the Q&A page structure. The Document Object Model (DOM) tree of the Q&A page provides such a representation. However, the DOM of a rendered Q&A page consists of many nodes that have no Java code inside. These are nodes that are uninteresting to code foragers and thus should be ignored. Consequently, we need a focused method that detects Java code blocks in the DOM, while minimizing resources spent inspecting the DOM.

Vesperin introduces an algorithm for inspecting the DOM, transforming out
(i.e., “reDOMing”) Java code blocks and ignoring those out of concern. The algorithm consists of three stages. First, it collects each valid code block (specifically `pre` or `code` HTML tags) found on a StackOverflow Q&A page, and ignores invalid ones. A valid code block is a block that has the following characteristics: (1) it contains Java code, (2) it does not have nested `pre` tags, (3) it has an `outerWidth` value greater than 100 pixels, and (4) it has wrapped code with more than 10 lines of code. Second, it creates a scratch space for each collected code block. Third, it binds a scratch space element to each of the collected code blocks on the rendered page. This process includes the addition of meta-data to each of the bound HTML tags to guarantee scratch space uniqueness. After binding, valid code blocks will have an Edit button appear in the top right corner on hover. In order to create a scratch space, one would click on this button.

When using the scratch space, the code forager modifies a code example directly by editing it. At any time, during the editing step, this individual may trigger a semi-automatic code transformation, or refactoring. Vesperin follows this action by issuing a remote call to Kiwi, requesting an update to the current code example. Kiwi’s reply contains either a new version (updated code), or a message describing a failed updating attempt. At any time during editing, the code forager can verify if the current version is syntactically correct (i.e., no compiler errors).

Lastly, as previously discussed, the scratch space keeps a history of all marked drafts. At any time, the code forager may view any previous point in the curation.

---

3We intentionally selected code examples that were as short as possible to adequately support analysis by programmers, yet were long enough to allow programmers to make significant judgments on them via source code curation.
process. Or, they may restart their actions from a marked draft in case of a mistake.

3.2.4 D.R.Y.ING Source Code Curation

In some cases, the code examples of a set of Q&A pages may have been curated by some code foragers, and continue to be viewed by others. Before curating any of those examples, these viewers can check if the code examples have already been curated, thus avoiding any unnecessary duplication of curation work. This is an example of applying the “Don’t Repeat Yourself” (D.R.Y) principle to Source Code Curation.

D.R.Y.ing curation facilitates code examples scrutiny, as it maintains fewer curated code examples (and their revisions) that can be displayed to the code foragers. Vesperin keeps unique records of all curated code examples, including their revisions, in a database. The main identifier for a curated code example is created by combining the StackOverflow Q&A page URL with the ID of the accepted answer that provided the original code snippet. In the current version of Vesperin, tracked revisions are available through Vesperin’s Web interface, which is described in Section 3.3. We used a Most-Recently-Used (MRU) strategy, used in cache algorithms (Chou and DeWitt, 1986), to provide a list of alternative curated examples to the code forager. We limit the number of available alternatives in the list to 5.

3.3 Vesperin’s Architecture

Figure 3.2 shows Vesperin’s architecture. Indicated in the figure are Violette and Kiwi components. Kiwi provides a RESTful interface to Vesperin back end services.
We describe both components in this section.

Figure 3.2: *Vesperin*’s architecture. Dimmed elements (e.g., codepacking) in the architecture are features that we’ll unpack in Chapter 5 and Chapter 6 respectively.

### 3.3.1 Violette

*Violette* is *Vesperin*’s companion Chrome extension. When installed, *Violette* detects Java code blocks on StackOverflow Q&A pages and then transforms them into a form that allows code foragers to curate the source code in-place. We call this form the scratch space. *Violette* is implemented using the Javascript programming language. Its source code can be found online at [https://bitbucket.org/huascarsanchez/violette](https://bitbucket.org/huascarsanchez/violette)
Figure 3.3: Violette’s User Interface. This interface is designed to make it easy for code foragers to examine (via curation) the online source code they encounter. The stage visual component, along with its functionality, is described in Chapter 6.

Figure 3.3 shows Violette’s User Interface. Indicated in the figure are Violette’s components. Out of the box, Violette supports the following operations:

- Programming language detection. Detecting any Java code on a Q&A page is key for the creation of scratch spaces. Consequently, Violette combines disambiguation heuristics with other greedy heuristics used by syntax highlighters to guess the language of a code fragment.

- In-place code modifications. Besides support for manual code modifications via direct editing, Violette supports a few behaviour-preserving code transformations (Fowler 1999; Mens and Tourwé 2004). These code transformations are described in terms of the atomic procedures of renaming, moving, deleting, and introducing new code. With the exception of deduplication, which analyzes an entire code

---

4 As seen on [https://github.com/github/linguist/](https://github.com/github/linguist/)

5 See [https://highlightjs.org/](https://highlightjs.org/) for information on programming language detection.
snippet in search for clones (Baxter et al., 1998) to remove, each of these transformations analyzes the scope of a code selection, and uses this information to update the structure of a code example. The list of supported transformations includes:

1. Rename member. This strategy allows code foragers to change the name of any member (e.g., field, parameter, variable, method, class) in the code example. All references to and usages of the member are updated automatically. Code foragers can invoke this strategy by clicking the Rename member button or by pressing either the Ctrl+R (Windows) or Cmd+R (Mac) keys on their keyboards.

2. Delete member. This strategy allows code foragers to delete members (e.g., field, parameter, variable, method, class) in the code examples. Two forms of delete are supported: safe delete and normal delete. Safe delete ensures that the code example will compile after the deletion. If no compiler errors are found, then it removes the member right away; otherwise it issues a warning or a set of warnings. Normal delete removes the member right away, without ensuring anything. Code foragers can invoke safe delete by clicking the Delete member button or by pressing either the Ctrl+D (Windows) or Cmd+D (Mac) keys on their keyboards. They can invoke normal delete by first selecting the code member to be deleted and then pressing the delete key on their keyboards.

3. Code cleanup. This is a composite strategy. It sequentially performs 3 op-
erations: Remove Unused References, Deduplicate Members, and Reformat Code According to Predefined Code Style. This strategy allows code foragers to instantly eliminate unused methods or fields, duplicated references, and to apply a code style to the existing source code. Code foragers can cleanup the code example by clicking the Code Cleanup button.

4. Clip fragment. It allows code foragers to select a code fragment of interest, then either extract it as a new method within the same version of the code example or save it as the core method (including its needed dependencies) in a new version of the code example. Clipping a fragment is a way of chunking related code statements (See Section [1.6.3 in Chapter 1]). Code foragers can invoke this strategy by clicking the Clip fragment button or by pressing the Cmd+E keys on their keyboards. Choosing to either extract this fragment inside the same version or save it as the core method in a new version depends on the scope of the selection. If the code selection does not include exactly a whole method, then the former strategy is selected, otherwise the latter is selected.

5. Create new method. This strategy allows code foragers to create new methods inside the current version of the code example. As inputs, it takes the name of the method, a list of parameter-and-type pairs, and a return type. Code foragers can invoke this strategy by pressing the Ctrl+Shift+N (Windows) or Cmd+Shift+N (Mac) keys on their keyboards. If used in conjunction with the Clip fragment function, then the body of the newly created method is
basically the clipped fragment.

• Content annotation via notes. Notes can highlight important code sections within the scratch space. They are in context. In other words, they live on the code fragment level rather than on the entire code example. Notes are shown by clicking the Notes button on the upper right corner of the scratch space. Leaving notes in context is beneficial to code foragers since they can set specific feedback on what parts of the code example are challenging (or resonating with) to them. Notes are added by first selecting the code fragment of interest and then pressing the Create Note button. This will display an input dialog where the code forager can enter the text for the new note.

• Lightweight drafts management. It provides an auxiliary interface to control the version structure of an edited code example. We call this interface the Edit Tracker. This interface tracks drafts marked by code foragers; produced during their curating session. Moreover, code foragers can use it to compare old drafts with the current version of the code example (if they’d like to see how the example has changed over time). Code foragers can click the History button to turn on the Edit Tracker. Code foragers can mark drafts by pressing the Ctrl+S (Windows) or Cmd+S (Mac) keys on their keyboards.

• Syntax correctness checking. Transforming a code example is by no means trivial. It requires access to the code example's syntax and static semantics (Fowler 1999). Among other criteria, such as efficiency and usability, the reliability of a
code transformation and/or refactoring is vital for its use. Consequently, Vesperin provides a mechanism for automatically checking if the edited code examples are syntactically correct. Code foragers can access this mechanism via Violette. They can execute this functionality when performing curation operations that require Kiwi’s help, when marking a draft, or when pressing the Ctrl+I (Windows) or Cmd+I (Mac) keys on their keyboards. The result will be either a light blue dialog indicating a non-error state or a light red dialog indicating an error state.

- Documentation mode. It provides the option of scrubbing the edited Q&A page of distractions and provides a clean view for examining the curated code example. In this mode, code foragers can provide additional metadata, such as structured tags, that could aid with their retrieval. Moreover, they can save it in a comfortable reading view for examining now or later on one’s computer. This mode is optional. Code foragers can choose to activate it during the sharing of a curated code example.

### 3.3.2 Kiwi

The Kiwi API is implemented in Scala. Its source code can be found online at [https://github.com/hsanchez/vesper-http](https://github.com/hsanchez/vesper-http).

Kiwi consists of two logical services: Curation and Parsing. The Curation service manages Vesperin’s curation operations, such as code transformations and refactorings, code presentation, and publication (i.e., publish on Twitter). The Parsing service
manages the access to a Java parsing engine, built atop the Eclipse JDT library. All requests are, by default, provided as JSON. There is no authentication required to make Curation and Parsing API calls.

The Kiwi HTTP layer uses the Spray framework to build a REST/HTTP-based integration layer. This layer is responsible for serving HTTP requests coming from Violette. Table 3.1 describes the Kiwi API in terms of a set of HTTP endpoints.

<table>
<thead>
<tr>
<th>HTTP Verb</th>
<th>Endpoints</th>
<th>HTTP response code</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST /kiwi/eval?command</td>
<td>201 (Created), 404 (Not Found), 409 (Conflict)</td>
<td></td>
</tr>
<tr>
<td>PUT /kiwi/eval?command</td>
<td>200 (OK), 404 (Not Found), 204 (No Content)</td>
<td></td>
</tr>
<tr>
<td>GET /kiwi/find?q=query</td>
<td>200 (OK), 404 (Not Found), 400 (Bad Request)</td>
<td></td>
</tr>
<tr>
<td>GET /kiwi/render?id=id</td>
<td>200 (OK), 404 (Not Found), 400 (Bad Request)</td>
<td></td>
</tr>
<tr>
<td>DELETE /kiwi/sourcecode</td>
<td>200 (OK), 404 (Not Found), 204 (No Content)</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Supported HTTP Endpoints by Kiwi. All of the endpoints are preceded by vesperin.com host.

Below is a more-detailed description of the main HTTP methods and HTTP response codes supported by Kiwi:

1. POST verb. The POST verb is most often used to create new resources. Kiwi uses it to publish a curated source code on Twitter and saves it in a mongodb database, hosted on Heroku, a Web hosting company and service. As input, it takes a command, provided as a JSON request. Upon a successful creation, it returns a location header with a link to the newly-published resource with the 201 HTTP status. POST is neither safe nor idempotent. In other words, making two identical POST requests will most likely result in two published and saved

---

[http://spray.io/](http://spray.io/)
instances of source code containing the same information.

2. PUT verb. The PUT verb is often used to update resources. *Kiwi* uses it to issue code modification actions against a source code of interest. As input, it takes a command in the form of a JSON request. Upon a successful update, it returns a 200 HTTP status and a result package. This result package contains the new version of the source code, as well as its old version. PUT is not a safe operation, in that it modifies a source code of interest on the server, but it is idempotent (i.e., it can be called many times without different outcomes).

3. GET verb. The GET verb is used to retrieve a representation of a resource. In this context, a representation of the curated code example. In a non-error path, *Kiwi* uses it to return a representation of the curated code example in JSON (See first GET in Table 3.1) or HTML (See second GET in Table 3.1) and an HTTP response code of 200 (OK). GET is a safe and an idempotent operation.

4. DELETE verb. The DELETE verb is used to delete a resource identified by the Uniform Resource Identifier (URI). Code foragers cannot invoke this verb from *Violette*. We are the only ones who have the credentials to do that. We use it only to delete test data.

*Kiwi* supports an array of commands. These commands represent either curation operations or syntax checking operations (See Section 3.3.1 for details). With the exception of DELETE, *Kiwi* supports the same set of operations found in *Violette*. Figure 3.4 provides an example of a *Kiwi* command, in its JSON form.
In summary, this section provides an overview of the Kiwi RESTful API. In what follows, we provide an example of a source code curation session.

### 3.4 Putting It All Together

Now that Vesperin has been described, along with its Chrome extension and RESTful API, it is natural to ask how these can be put together to perform a real-world curation task. We chose the task of locating a code example that shows how to parallelize the QuickSort algorithm. We wanted an end-to-end example that would start
with a raw code snippet (i.e., uncurated), and ends with curated version of that source code; a version a code forager understands better. This section describes this real-world curation task. In the remainder of this section, we will refer to the code forager simply as Bella.

![Google Search](image)

**Figure 3.5**: Searching for a code example that shows how to parallelize the QuickSort algorithm.

Figure 3.5 shows the first step. This step takes as input a query describing Bella’s information need. Bella issues this query to the Google search engine. After issuing the query, the search engine comes back with a set of generated results matching the query. At this point, Bella eagerly selects the first result. Figure 3.6 shows the results returned by the search engine.

---

*Bella can use any Web search engine to complete this step, such as DuckDuckGo, Bing, etc.*
Upon loading the Q&A page, Violette makes sure that every valid `pre` HTML tag (usually denoting code blocks) will now have an Edit button which appears when Bella hovers over the block with the on-screen pointer. Bella then looks for the Q&A’s accepted answer. Once found, she hovers over its code block and then sees the red Edit button (See Figure 3.7).

Figure 3.6: Selecting one of the many results generated by the search engine.
Next, Bella clicks the Edit button to create a new scratch space (Figure 3.8).

Violette responds to this action by tracking the newly-created scratch space.
Figure 3.8: The code forager clicks the Edit button. A new scratch space is created.

Bella takes a glance at the source code included in the scratch space. Then, in order to jumpstart her curation process, she begins by cleaning the source code. To do this, she triggers the Code Cleanup function by clicking the Code Cleanup button in Violette (See Figure 3.9). The first column on this figure shows the clicking of the cleanup function. The second column shows the output of this function. The output of this function is a version of the snippet formatted according to Vesperin’s code style. No deduplication or unused references removal were needed.
Bella senses this action will change the source code in many ways, and decides to mark a draft before continuing the curation task. Bella marks a new draft by using the Cmd+S key. *Violette* saves a new snapshot of the source code and then indicates to Bella that a new draft has been created. Additionally, it indicates the source code has errors. As a result, it outlines the scratch space in red and displays the error in a red message box (See Figure 3.10).
Figure 3.10: The code forager has marked a new draft and has found that the source code lacks some external references.

Bella reacts to this error by adding any missing external references. In total, she finds 4 missing references. That is, `Executors`, `ExecutorService`, `Future`, and `TimeUnit`. She follows this with another inspection. To her surprise, Violette comes back with two more errors. This time, the errors are missing methods (See Figure 3.11).

Figure 3.11: The code forager has added the missing references and then re-inspected the source code. Violette has come back with new errors.
Bella uses a combination of Web search, her own experience, and `Cmd+Shift+N` key (i.e., Create new method operation) to implement these two missing methods. After implementing these methods, Violette finds no more errors (See Figure 3.12).

![Image of code implementation](image)

Figure 3.12: The code forager implements the missing methods and then re-inspects the source code. Violette finds no more errors.

Now that everything looks good, Bella marks another draft. Then, she tries to understand the rest of the source code. To do that, she uses a bottom-up comprehension strategy (Brooks, 1983). Bella reads the body of the `quicksort` method. Then, she chunks a code fragment she thinks is related to the selection of a pivot. She uses the `Cmd+E` key to chunk this fragment, and then the `Cmd+Shift+N` key to put it in a new method. She names this method `selectArrayPivot`. Figure 3.13 illustrates this action.
Bella realizes now the `selectArrayPivot` she just created is none other than the QuickSort’s `Partition` method. As a result, she renames this method to `partition` by selecting the method’s name, and then clicking the Rename member function. Figure 3.14 illustrates the steps involved in the Rename member operation, as well as its output.
At this point, Bella is getting the hang of chunking. As a result, she reads the next code fragment in the `quicksort` method, and then immediately chunks it. This time, she names the new method `futureQuicksort`, as she sees this fragment is wrapping a `quicksort` call into an asynchronous object. Figure 3.15 illustrates the result of Bella’s action.
Next, Bella wants to compare the current version of the source with old versions in order to see how this source code has changed over time. She does that by clicking the History function. Figure 3.16 illustrates this action.

After all this work, Bella feels confident with the curated code. She thinks she can use it in her own work. As a result, she brings it into her desktop. She does that by
clicking the Bring to Desktop function, located at the bottom left corner of Violette (See Figure 3.17). And with that, Bella ends the curation session.

Figure 3.17: The code forager ends the curation session by downloading the curated source code.

3.5 Conclusion

This section has presented the Vesperin system, as well as a source code curation concept that combines manual and semi-automatic code transformations into the code foraging process. Unlike other source-code-centric tools, Vesperin deals with the inherently questionable quality of online source code upfront (in the source code’s origin Web page), before consumption in some external development environment. Vesperin is
relevant to code foragers by providing an editing scratch space for exploratory learning, thus allowing the application of code modifications ideas in-place to steer source code understanding. Section 3.4 has only scratched the surface of the things one can do with the Vesperin system. In the remainder of this thesis, additional features will be presented, such our framework for presenting curated source, and other code annotations (i.e., Notes) in context. In what follows, we will discuss Vesperin’s empirical validation.
Chapter 4

Evaluating Vesperin

This chapter describes the evaluation of Vesperin. This evaluation seeks to establish evidence of the feasibility of the Vesperin end-to-end approach for curating code examples. Particularly, we focused on evaluating Vesperin with respect to two attributes: (1) better understanding, and (2) added value.

To evaluate these attributes, we conducted an exploratory user study. We asked people from both academia and industry to participate in the study. Our evaluation considered the following three concrete questions:

1. How will developers use these source code curation interfaces?

2. Will the set of provided facilities be sufficient for curating code examples?

3. Will developers be able to better understand unfamiliar code examples via curation?
4.1 Method

4.1.1 Participants and Design

The study involved 15 users, solicited from a public mailing list at two college campuses. Six were undergraduate students (computer science majors), and nine were master’s students (software engineering). Participants had previous experience with the Java programming language, had used code foraging, were familiar with StackOverflow, and had previous experience with code refactoring.

We set up the experiment in two locations on separate dates. We gathered the 6 undergraduate students in a classroom at UC Santa Cruz, and the 9 master’s students in a classroom at San Jose State University. We were physically present at both locations.

We used a one group pretest posttest design (Campbell et al., 1963) to address the 3 concrete questions we stated above. This is a pre-experimental study, as it only studies a single group and it makes no comparison with an equivalent non-treatment group (Babbie, 2015). Nonetheless, this design allows us to report on facts of real user behavior, even those observed in under-controlled and limited-sample experiences.

Each participant’s session consisted of three distinct parts: (1) measure the development experience of the participants, and their familiarity with certain practices and technology relevant to our study, (2) observe what strategies participants employ when trying to understand Java code examples using Source Code Curation; and (3) observe whether Vesperin’s source code curation facilities can improve understanding, and
whether these facilities are sufficient.

Our study is exploratory, as we are trying to understand the nature of how participants work with Vesperin. The results of the study will help us understand the dynamics involved in Source Code Curation. In what follows, we describe the materials, procedure, and manipulations used in the study.

4.1.2 Used materials

4.1.2.1 Code Examples

Java was the language chosen for the code examples, being both the language with which the participants had most experience and the only language supported by Vesperin. The 3 Java code examples presented to the participants were obtained using our StackOverflow search application (see Figure 4.1). These examples were: (1) how to use client server certificates on an Android application, (2) how to add push notifications to an Android application, and (3) how to use Twitter4J’s Twitter Stream API.
Search Cue, a StackOverflow Search Application

A brief introduction to get started with Search Cue.

It’s easy to get started with it. You use it by pressing the START button below. Behind the scenes, the application fetches a few batches of code examples from StackOverflow. It only the ones that match some criteria.

The first time you use it, the results get cached on your browser. After that, every time you search, this is done on your computer.

**Figure 4.1:** Search Cue is a StackOverflow Search Application.

*Search Cue* is an application that uses StackExchange API[^1] for searching StackOverflow for any questions which fit a given search criteria. The search criteria used includes: (1) the code example should contain valid Java code block[^2] and (2) the code example is an accepted answer. These criteria are hardcoded within *Search Cue*’s main functionality. Therefore, they cannot be modified by *Search Cue*’s users.

The context and domain of the code were familiar to the participants, even though all the Java code examples were new to them.

[^1]: https://api.stackexchange.com/
[^2]: The characteristics of what a valid Java code block are described in Section 3.2.3
4.1.2.2 Background Questionnaire

The set of questions included in this questionnaire are displayed in Figure 4.2.

We provided this questionnaire before the pre-testing of participants, which we will cover in Section 4.1.3.

![Figure 4.2: Background questionnaire given to participants at the beginning of the study.](image)

The above questions are closed-ended questions. Their intent is to gauge participants' programming experience and familiarity with StackOverflow, Java, and Refactoring.
4.1.2.3 Pretest and Posttest Questionnaires

We collect data to evaluate whether the use of Vesperin leads to better understanding and added value via the use of both pretest and posttest questions containing multiple choice questions. Specifically, we use a 5-point likert scale, ranging from strongly-disagree to strongly-agree. See Tables 4.1 and 4.2 for the pretest and posttest questions.

<table>
<thead>
<tr>
<th>Understanding Java Code Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>In this experiment, you will be using a tool for curating Java code examples. This tool allows you to explore code modifications in-place. It can help you break code examples up into smaller functions, deduplicate, and then rename those smaller functions accordingly.</em></td>
</tr>
</tbody>
</table>

For each of the statements, please indicate to what extent you agree with them, ranging from 1 (strongly disagree) to 5 (strongly agree).

1. Such a tool could allow me to better understand a code example.
2. Such a tool could allow me to feel certain, or confident, that I really understand the code example that I’m investigating.
3. Such a tool’s support for in-place code modifications adds value.
4. The value added by such a tool will be minimal.
5. Such a tool could save me time when trying to adapt a code example to match my task needs.

Table 4.1: Pretest questionnaire.
For each of the statements, please indicate to what extent you agree with them, ranging from 1 (strongly disagree) to 5 (strongly agree).

1. Tool Experience
   (a) Found Vesperin easy to use.
   (b) Vesperin should have been integrated with an IDE.
   (c) Vesperin aided with code understanding.

2. Tool Adequacy
   (a) Vesperin’s support for in-place code modifications (e.g., manual editing, semi-automatic code transformations, and search & replace) adds value.
   (b) The value added by systems like Vesperin is minimal.
   (c) A system like Vesperin saves me time when adapting code examples to match my task needs.
   (d) A system like Vesperin allows me to better understand online code examples.
   (e) A system like Vesperin makes me feel certain, or more confident, about my understanding of the code examples I reviewed.

3. Tool Features.
   Below are a number of facilities offered by Vesperin. Please select your top 3 facilities. E.g., write “1” for the best feature, “2” for the second best feature, and “3” for the third best feature. If choose Other, then please state the missing feature.
   (a) Notes in context.
   (b) In-place code modifications.
   (c) Drafts management.
   (d) Syntax correctness checking.
   (e) Sharing of Curated code.
   (f) Search & Replace.
   (g) Other: ________________

Table 4.2: Posttest questionnaire.
4.1.3 Procedure and manipulations

Our strategy for establishing evidence of the feasibility of the Vesperin end-to-end approach for curating code examples is twofold. First, we ask participants to work on a set of source code curation tasks. Second, we test them before and after finishing the source code curation tasks. The goal is to measure participants’ perception prior to using Vesperin and their experience after using it.

4.1.3.1 Pretest

We started this part of the study with a background questionnaire (See Figure 4.2). Information gathered from this questionnaire helped us measure the development experience of the participants, familiarity with StackOverflow, Java, and Refactoring.

Upon the completion of this background questionnaire, we gave a 20 minute overview of the Web sites and tools used in the study: StackOverflow, Google Chrome, and Vesperin. The main focus was to walk participants through a source code curation session, using Vesperin. The main goal was to expose participants to all the source code curation facilities offered by Vesperin.

After the overview, we gave participants a pretest questionnaire that measures their perception (and general knowledge) of Vesperin. Next, participants took a short break to regroup their thoughts.
4.1.3.2 Posttest

After the short break, we directed participants to work on a set of program understanding tasks for 60 minutes. There were a total of 3 tasks (i.e., 3 code examples to review). A single task consisted of 4 parts, which we illustrate in Table 4.3.

<table>
<thead>
<tr>
<th>Goal:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Understand the overall functionality of a Java code example (with \textit{Vesperin}).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task:</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{If you are having trouble completing this task or you are not sure what to do next, then please indicate the problem you are having with the task, and then move on to the next task.}.</td>
</tr>
</tbody>
</table>

1. Randomly select a result from the set of results presented by us.

2. Examine the content of a selected result.

   (a) Look for ways to break it up into smaller functions, deduplicate, and then find good names for those smaller functions.

   (b) Mark drafts frequently.

3. Consume the examined content by describing in detail its intent (including a description of any possible tradeoffs involved in its reuse).

4. Move to the next result.

Table 4.3: Single program understanding task.

For the 3 tasks, we gave no direction on how participants should read a given code example. It was up to them to choose their code reading strategy (\cite{Spinellis2003}, gain some understanding of the code example’s main abstractions, and then determine
its intent. However, we recommended that participants use Vesperin whenever possible. Specifically, we recommended that subjects look for ways to break up the code example into smaller functions, deduplicate, and then rename them accordingly. If the participants struggled with a task, we instructed them to indicate the problem they experienced and then move on to the next task. This strategy prevented participants from getting stuck and not making any progress in the overall experiment. Moreover, it allowed us to focus on finding out as much information as possible about the curation strategies used by the participants.

Upon the completion of the assigned tasks, we gave participants a posttest questionnaire that would measure their experience (and general knowledge) with Vesperin. We also asked them to rate their top 3 features in Vesperin.

At the end of this part of the study, we interviewed participants to learn more about their encountered problems (experienced during the experiment), their understanding of what was required from them, and as well as their recommendations for improving the Vesperin system.

4.2 Pilot Sessions

Before performing the main user study, we conducted 2 pilot sessions to fine tune it. The pilot sessions involved 4 participants, solicited from a public mailing list at a college campus. None of the participants from the pilot sessions were included in the main study. The pilot sessions were performed similarly to the actual study. However,
they differed in purpose. The purpose of the pilot sessions was twofold. First, to check whether the tasks were clear. If they were not clear, then we made the necessary improvements. Second, to check whether we could record participants’ actions throughout the sessions.

The first pilot session turned out to be cumbersome. We had a hard time reconstructing the participants’ thinking steps and their respective actions. Consequently, we developed an interaction event model to help us gather user interaction telemetry (user interaction event messages). Under this model, user behavior is represented as a cycle of alternating reviewing and editing phases: an editing session is persisted when the user starts reviewing some code. We assume a user is reviewing some code when this user either goes idle or interacts with some code element. The user interacts with a code element by either clicking within the scratch space, scrolling until that code element is in view, or highlighting that code element. An editing session is a sequence of user interaction event messages concerning code edits that have occurred within a given time period.

The interaction event model is implemented using a 5-tuple finite state machine \((Q, \Sigma, \delta, q_0, F)\) (illustrated in Figure 4.3). In this finite state machine (FSM), (1) \(Q = \{\text{REVIEWING, EDITING}\}\) is a finite set of states; (2) \(\Sigma = \{\text{Review, Edit}\}\) is a finite set of events; (3) \(\delta\) is a transition function mapping \(Q \times \Sigma\) to \(Q\); (4) \(q_0 \in Q\) is FSM’s initial state; and (5) \(F \subseteq Q\) is the set of final states (i.e., \(F = \{\text{REVIEWING}\}\)).

\(^3\text{Choosing an appropriate idle period is highly dependent on the application. In this thesis, we assume an idle period of 5 seconds.}\)
We instantiate this model when using Vesperin. During reviewing, we add created sessions to an oracle (See Review⊙) and save it locally if dirty (See Review⊙). The oracle is a dictionary object initialized during the creation of a Vesperin’s scratch space. When the user switches to editing (See Edit□), we create a session and then assign a queue to it. While at this state (See Edit□), we add any created interaction event message (i.e., \((\text{elapsedtime}, \text{interaction}, \text{text}, \text{errors})\)) to the queue of the newly created session. When the user switches back to reviewing, we add the current session to the oracle. After that, we removed it from the queue. The categories of interactions with Vesperin are as follows:

- **Query**: A search for the location of a code element (e.g., method, field).

- **Navigation**: A command to turn on either the Edit Tracker or the Notes view.

- **Edit**: Insertions or deletions of text; performed either manually via direct editing or semi-automatically via Vesperin’s built-in operations (e.g., Delete Member).

\[4\text{We save the oracle into Chrome’s local storage.}\]
• **Click:** A mouse selection of a code fragment within *Vesperin*’s scratch space.

With this model, we are able to catalog the time and the affected code sections the participants interact with during a source code curation session, the frequency of interaction events, as well as the interactions that have led to errors. This information represents a record of a user’s interactions with *Vesperin*. By examining the structure of this record, we can reconstruct participants’ thinking steps and their respective actions during a source code curation session. Particularly, we can look at how long participants stayed in a given code section, the changes they made in that section, and their transition patterns between the changed section and other code sections. A transition is a change of location from one code section to another.

The second pilot session went more smoothly than the first pilot. However, we found that some of the tasks were too difficult to complete. As a result, we modified them in order to reduce their difficulty. Inline with this action, we included in the introduction of the tasks a clause that indicates that participants can skip tasks they consider too difficult to complete, in order to continue making progress. Moreover, we reduced the number of questions per questionnaire. Lastly, we improved *Vesperin*’s front end interface (*Violette*) to match some of the 10 usability heuristics introduced by Nielsen (1994). One example would be to favor recognition over recall. In this example, the goal is to minimize users’ memory load by making objects, actions, and options visible.
4.3 Participant Background

We required all participants to have a college degree in computer science or related discipline. Participants had previous experience with the Java programming language, had used code foraging, were familiar with StackOverflow, and had previous experience with code refactoring.

In this study (see Figure 4.4), 40% of these participants have between 5 and 10 years programming experience. 40% of them visit StackOverflow multiple times a day. Moreover, nearly 70% of the participants were extremely familiar with Java and Refactoring.

<table>
<thead>
<tr>
<th>Programming Experience</th>
<th>StackOverflow Visit Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [5, 10]</td>
<td>6 / 40%</td>
</tr>
<tr>
<td>2 &gt; 10</td>
<td>4 / 27%</td>
</tr>
<tr>
<td>3 [2, 5]</td>
<td>3 / 20%</td>
</tr>
<tr>
<td>4 [1, 2]</td>
<td>2 / 13%</td>
</tr>
<tr>
<td>5 &lt; 1</td>
<td>0 / 0%</td>
</tr>
</tbody>
</table>

(a) Programming Experience.  
(b) StackOverflow Visit Frequency.

<table>
<thead>
<tr>
<th>Level of Java Familiarity</th>
<th>Level of Refactoring Familiarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Extremely familiar</td>
<td>10 / 67%</td>
</tr>
<tr>
<td>2 Moderately familiar</td>
<td>3 / 20%</td>
</tr>
<tr>
<td>3 Somewhat familiar</td>
<td>2 / 13%</td>
</tr>
<tr>
<td>4 Not at all familiar</td>
<td>0 / 0%</td>
</tr>
<tr>
<td>5 Slightly familiar</td>
<td>0 / 0%</td>
</tr>
</tbody>
</table>

(c) Level of Java Familiarity.  
(d) Level of Refactoring Familiarity.

Figure 4.4: Summary of participants’ background information.
Information gathered from participants concerning their background (summarized in Figure 4.4) helped us measure participants’ programming experience, as well as their familiarity with the technology required in this study.

4.4 Results

Our results cover how the participants used Vesperin, including the evaluation of the attributes mentioned in the beginning of this chapter.

Specifically, our results cover (1) what strategies participants used to evaluate their built mental models; (2) whether the provided curation facilities were sufficient for curating code examples; and (3) whether participants were able to better understand them via curation. This last one also included questions related to whether the overall system added value.

4.4.1 How will developers use these source code curation interfaces?

We posed this question in order to discover what use strategies participants employed when using our system, and to evaluate their mental model representations. We obtained these results from observations performed during the whole experiment and from the output of our interaction event model (summarized in Table 4.4).

In 60 minutes, participants curated 3 Java code examples. In that time window, their actions yielded an average of 10 different editing sessions. These sessions contained interactions targeting a few code locations. Although programmers interacted with many methods on each source code curation task, only few exhibited a higher visit frequency.
The visit frequency of a method is the number of times a programmer has accessed (or interacted with) that method inside the scratch space. On average, 2 methods exhibited a higher visit frequency. One possible explanation of high visit frequency is that method groups with high visit frequency are the methods of interest to a participant (Parnin and Görg, 2006).

<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sessions created</td>
<td>10</td>
<td>7</td>
<td>15</td>
</tr>
<tr>
<td>Edits per session</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Visited methods per session</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 4.4: Interactions with Vesperin

On average, participants performed 2 edits per session. The minimum number of performed edits was 1 and the maximum was 3. The two most frequent edits were Rename member and Clip fragment. Editing activity levels surged early on, then subsided over time (illustrated in Figure 4.5). Figure 4.5 illustrates aggregated edit actions of participants over 2 minute intervals. These actions were gathered from the output of the interaction event model that was introduced in Section 4.2. We tried aggregating all edit actions over 1 minute intervals. However, we could not get the sums large enough to produce a strong pattern. Once we changed the intervals to 2 minutes, we were able to show a much stronger pattern.
Number of Tasks: 3. Each task involves curating a code example. Each curation task had a 20 min limit (60/3).

One possible explanation for the above finding comes from the assumptions of dual-process theories (Chaiken and Eagly, 1989). This is the model that best explains the finding. These assumptions explain that when participants feel doubt, they assume they lack sufficient knowledge. Consequently, systematic information processing (e.g., edits) is one means to obtain that knowledge and help establish certainty. Conversely, this suggests that when participants feel certain, they interpret their feeling as having gained sufficient knowledge and thus eliminate the need for further information processing.

From the gathered data, we also determined the comprehension strategy used by participants. Participants relied on a hybrid comprehension strategy; mixing bottom-
up and top-down comprehension strategies (Corritore and Wiedenbeck 2001). As participants were trying to comprehend the code, they often tried to verify whether it was syntactically correct. For some participants, such feedback gave them some reassurance of the validity of the code. For others, however, the effect of such feedback was the opposite. During editing and verifying, new and old compilation errors sometimes kept occurring. Consequently, they felt frustrated due to not making sufficient progress after a few edits.

Such frustration made these participants apathetic. Rather than focusing on trying understand the example, they were just consumed with verifying a series of impromptu changes. Consequently, they were derailed from the asked task; making their answers to the posttest questionnaire outliers (not representative of the overall collected data). Since the actions taken by these participants were clearly outliers, we excluded these participants' data from our main results. Figures 4.6 and 4.7 show the results after the exclusion of the outliers and the results before the exclusion of the outliers, respectively. Both figures are provided in Section 4.4.3.

We also observed participants exploring control flow relationships via text search-and-replace actions, followed by adding notes and cleanup operations. By looking at the output of our interaction event model, we discovered that the mechanics used by some of these participants matched the mechanics of some of Vesperin’s supported refactorings. When asked why they did not use this functionality, they answered that Vesperin’s front end (Violette) looked like an editor. Consequently, it was natural to them to edit code this way.
Lastly, we asked the participants’ opinion on the following seven features of Vesperin (see Table 4.2): Notes in context (F1), In-place code modifications (F2), Drafts management (F3), Syntax correctness checking (F4), Sharing of curated code (F5), Search & Replace (F6), and Codepacking (F7) – specified by most participants in the Other field.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td>2</td>
<td>2</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>F4</td>
<td>3</td>
<td>3</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F6</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>F7</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Participants’ top 3 features. Each column represents a participant. The 1’s, 2’s, and 3’s represent a participant’s first, second, and third choice respectively.

Participants’ subjective preferences are illustrated in Table 4.5. As can be seen in the table, there is no clear preferred feature. However, we can observe a trend in feature popularity. For instance, In-place code modifications, Search & Replace, and Codepacking seem to be popular. A possible explanation for this popularity is that these features all play a role in enabling a more hybrid source code understanding process throughout the completion of the three source code curation tasks.

4.4.2 Will the set of provided facilities be sufficient for curating code examples?

The principal realization from this experiment is that participants employed a set of Vesperin operations to curate examples. With those operations, they were able
to cope with some of the characteristics we described in the introduction of Chapter 3. However, given the other unused operations, we concluded that a small set of operations were necessary, but not sufficient, for curating different types of code examples. Examples of Vesperin limitations, brought up by participants, include: (1) lack of ability to cope with multiple independent classes simultaneously in a single scratch space; (2) lack of ability to link multiple scratch spaces that should be considered in concert (e.g., tutorials); and (3) lack of ability to automatically resolve missing dependencies or body declarations in the code examples.

More than half (60%) of the participants were pretty vocal about the third limitation. They expressed their interest in having Vesperin deal with the incompleteness of code examples, so they could focus on other more meaningful curation actions. They felt it was simply not feasible to manually resolve dependencies or missing body declarations for every code example they encounter. To address this issue, we developed a technique to help address this incompleteness issue, called Codepacking. Briefly, Codepacking packs each code example with missing code elements in order to give a good start with code foragers’ curation activities. We describe Codepacking in more detail in Chapter 5.

4.4.3 Will developers be able to better understand the unfamiliar code examples via curation?

Figure 4.6 shows the results of the pretest and posttest. Excluding the case where participants lost focus on the task (Section 4.4.1), the pretest and posttest re-
results are similar. Despite any issues encountered, participants came in with a positive expectation and Vesperin did not disappoint. It added value and better understanding as predicted. Interested readers can assess the results of the pretest and posttest before the exclusion of the outliers in Figure 4.7.

Figure 4.6: Distribution of participants’ expectations before (Pretest), and experiences after (Posttest) using Vesperin. Horizontal axes: 5-point Likert scale, ranging from strongly disagree (“- -”) to strongly agree (“++”). Vertical axes: number of participants.

Figure 4.7: Distribution of participants’ expectations before (Pretest), and experiences after (Posttest) using Vesperin. Outlier values are included. Horizontal axes: 5-point Likert scale, ranging from strongly disagree (“- -”) to strongly agree (“++”). Vertical axes: number of participants.
The majority of participants indicated that *Vesperin* could help them understand the code examples more effectively (See Figure 4.6a) with the current operations. When asked whether the system added value, they agreed (See Figure 4.6b). They felt it did provide value in better understanding of unfamiliar code examples.

4.5 Conclusion

In this chapter we described the evaluation of the *Vesperin* system. We evaluated *Vesperin* with respect to two attributes: (1) better understanding, and (2) added value. While our findings are preliminary, they are encouraging. Our evaluation suggests that the *Vesperin* end-to-end approach for curating code examples is feasible. The majority of participants felt they were able to better understand the unfamiliar code examples via curation. Furthermore, they felt the system provided value.
Chapter 5

Codepacking

While curating, programmers often know the approximate structure of examples they are editing, but may yet find it difficult to curate code that does not compile because some fragments are either ill-typed \cite{Lee et al. 2013} or lack body declarations. Such situations occur mainly because most online code examples are incomplete and ambiguous \cite{Dagenais and Robillard 2012}. For this reason it is hard to curate code examples.

In this chapter we propose an approach that takes code examples with either *ill-typed* expressions, missing body declarations, or both and then automatically suggests their appropriate corrections. The suggested corrections follow the structure outlined in the original code example as closely as possible. Expressions that disobey a programming language’s type rules are called *ill-typed* expressions \cite{Lee et al. 2013}, and will cause type errors.
5.1 Our approach

Repairing code examples with ill-typed expressions is difficult. This is especially the case if the local information present in their source code is incomplete, and thus insufficient for resolving all their bindings at compile time. Bindings provide an integrated picture of the structure of the program as seen from the compiler’s point of view (Jdt, 2015). They draw connections between the different parts of a program. As such, they allow us to find out to which declaration in the AST a reference belongs. They also allow us to detect whether two or more elements in the AST are references to the same element. If two or more elements are references to the same element, then the bindings returned by both reference and declaration nodes are identical. In the case of partial programs with ill-typed expressions, it is possible to have many ambiguous bindings that cannot be resolved. Consequently, given a code example with ill-typed expressions, the challenge is then to resolve as many bindings as possible in its code based on local information.

Whereas previous work has addressed the problem of automatically linking unresolved code elements in source code to API documentation (Subramanian et al., 2014) and recovering external libraries using oracles (Ossher et al., 2010), we address a different problem in a similar fashion. We address the problem of recovering as many bindings as possible in code examples based on both local information and a repository (oracle) of pre-acquired type information. Examples of type information include types and methods available in an existing classpath. We generalize this problem as follows:
Problem 5.1 Binding Recovery Problem. Given a Java code example $J$ with ill-typed expressions, recover the bindings that will correct these ill-typed expressions.

To solve this problem, we reduce it to the problem of finding the closest Java constructs to a list of unresolved code elements in $J$, where a Java construct is defined as a code element we can use to fix an unresolved code element in $J$. We call this find and fix process, Codepacking. Codepacking relies both on a closeness metric to match unresolved elements to Java constructs and on the Eclipse JDT’s DOM/AST modification utilities to add these constructs to the example’s code.

We use the Explanatory Power metric (Little, 2007) to determine closeness between Java constructs and unresolved elements. At a high level, the closeness of a Java construct to an unresolved element is measured as the number of matched substrings between the two. A high number of substring matches means a high explanation of the unresolved element by the Java construct.

Like previous work, Codepacking uses an oracle to ask for information (types and their methods) that can help it recover bindings automatically. The oracle is a database that contains type and method signatures as well as the labels that identify the elements in those signatures. We call this oracle the PackingSpace. More details about the definition and construction of the PackingSpace are provided in Sections 6.5.2 and 5.3.5. Unlike previous work, Codepacking uses this oracle to get candidate constructs that can repair ill-typed expressions in code examples. It does it through a series of find and fix operations.
In what follows, we describe Codepacking in the context of automatically resolving missing dependencies based on the examples’ existing code; our approach, however, can also be extended to keep programmers in the loop by generating code element recommendations that use feedback based on their preferences. This extension is left for future work.

5.2 Codepacking session

We show, through an example, how Codepacking recovers bindings in code examples. For this section only, we make certain restrictive assumptions on the code example to be packed. This section sets the stage for the complete description of Codepacking.

We look at this session from the perspective of an incomplete Java code example. This code example lacks a class declaration and contains an ambiguous syntax construct (illustrated in Figure 5.1).

```java
private static int randomizedPartition(int[] arr,
    int left, int right){
    int swapIndex = left + rand.nextInt(right - left) + 1;
    swap(arr, left, swapIndex);
    return partition(arr, left, right);
}

private static int partition(int[] arr,
    int left, int right){...}

private static void swap(int[] arr,
    int i, int j){...}
```

Figure 5.1: Incomplete code example
5.2.1 Setup: Incremental parsing

Consider a Codepacking session with a tool having the architecture illustrated in Figure 5.2. The tool takes as input a Java code example $J$ (illustrated in Figure 5.1), incrementally parses it, and then produces a packed abstract syntax tree ($PAST$). A $PAST$ is an $AST$ containing the appropriate enclosing body declarations (e.g., class, method, or both), recovered types, and the types’ dependencies. Without any loss of generality, we will refer to the $PAST$ simply as $AST$ hereafter.

To incrementally parse $J$, the tool uses an incremental parsing mechanism we built atop Eclipse’s JDT Java parser. This mechanism relies on the JDT Java parser’s input type selection and error handling to detect and record missing body declarations, type declarations, and other code declarations. Input type selection allows us to specify the type of input that the parser should consume (e.g., entire code, block). Error handling allows us to determine the missing declarations in $J$.

Based on this record, the tool generates a list of Java constructs. Given this list, the tool inspects the list’s elements to produce some integration metadata. This metadata provides additional information regarding the Java constructs to be integrated into $J$, such as access and non-access Java modifiers (e.g., static, final). For the remainder of this chapter we use the term execution plan to denote this metadata. Execution plans help us (1) surround $J$ with a class declaration, and (2) resolve missing dependencies in $J$. At a high level, this mechanism allows Codepacking to work with partial and non-compiling code examples, not just complete ones.
Surrounding $J$ with the needed body declarations is straightforward: one can easily prepend and append the appropriate code elements to wrap $J$ entirely into either a class, a method, or both body declarations. We therefore focus on the more challenging problem of resolving missing dependencies.

### 5.2.2 Resolving missing dependencies

Assume that the tool has surrounded $J$ with the appropriate class declaration named Scratched (illustrated in Figure 5.3). The next step is to resolve $J$’s missing dependencies.
A necessary step in resolving missing dependencies in any source is the identification of its missing types. Reliably identifying these missing types is not easy, as this source code might have some ambiguous syntax constructs (e.g., ambiguous references, method calls).

The code example $J$ contains an ambiguous syntax construct, specifically, an ambiguous reference. An ambiguous reference is a Java expression of the form $C.dosomething()$, where $C$ can be either a Field (in lowercase) or a Class (in uppercase) construct, and $dosomething()$ can be either an instance or a static method invocation (decided based on whether $C$ is either a Field or a Class construct). For convenience, we define $\text{typeOf}(o)$, and $\text{modifiers}(s, \text{AST})$ to return a type of code element $o$ to integrate in $J$ (e.g., field, class), and the Java modifiers of $o$'s enclosing body declaration (e.g., static, final). If $C$ is in lowercase, then $\text{typeOf}(C)$ returns Field; otherwise it returns Class.

```java
class Scratched {
    private static int randomizedPartition(int[] arr, int left, int right) {
        int swapIndex = left + rand.nextInt(right - left) + 1;
        swap(arr, left, swapIndex);
        return partition(arr, left, right);
    }

    private static int partition(int[] arr, int left, int right) {
        ...
    }

    private static void swap(int[] arr, int i, int j) {
        ...
    }
}
```

Figure 5.3: Java code with an ambiguous reference.
The tool approaches the resolution of the ambiguous reference `rand` in three steps (illustrated in Figure 5.4). Some of the elements in this figure are dimmed to indicate they are unambiguous syntax constructs. First, it locates, in the AST, the expression containing the ambiguous reference (see `rand.nextInt` in Figure 5.3). Second, it converts that expression to a signature. Third, it uses that signature to suggest the expression’s correction (with a high explanation score). A signature is a structure with a list of terms: (1) a receiver object, (2) a sequence of tokens (labels), and (3) a sequence of arguments.

For example, given the input expression

```
rand.nextInt(int)
```

with a signature `s`

```
(rand, (rand, next, int, int), (int))
```

the tool produces the following 3 results:
1. \texttt{(int, (next, int, rand, random), java.util.Random, (int))}

2. \texttt{(int, (next, int, scan, scanner), java.util.Scanner, (int))}

3. \texttt{(boolean, (has, next, int, scan, scanner), java.util.Scanner, (int))}

where each result is represented as a structure with a list of terms: (1) a return type, (2) a sequence of labels, (3) a receiver object, and (4) a sequence of arguments.

The tool evaluates each result as follows. It first creates an explanation vector from the sequences of labels of both signature and result. It then computes its vector magnitude $\mu$. The tool uses $\mu$ to evaluate how well a result explains a signature. Figure 5.5 illustrates the evaluation of each result.

$$\mu_{\text{result}_1,s}((-0.03, 0.5, 0, 1, 1, 0.5, 0)) = 1.58$$

$$\mu_{\text{result}_2,s}((-0.05, 0.5, 0, 0, 1, 0.5, 0)) = 1.22$$

$$\mu_{\text{result}_3,s}((-0.06, 0.5, 0, 0, 1, 0.5, 0)) = 1.22$$

Figure 5.5: Results evaluation (i.e., explanations of a signature).

Upon a successful evaluation, the tool picks the result with the best explanation, where the best explanation is equal to the highest vector magnitude:

$$[(\text{int, (next, int, rand, random)}, \text{java.util.Random, (int)}), \mu = 1.58]$$

The tool uses explanation vectors to decode how well generated results explain a signature. The tool creates explanation vectors as follows. If the signature's sequence of

\[ the \text{ magnitude of a vector } v = [v_1, v_2, \ldots, v_n] \text{ in a } n \text{ dimensional Euclidean space is } ||v|| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}. \]
labels contains $n$ words $w_1, w_2, \ldots, w_n$, then the explanation vector $e$ has $n + 1$ elements $e_0, e_1, \ldots, e_n$. $e_0$ represents a penalty for unmatched words. We set $e_0$ to $-0.01\kappa$, where $\kappa$ is the number of non-intersecting words between the sequences of labels of both signature and result. We ignore capitalization of words when matching elements in both sequences.

The tool sets the rest of elements $e_1, \ldots, e_n$ in $e$ to $e_i = if(\frac{x}{y} > 1) 1 else \frac{x}{y}$, where $1 \leq i \leq n$ and $x$ is the number of times a word $w_i$ appears in the result’s sequence of labels, and $y$ is the number of times $w_i$ appears in the signature’s one.

5.2.3 Reusing information from signature and its top scoring construct

Assume the tool has located an ambiguous syntax construct, converted it to a signature, searched for suitable candidates that explain it, and selected the one with the highest explanation score. Now, we specify a procedure for reusing information from both the signature and the top scoring construct. This procedure produces an execution plan (illustrated in Figure 5.6). For convenience, we define $\text{depOf}(r)$ to return the receiver object (or dependency) found in the top scoring result $r$. 
To produce an execution plan under both a signature and a top scoring Java construct, we first determine whether the code element represented by the construct is a field or a class. We then collect its required Java modifiers using J's AST. At this point, we extract the receiver object of the top scoring result. This object is the required dependency we need to resolve a code element that needs a binding. Once this information is extracted, the tool applies the generated plan, and thus fixes J. Figure 5.6 illustrates J after resolving its ambiguous reference.
import java.util.Random;

class Scratched {
    static Random rand = new Random();

    private static int randomizedPartition(int[] arr, 
        int left, int right){
        int swapIndex = left + rand.nextInt(right - left) + 1;
        swap(arr, left, swapIndex);
        return partition(arr, left, right);
    }

    private static int partition(int[] arr, 
        int left, int right){...}

    private static void swap(int[] arr, 
        int i, int j){...}
}

Figure 5.7: Java code after resolving an ambiguous reference.

5.3 Codepacking

Unlike the scenario of Section 5.2 in a real Codepacking session, a code example may contain other types of syntax ambiguity. Examples include ambiguous method calls and ambiguous inner or nested types. An example of the former could be a method call, where the method has no receiver object and no declaration in the source code. An example of the latter could be represented by the following Java expression: \texttt{an.	extbf{apple}.D}

\texttt{a = new an.	extbf{apple}.D()}, where \textit{apple} could be a package or a class.

\textit{Codepacking} handles ambiguous method calls using the same process we described in the previous section. With regards to ambiguous inner or nested types, \textit{Codepacking} uses a different method. It uses Java naming convention\footnote{Available online at \url{http://geosoft.no/development/javastyle.html}} to determine if the ambiguous type is a class or a package. For example, if “\textit{apple}” starts with an uppercase letter, and is followed by a lowercase character, then “\textit{apple}” is an inner or nested type;
otherwise it is a package. In what follows, we describe Codepacking’s main components.

### 5.3.1 The PackingSpace

To cross reference missing type information in code examples with a repository of candidate code elements, we introduce the concept of a PackingSpace (See Definition 5.1).

**Definition 5.1 (PackingSpace)** The PackingSpace is a triple \((T, F, L)\), where

- \(T\) is a set of types,
- \(F\) is a set of functions in types in \(T\), and
- \(L\) is a set of labels that identifies elements in \(T\) and \(F\).

The PackingSpace is a repository of type information. When Codepacking encounters an ill-typed code element, it uses this PackingSpace to suggest its well-typed correction.

The first element of the PackingSpace is a set of Java types \(T\). Each type in \(T\) has a unique name. Examples include `int` and `java.lang.Object`.

The second element of the PackingSpace is a set of functions \(F\). Functions represent Java constructs (e.g., methods, constructors) that we want to match against the labels sequence of some signature. We define functions in \(F\) as tuples in \(T \times L \times T \times \cdots \times T\), where the first \(T\) represents the return type of a function, the \(L\) represents its labels, the second \(T\) represents its receiver object, and the rest of the elements represent its arguments. An example of functions is illustrated in Section 5.2.
Static functions in $F$ have their receiver object prepended with the `static:` keyword. This keyword provides additional information to Codepacking regarding the code element it should pack. For example, a static Java method

```
static double ceil(double x)
```

of `java.lang.Math` is modeled as

```
(double, (math, ceil), static:java.lang.Math, (double))
```

The last element of the `PackingSpace` is a set of labels $L$. Labels are strings of characters that we used to refer to either a particular type or function in the `PackingSpace`. In other words, they represent the names of functions and types, so that we can match them against the labels in a signature. Functions in $F$ and types in $T$ can have many labels, usually consisting of synonyms. For example, a function named `perform` can have labels like `execute`, `run`, etc.

Labels in a signature are obtained by splitting the text of an ill-typed expression at capitalization boundaries, as well at spaces. Special characters (e.g., `;`, `*`) and numeric sequences are removed. The final result is then set to lowercase. For example, the expression

```
int sum = index.getCurrent() + 1
```

can be split into 5 labels:

```
(int, sum, index, get, current)
```
Labels in the PackingSpace are obtained from two sources: a list of pre-acquired abbreviations\textsuperscript{3} for types in $T$, and a list of words for Java methods and types in the classpath. We get these words by splitting types and method names at capitalization boundaries. For example, we split the method name `valueOf` into `(value, of)` labels.

5.3.2 The Explanatory Power of Results

Selecting a good closeness metric to help us determine how close a Java construct is to an unresolved code element is challenging. This metric should be intuitive to humans and easy to evaluate algorithmically. Given these conditions, we chose the Explanatory Power metric \cite{Little2007}.

Under the selected metric, we award each label $l$ in a signature’s labels sequence 1 point, and we give that point to a Java construct $c$ if that $c$ explains $l$. Specifically, $c$ explains $l$ if it matches $l$ with one of its own labels. We use explanation vectors to measure how well a Java construct explains a signature. Section 5.2 describes the construction and evaluation of explanation vectors.

5.3.3 Mapping Signatures to Java constructs

Searching the entire PackingSpace for Java constructs that can explain a signature can be an expensive operation. Consequently, Codepacking uses information stored in the signature to reduce the search space. Given the the size of the signature’s argument sequence, Codepacking searches the PackingSpace and considers only the functions

\textsuperscript{3}Extracted from http://www.javapactices.com/
in $F$ that have that same size of arguments. If the returned subset is non empty, then

`Codepacking` looks into it and then picks functions that take the signature’s same argument types. This operation reduces the returned subset. Given this reduced subset, `Codepacking` picks only the functions in that subset that match at least one of the labels in the signature’s labels sequence. The remaining elements are then evaluated on the basis of their explanatory power over the signature.

### 5.3.4 Execution Plans

An execution plan is represented as a sequence of options, which include: (1) modifiers, (2) element, and (3) dependency. These options provide `Codepacking` with additional information regarding the Java constructs that it should pack. Specifically, the modifiers option tells `Codepacking` what Java modifiers it should use for the new code element. The element option tells `Codepacking` what code element it should construct. The last dependency option tells `Codepacking` what dependency it should link to the new code element. For example, the following execution plan

```
((public), (class, Scratched), ())
```

tells `Codepacking` to surround a code example with a `public class`, named `Scratched`.

Overall, `Codepacking` supports three types of execution plans: (1) plans that wrap a source code into a body declaration, (2) plans that recover missing type declarations, and (3) plans that recover missing dependencies.
5.3.5 Codepacking Architecture

Figure 5.8 extends Figure 5.2 with the modules required to implement Codepacking functionality. Codepacking takes as input packing requests from Violette consisting of a Java code example, and a flag indicating whether the changes to be made by Codepacking should be saved in Vesperin’s database. The output is a Java code example with an approximated declarative completeness.

The incremental parser parses a Java example and then produces a set of recordings. The recordings are used to produce signatures that should be mapped to top scoring Java constructs found in the PackingSpace. The signatures and their top scoring constructs are used to produce a list of execution plans. These plans are then applied by Vesperin’s Java code transformer.

The PackingSpace is a repository of type information according to Definition 5.1. We create this repository offline, and make it available to Codepacking via Kiwi’s
API. We used information about the code elements in 4 popular APIs on StackOverflow to bootstrap it: (1) Java 1.6, (2) Guice, (3) Apache commons, and (4) Guava. Tables 5.1 shows the number of types, functions, and labels that are currently available in the PackingSpace.

<table>
<thead>
<tr>
<th># types in $T$</th>
<th># functions in $F$</th>
<th># labels in $L$</th>
</tr>
</thead>
<tbody>
<tr>
<td>6,436</td>
<td>42,164</td>
<td>751,856</td>
</tr>
</tbody>
</table>

Table 5.1: PackingSpace numbers

5.4 Evaluation

We can see Codepacking as an information retrieval problem (using local information to decide what code elements to bring back into the scratch space). Consequently, its evaluation is performed offline, focused on accuracy, and measured using precision and recall (Manning et al., 2008). We are interested in answering one question: can Codepacking correctly recover missing code elements in Java code examples?

5.4.1 Sample population

We used a sample of Java code examples to exercise Codepacking. This sample was small enough to be conveniently handled by us, yet large enough to help us evaluate Codepacking. We randomly selected 100 Java code examples from our own StackOverflow data repository (collected for the evaluation of Vesperin⁴). The selected examples cover 4 groups of APIs: (1) Java 1.6, (2) Guice, (3) Apache commons, and (4) Guava.

⁴https://gist.github.com/hsanchez/fde4103db09c865010b2
Our selected sample contains Java code examples with differing characteristics, as shown in Table 5.2: both large and small examples, and both complete and incomplete examples.

<table>
<thead>
<tr>
<th>groups</th>
<th># classes</th>
<th># methods</th>
<th># types</th>
</tr>
</thead>
<tbody>
<tr>
<td>group 1</td>
<td>14</td>
<td>74</td>
<td>157</td>
</tr>
<tr>
<td>group 2</td>
<td>5</td>
<td>54</td>
<td>119</td>
</tr>
<tr>
<td>group 3</td>
<td>23</td>
<td>73</td>
<td>165</td>
</tr>
<tr>
<td>group 4</td>
<td>12</td>
<td>62</td>
<td>158</td>
</tr>
</tbody>
</table>

Table 5.2: Evaluation sample

Java code examples (or snippets) are incomplete by nature. They may span either consecutive simple statements (e.g., assignments, function calls, throws, returns), blocks of statements (e.g., method declaration), nested classes, or all the above. We find by experience that snippets spanning both consecutive simple statements and method declarations are sometimes too difficult for incremental parsing and could lead to the wrong set of recordings. For instance, the method declaration could be misidentified by the incremental parsing mechanism as a set of consecutive simple statements. This misidentification could lead to unexpected behavior, such as enclosing the wrong code elements in a method body declaration. This issue makes the task of enclosing code examples into the right set of body declarations an important concern. We have implemented a regular-expressions-based solution in Codepacking that could help us deal with the aforementioned issue.
5.4.2 Evaluation approach

We have two primary evaluations: the recovery of body declarations, and the resolution of missing dependencies. We performed our first evaluation as follows: (1) we pick a random group from the sample’s groups as a candidate for evaluation; (2) we automatically select the files containing missing body declarations; (3) we shuffle the resulting files; (4) we choose the first file in the shuffled list; and then (5) we call Codepacking 20 times against the selected file; recording each call’s output. Each result contained the original Java code example and its updated version. We repeated this process over the remaining groups.

We performed our second evaluation as follows: (1) we pick a random group from the sample’s groups as a candidate for evaluation; (2) we automatically select the files containing both body declarations and missing dependencies; (3) we shuffle the resulting files; (4) we choose the first file in the shuffled list; and then (5) we call Codepacking 20 times against the selected file, also recording each call’s output. We also repeated this process over the remaining groups.

Given the two lists of produced results (80 results per list), we evaluated our results. Specifically, we evaluated both body declaration recovery and dependencies resolution. Then, based on our evaluation, we measure Codepacking precision (how many suggestions are valid) and Codepacking recall (how many valid suggestions have we missed). We use $F_1$-measure (harmonic mean of precision and recall) to determine Codepacking quality.
5.4.3 Results

All tests were run on iMac 2.7 GHz Intel Core i5 machine with 8 GB of RAM. Codepacking was run 20 times over four randomly selected files. Then, we recorded its output (160 results). Table 5.3 show the results of the recovery of body declarations task. Table 5.4 show the results of the resolution of missing dependencies task.

<table>
<thead>
<tr>
<th>files</th>
<th>precision</th>
<th>recall</th>
<th>F₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>(group 1) file</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>(group 2) file</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>(group 3) file</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>(group 4) file</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 5.3: Recovery of body declarations evaluation

<table>
<thead>
<tr>
<th>files</th>
<th>precision</th>
<th>recall</th>
<th>F₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>(group 2) file</td>
<td>0.98</td>
<td>0.96</td>
<td>0.97</td>
</tr>
<tr>
<td>(group 1) file</td>
<td>0.96</td>
<td>0.91</td>
<td>0.93</td>
</tr>
<tr>
<td>(group 3) file</td>
<td>0.98</td>
<td>0.94</td>
<td>0.96</td>
</tr>
<tr>
<td>(group 4) file</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 5.4: Resolution of missing dependencies evaluation

We can see that, in general, Codepacking recovers missing body declarations, as well as it resolves missing dependencies. We can see in Table 5.3 that Codepacking has a very high precision and recall, and a high quality over all the selected files.

5.5 Conclusions

In this chapter we present a practical and automated approach for recovering missing body declarations, as well as resolving missing dependencies in Java code exam-
amples. This approach is based on the formulation of the Binding Recovery problem. Our experiments support the claim that *Codepacking* is a valuable tool for repairing code examples with ill-typed expressions and missing body declarations. Moreover, these repairs guarantee access to code examples’ syntax and static semantics, which are needed by *Vesperin*’s built-in curation operations. In fact, the approach has very high precision and recall measures, as well as a very high quality over all the files it packed.
Chapter 6

Multistaging to Understand

We define the essence of a code example as a set of cohesive chunks of behavior that convey the most important aspects of the example’s intended function. The essence of a code example is related to the example’s decomposition, and thus a key factor in overall code understanding. Decomposition a code example into these chunks is called distillation. Distilling a code example’s essence is a process likely to play a significant part in code foraging.

We hypothesize that programmers who forage code on the Web have some intuitive notion of this distillation concept, and that frequently use it to get an overview of code examples’ operation. While the act of distilling code examples’ essence can be invaluable to programmers, it is still a cumbersome act \cite{Brandt et al., 2009}. This represents a problem for those programmers wishing to get a quick overview of code examples’ operation on first viewing. We address this problem in this chapter.
6.1 Accuracy versus Efficiency

When programmers find code examples online, they often go outside the Web browser in order to experiment with them inside an editor. They do that before they fully understand the included code samples (Brandt et al., 2009). This premature form of reuse might negatively affect programmers in multiple ways (Parnin and Görg, 2006), such as the loss of example context, and the weakening of the example understanding.

Although there are tools that try to bridge the gap between the Web browser and the editor to help re-establish context and improve the code example understanding (Ginosar et al., 2013; Hartmann et al., 2011), they all critically rely on humans for making distilling decisions. The rationale is that humans are better and more accurate at making these types of decisions. The downside of relying entirely on humans is that the distillation process is still time-consuming. This clashes with our goal of making the distillation process quick and easy. If we can automate the distillation process, right at the origin Web page, and then present the distilled code, then we can help programmers complete their source code understanding tasks quickly and accurately.

One way to speed up the distillation process is through multistage code examples (Robins et al., 2003). Multistage code examples have their functionality broken into discrete chunks of behavior (code stages). Each code stage is self-contained and builds upon, and in relation to, preceding code stages. All together, they provide an optional and yet easily accessible roadmap of the code example. By following such a roadmap, their learners can complete their code understanding tasks quickly and accurately.
6.2 Our approach

We approach the distillation process by automatically multistaging code examples. Specifically, we decompose a code example into a series of discrete chunks of behavior. This decomposition continues until the point when a chunk cannot be further decomposed. Non-Decomposing chunks of behavior are called *prime* subsets of behavior. These chunks are self-contained, and thus can be accessed or explored non-sequentially by their learners to steer understanding.

Following the distilled code suggests a form of code inspection we call Multistaging to Understand. By adopting this form of inspection, we can guide programmers towards specific units of functionality that might be of interest. Programmers can read and explore these units in any order; enabling some form of exploratory learning (Carroll 1990). Similar to other code reading techniques (Linger et al. 1979), Multistaging to Understand can be valuable if the code is poorly specified, as it is often the case for online code examples (Sanchez and Whitehead 2015). Unlike these code reading techniques, the identification of all of the prime subsets of behavior is done automatically, based on the source code's content.

The key problem in content-based multistaging is determining the minimal set of declarations (e.g., used methods, classes, fields) that are required for each code stage to compile properly. This problem is magnified by the incomplete nature of code examples and their ambiguities (Dagenais and Robillard 2012). We use an algorithm called MethodSlicing for decomposing code examples into an ordered set of code stages.
The algorithm doesn’t place any limitations on the completeness of the example’s code. The only limitation is that the code to be multistaged must be written in Java.

Whereas previous work has considered authoring multistage code examples using either direct editing and editable code histories (Ginosar et al., 2013), record and replay of code-based tutorials (Kojouharov et al., 2004), or annotated code tours that highlight important code locations (Oezbek and Prechelt, 2007), we propose something different. We propose that code stages can be algorithmically generated by statically analyzing the examples’ Java code.

### 6.3 Multistaging to Understand Session

The idea behind Multistaging to Understand is that, as the programmers inspect a few generated code stages, their functionality is mentally abstracted, and then combined to understand the intended function of a code example. This section walks the reader through a Multistaging to Understand session (summarized in Figure 6.5). In the remainder of this section, we will refer to the programmer using our technique simply as Bella.
Figure 6.1: Searching for a code example that shows how to find the smallest number in an array without using sorting.

Figure 6.1 shows the first step. This step takes as an input a query describing Bella’s information need. Bella then issues this query to the Google search engine 1. After issuing the query, the search engine comes back with a set of generated results matching the query (see Figure 6.2). At this point, Bella eagerly selects the first result. She does that because she wishes to get a quick overview of the result’s operation. Figure 6.3 shows this Java code example 2.

1Bella can use any Web search engine to complete this step, such as DuckDuckGo, Bing, etc.
2This code example is available online at http://stackoverflow.com/q/29802290/#29802635.
Figure 6.2: Generated results by Google search engine. Programmer selects first one.

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class SmallestNum {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter size of array:");
        int size = sc.nextInt();
        List<Integer> numbers = new ArrayList<Integer>[
            For (int i = 0; i < size; i++)
            numbers.add(sc.nextInt());
            }
        System.out.println("Search value");
        int n = sc.nextInt();
        if (n >= 0 && n < size) {
            if (n < size) {
            System.out.println("Minimum value is ", min); System.out.println("Minimum value is ", min2);
            } else {
            System.out.println("Invalid entry");
            }
        }

    private static int select(List<Integer> list, int left, int right, int n) {
        int pivotIndex = getMedianIndex(list, left, right);
        if (pivotIndex == n - 1) {
            return list.get(pivotIndex);
        } else {
            return select(list, left, pivotIndex - 1, n);
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Find smallest integer value in array list in Java without..." );
        Dec 14, 2013 - Find smallest integer value in array list in Java without Arrays.sort. No problem. Just to be safe when you start using doubles or floats.

        Java program to find the largest & smallest number in an array without using sorting.
        stackoverflow.com/.../Java-program-to-find-the-largest-smallest-number...
        Feb 21, 2014 - I could get the largest without using arrays but, unable to get the... public static void main(String[] args) { int smallest = 0; int large = 0; int num;... Channel.com/.../Java-find-minimum-and-maximum...
        May 21, 2010 - That approach used Arrays but the reader wanted to find largest and smallest values from a group of numbers without using Arrays. So here is...

        Figure 6.3: Finds the smallest number in an array without using sorting.

        Bella uses Violette to create a new scratch space for the found code example.
```
This is only possible due to it being hosted on StackOverflow. Then, she presses the Stage button. Violette responds to this action by asking Kiwi to multistage the code example. Kiwi comes back with 3 generated code stages (illustrated in Figure 6.4).

![Figure 6.4: Generated code stages (green buttons).](image)

After reading the labels of the generated code stages, Bella builds a general hypothesis about the nature of the code example. Using the fact that code stages are self-contained and can be accessed in any order, Bella clicks the first code stage that comes to her mind. This is the Get Pivot Index code stage (see Figure 6.5a).

While looking at this code stage, Bella skims the method signature of the getPivotIndex method. Then she starts refining her general hypothesis about the code example. She hypothesizes that getPivotIndex’s function is to get an index between two indexes (i.e., left and right).

With this new hypothesis in mind, she inspects the visible code blocks in the
code stage (see Figure 6.5a), opening any hidden code blocks as she goes along. The first hidden block she opens is the one located at line 7 in Figure 6.5b. The logic inside this block is responsible for incrementing the left index as long as this index is less than the pivot value (calculated in line 5). She uses this information to deduce the function of the next hidden code block, located at line 8 in Figure 6.5b. The elements in this block are responsible for navigating a list of integers from right to left.

After having verified the function of the two code blocks, Bella notices certain similarities between the `getPivotIndex` method and the Quicksort’s `Partition` method. She uses this information to guess the function of the next hidden code block, which starts at line 11 in Figure 6.5c. The function of this hidden code block is to swap elements in a list of integers only if the left index is less than the right index.
At this point, Bella is getting the hang of inspecting code stages. Consequently, she approaches the Select code stage in the same way (see Figures 6.5d, 6.5e, and 6.5f).

After having inspected the Select code stage, and learned its function, she feels she has achieved her desired level of compression. As a result, instead of inspecting the Main code stage, she combines all her gained knowledge to determine the function of the entire code example. The function is to find the kth smallest element in a list using continuous list partitioning and careful recursive calls. She now thinks she can use this example in her own work. This ends the Multistaging to Understand session.

6.4 The Multistaging Problem

The goal of multistaging code examples is to reveal segments of code in an easily understood sequence. We generalize this problem as follows:

**Problem 6.1 The Multistaging Problem.** Given a code example’s packed abstract syntax tree (AST), with a set of n method declarations $D = D_1 \cup D_2 \cdots \cup D_n$, compute an ordered set of code stages $\{ S \mid S \subseteq D \times D \}$, such that each code stage $s \in S$ builds upon, and in relation to, preceding code stages; i.e., $s_i \leq s_j, s_i$ precedes $s_j$, where $i, j = 1, \ldots, |S|$.

In this problem, the first code stage of a Java code example always lacks a preceding code stage. As such, without loss of generality, we add a special null code stage to the set $S$; called $s_\emptyset$. The preceding code stage of $s_\emptyset$ is itself.

Unlike the work in [Ginosar et al., 2013], we consider a code stage as a group
of code fragments that captures a prime subset of the behavior found in the Java code example. Whenever a code stage is generated, a composition relationship is established between the new code stage and a previous code stage. For example, the Select code stage (see Figure 6.5f) contains a method from the Get Pivot Index code stage (see Figure 6.5a). We generalize these insights using the notion of Code Stages (See Definition 6.1). This definition simplifies the multistaging process, as we describe it later in this section.

**Definition 6.1** Code stages are a set of subsets of behavior found in a code example, such that

- Each subset contains a method or a set of collaborating methods.
- Each subset builds upon, and in relation to preceding subsets.
- Code stages enumerate all the subsets with the above properties.

We describe an algorithm for multistaging Java code examples in Section 6.4.1. To facilitate presentation, we consider the same Java code example used in Section 6.3.

Figure 6.7 illustrates the computed Code Stages, sorted in ascending order by code stage length. For the remaining sections, we make certain restrictive assumptions on the Java code examples input; e.g., the code example is complete and syntactically correct, and all of its members are included in the Java Standard Edition 6 API specification.

---

3 Methods collaborate with other methods using the method invocation mechanism.  
4 See [http://docs.oracle.com/javase/6/docs/api/](http://docs.oracle.com/javase/6/docs/api/) for the full API specification.
6.4.1 MethodSlicing

In this section, we introduce MethodSlicing. MethodSlicing is an algorithm for solving the Multistaging Problem (described in Problem 6.1). At a high level, MethodSlicing takes an AST as input, statically analyzes it, and then slices it into Code Stages. Based on the Definition 6.1, the Code Stages are modeled as a set of tuples containing one or more collaborating methods. Figure 6.6 describes MethodSlicing.

**Input:** AST \( p \), generated for a Java code example  
**Output:** Code Stages, sorted in ascending order by code stage length  
**Algorithm** MethodSlicing\((p)\)

\[
\text{let } S ← ∅ ∪ \{s_∅\}
\]

\[
\text{for each method } m ∈ p \text{ do}
\]

\[
\text{let } B ← ∅ ∪ \text{GetAllAstNodeBindings}(m)
\]

\[
\text{let } d ← \text{JDT.FindAstNodeDeclarations}(B)
\]

\[
\text{let } s ← \text{ReconstructSourceCode}(p, d)
\]

\[
S ∪ \{s\}
\]

end

sort \( S \) in ascending order  
**return** sorted \( S \)

end

Figure 6.6: Pseudocode for MethodSlicing.

While the general algorithm may apply to any object-oriented language, some of the details (and used libraries) are tied to the specifics of the Java language. For instance, we use the Eclipse Java Development tools\(^5\) (JDT) to parse the Java code, as well as to manipulate and reconstruct Java code via its DOM/AST utilities.

\(^5\)Access it online at [http://www.eclipse.org/jdt/](http://www.eclipse.org/jdt/)

103
Figure 6.7: One application of MethodSlicing against the SmallestNum code example.

MethodSlicing uses two subroutines (illustrated in Figure 6.8 and Figure 6.9): GetAllAstNodeBindings and ReconstructSourceCode. GetAllAstNodeBindings subroutine collects binding information of AST nodes. This subroutine (see Figure 6.8) uses the Eclipse JDT’s traversal utilities to walk the AST node representing a method declaration. During this walk, it collects the binding information for variables, fields, parameters, packages, methods, and inner class (or nested) declarations used within that method declaration. The declarations corresponding to these bindings are then obtained by calling JDT’s FindAstNodeDeclarations subroutine. Given these declarations and the entire AST, ReconstructSourceCode subroutine (see Figure 6.9) uses Eclipse JDT’s
DOM/AST utilities to reconstruct the source code for a generated code stage.

**Input**: AST Node $p$

**Output**: Set of bindings in $p$

**Function** `GetAllAstNodeBindings(p)`

```plaintext
let $V, S, R \leftarrow \emptyset$  // visited, stack, result
let $W \leftarrow \{$target node types$\}$
$S \cup p.root$

while $S$ is not empty do
    let $u \leftarrow$ pop $S$
    continue unless $u \notin V$
    $V \cup \{u\}$
    for each node $w \in JDT.Children(u)$ do
        $R \cup \{JDT.ResolveBinding(w)\}$ if $w \in W$
        $S \cup \{w\}$
    end
end
return $R$
```

Figure 6.8: Pseudocode for `GetAllAstNodeBindings`.

**Input**: AST Node $p$, and declarations $d \in p$

**Output**: Reconstructed source code

**Function** `ReconstructSourceCode(p, d)`

```plaintext
// delete nodes \{p \d d\} from AST
let $p' \leftarrow JDT.DeleteAstNodes(p, \{p \d d\})$
return $JDT.GetSourceCode(p')$
```

Figure 6.9: Pseudocode for `ReconstructSourceCode` subroutine.

We model binding information as a set of $n$ tuples $T$ made of name-and-ASTNodeType pairs; e.g., (name, nodetype)$_1$ ... (name, nodetype)$_n$, where $n$ is the number of named entities in a Java code example. For example, we show the set of global bindings $U$ for our input code example here:

105
MethodSlicing iterates over the methods in an AST in the exact order they were written in the code example. For each method, it collects the binding information of its children, as well as the binding information of collaborating methods. It then specifies the content of a code stage (see Figure 6.6) by getting the AST node declarations of each method’s local bindings $B$. The elements for the `getPivotIndex` method are:

$$B = \{(List, \text{package}), (SmallestNum, \text{type}), (getPivotIndex, \text{method}), (left', \text{parameter}), (right', \text{parameter}), (list', \text{parameter}), (pivot, \text{variable}), (temp, \text{variable})\}$$

The above elements represent the content of Get Pivot Index code stage. This code stage includes the `SmallestNum` class declaration, the `getPivotIndex` method.

---

6 The ordering of the methods in the AST is irrelevant to the algorithm, as the algorithm always converge to a unique set of code stages that are sorted in ascending order by code stage length.
declaration, and its children. See the reconstructed source code of the Get Pivot Index code stage in Figure 6.7.

MethodSlicing’s divide and conquer approach for distilling Java code examples’ essence can be beneficial during code comprehension tasks. However, there is one caveat that can hinder its effectiveness: produced code stages might consist of long methods\footnote{These are methods, which size is getting close to or beyond 15 lines of code.}. Long methods tend to take more time to understand than small methods\footnote{Mäntylä et al., 2003}. Consequently, code stages with long methods (i.e., large code stages) might be difficult to digest by those programmers wishing to obtain an overview of the given code example at first glance.

To address this issue, we investigated the obstacles programmers faced when inspecting the large code stages produced by MethodSlicing. We discovered that most of the issues were related to the navigation of the code stages’ content. One way to deal with these issues is via code folding. The efficiency of code folding on browsing tasks was validated by Cockburn et al.\cite{Cockburn:2003}.

In line with our findings, we extended MethodSlicing to automatically reduce large code stages (via code folding) whenever possible. Reduction in MethodSlicing shows the code elements that are most informative (i.e., code elements with a high usage score) in each code stage and hides (i.e., folds) the ones that are less informative on first viewing. However, despite their invisibility, these hidden elements are easily accessible if one chooses to see them. This technique is described in the next section.
6.4.2 MethodSlicing with Reduction

As generated code stages can vary in size, large code stages might occur. It is often difficult for programmers to identify the relevant information in those types of code stages. We can reduce this information-overloading problem by automatically reducing them. The rationale is that reduced code stages can be easily digested by those programmers wishing to obtain an overview of the given code example.

We make reduction decisions in MethodSlicing entirely based on examples’ source code structure. Our approach is consistent with how human abstractors approach inspecting unfamiliar source code. When inspecting an unfamiliar source code, they extract code fragments according to the hierarchical structure of control flow units present in the source code (Détienne and Bott 2002; Pennington 1987). This structure can be described as a series of interconnected code blocks. Each code block has an associated usage score. We compute the usage score of a code block using Equation 6.1. The usage score of a code block is representative of the demand of its elements throughout the code example. The usage frequency of each element in code block is the number of times this element appears in a code stage. As a result, we use the usage score of code blocks to show the blocks with a higher demand and hide the ones with a lesser demand.

\[
\text{UsageScore}(B) = \frac{\sum_{\text{elem} \in B} \text{UsageFreq}(\text{elem})}{\text{TotalChildren}(B)}
\] (6.1)

For example, given a nested code block at line 11 in Figure 6.5c, we first collect its children: temp, list, left, and right. Second, we compute each child’s usage frequency:
2, 7, 10, and 9. Lastly, we put it all together and calculate the nested code block’s usage score: \((2 + 7 + 10 + 9)/4 = 7\).

We cast the problem of reducing large code stages as an instance of the Precedence Constrained Knapsack Problem or PCKP \cite{samphaiboone2000}. This problem is specified herein.

**Problem 6.2 Code Stage Reduction.** Given a set of code blocks \(B\) (with weight \(w_b\) and profit \(p_b\) per block \(b \in B\)), a Knapsack capacity \(W\), a precedence order \(O \subseteq B \times B\), and a set of constraints \(C\), find \(H^*\) such that \(H^* = B \setminus X^*\), where \(w_b = \) number of lines of code in \(b\), \(p_b = \) UsageScore\((b)\), \(X^* = \) arg max \(\{\sum_{b \in B} p_b\}\), and \(X^*\) satisfies the constraints in \(C\). The constraints in \(C\) include: \(\sum_{b_j \in B} w_{b_j} \leq W\), where \(b_i \rightsquigarrow b_j\) \((b_i \text{ precedes } b_j) \in O\), and \(i, j = 1, \ldots, |B|\).

Similar to Samphaiboon et al. \cite{samphaiboone2000}, we solve this problem by using dynamic programming. Our solution generalizes the code stage reduction problem, also taking into account a precedence relation between code blocks in a code stage. We build a Directed Acyclic Graph (DAG) to represent such a relation, where nodes correspond to code blocks in a one–to–one fashion. This relation is expressed as a composition relation between code blocks. For instance, a code block \(k - 1\) precedes a code block \(k\), if code block \(k - 1\) contains the code block \(k\). We build this DAG when traversing a code stage’s AST. We extend JDT’s ASTVisitor to visit all of the code block nodes in the AST. Specifically, this visitor traverses an AST, adds all the visited code block nodes to a DAG, and then returns a non-empty DAG.
In this DAG, each visited code block has a profit (e.g., the usage score of a code block) and a weight (e.g., the number of lines of code in the code block). Our solution’s Knapsack has a fixed capacity (i.e., total number of lines of code to be displayed in the reduced code stage). So, given a code stage and a capacity, our solution automatically reduces a code stage. It does it by identifying the location of non essential code blocks; i.e., those code blocks that if added to the solution would exceed the fixed Knapsack’s capacity (see $H^*$ in Problem 6.2). The value of this capacity is tunable. We selected it based on feedback from our user studies.

**Input:** AST Node $p$, and declarations $d \in p$

**Output:** A tuple consisting of the reconstructed source code and $H^*$

**Function** `ReconstructSourceCode($p$, $d$)

// delete nodes $\{p \setminus d\}$ from AST
let $p' \leftarrow JDT\text{.DELETEASTNODES}(p, \{p \setminus d\})$

let $DAG_{p'} \leftarrow$ traverse $p'$ and then get built DAG

let $H^* \leftarrow$ computes $B_{p'} \setminus X_{p'}^*$ using $DAG_{p'}$ and a capacity of 15 LOC

return $(JDT\text{.GETSOURCECODE}(p'), H^*)$

end

Figure 6.10: Pseudocode for updated `ReconstructSourceCode`. This subroutine returns a tuple comprising the reconstructed source code and the code elements to hide.

We extend `ReconstructSourceCode` to include the code stage reduction step.

---

*Code blocks enclosing other code blocks have their weight calculated and distributed among their enclosing code blocks. For example, if a code block $A$ surrounds two code blocks $B$ and $C$, then $w_A = w_{A_{original}} - (w_B + w_C)$. #110
Figure 6.10 sketches this step. Our solution is based on the following recursive formula:

\[
X^*[k, w] = \begin{cases} 
X^*[k - 1, w] & w_k > w \\
\max(X^*[k - 1, w], X^*[k - 1, w - w_k] + p_k) & w_k \leq w \land k - 1 \Rightarrow k
\end{cases} 
\tag{6.2}
\]

This recurrence (see Equation 6.2) determines the set \(X^*\), which has a total weight \(w\), where \(w \leq W\). In the first case, a block \(k\) cannot be part of the solution since the total weight will be greater than \(w\). In the second case, a block \(k\) can be in the solution, and we choose the case with greater value only if there is an edge between a previously chosen block \(k - 1\) and the current block \(k\).

![Figure 6.11: Reduced large code stages. Knapsack capacity is 15 (LOC). Uninformative elements are hidden using code folding. The ... symbol represents hidden areas.](image)

(a) Reduced Code stage 1.  (b) Reduced Code stage 2.  (c) Reduced Code stage 3.

We use the same code example of Section 6.4 as input for MethodSlicing with Reduction. The Knapsack capacity is 15 lines of source code (LOC). We illustrate its output in Figure 6.11. This figure shows smaller and nicely decomposed code stages. In the next section we describe the architecture that implements our multistaging ideas.
6.5 Multistager Architecture

Figure 6.12 shows the architecture of our Code Example Multistager. This multistager implements the MethodSlicing with Reduction algorithm. Indicated in this figure are two execution paths of the multistager, and its relation with Vesperin (Sanchez and Whitehead, 2015) and its two main components: Violette and Kiwi.

We distributed the overall functionality of our multistager between Violette and Kiwi. Violette is its front-end, while Kiwi is its back-end. Consequently, we use both JavaScript (Violette) and Scala (Kiwi) to implement it.

6.5.1 Multistaging Requests

At any time, during a Java code example’s inspection, programmers may use this multistager through Violette’s interface. Programmers can press the Stage button.
Violette follows this action by issuing a remote call to Kiwi requesting the example’s Java code to be decomposed into an ordered set of code stages. Kiwi reacts to this call by first generating an AST for the code example. Then, it multistages it with code stage reduction in mind. Lastly, it ships the generated code stages back to the caller. In a non-error path, Kiwi’s reply contains the ordered set of code stages (including $H^*$). Otherwise, it contains a set of warnings describing a failed multistaging attempt.

6.5.2 Packing Code examples

To produce a packed AST, which we use to correctly multistage code examples, we use Codepacking. Briefly, Codepacking is a technique we developed to help us “pack” incomplete and often ambiguous code examples containing missing code elements. Once these elements are resolved and then recovered, Codepacking produces a packed AST. More information with regards to Codepacking is found at Chapter 5.

6.6 Conclusion

In this chapter we present a practical and automated approach for distilling the essence of Java code examples on StackOverflow. This approach is based on the formulation of the Multistaging Problem. Our experimental results described in the next chapter support the claim that our approach is a valuable tool for understanding code examples where most of the code is non-localized, but has only minor benefits when code is partially or fully delocalized. The technique provides consistent speed improvements, irrespective of delocalization.
Chapter 7

Evaluating Multistaging to Understand

In this chapter, we evaluate Multistaging to Understand. We assume that one of the factors that influences comprehension time and accuracy in Java code examples is code size. Code size dictates how long a programmer spends inspecting an example’s Java code. Consequently, it’s important to test Multistaging to Understand using code examples with varying sizes.

We have two null research hypotheses:

\( H_{0,\text{accuracy},X,Y} \). Multistaging to Understand does not increase the accuracy of answers to questions regarding abstractions \( X \) for code examples of size \( Y \).

\( H_{0,\text{reviewing-time},Y} \). Multistaging to Understand does not reduce the reviewing time of code examples of size \( Y \).

The rejection of \( H_{0,\text{accuracy},X,Y} \) and \( H_{0,\text{reviewing-time},Y} \) hypotheses would lead to the acceptance of two alternative hypotheses, which state: (1) Multistaging to Under-
stand increases comprehension accuracy with regards to the creation of abstraction \( X \) for code example of size \( Y \); and (2) Multistaging to Understand reduces reviewing time of code examples of size \( Y \).

Our experimental set-up is exploratory. Based on Du Bois et al. (2005), we assume a significance criterion \( \alpha \) of 0.10. This indicates a 10% probability of concluding that a group difference exists when there is no actual difference. This has to be kept in mind when considering the results.

### 7.1 Experimental Design

12 participants were recruited using Upwork.com, a popular crowdsourcing service company. Subjects had previous experience with the Java programming language, used code foraging, were familiar with StackOverflow, and had limited experience with code inspections.

The subjects were split into two groups (control and treatment) of 6 participants, where each group contained subjects with the same mix of abilities (based on their Java experience stated in their professional profiles at Upwork). Each group then focused on one technique, and was aware of the other technique, since the experimental processing was previously explained to them. Specifically, the treatment group applied Multistaging to Understand, while the control group applied the Read to Understand technique—a technique that involves reading a given source code to steer understanding.

We used a crossed factorial design with two factors to test the Multistaging to
Understand technique. Table 7.1 illustrates the details of this design. We use Multi-staging to Understand and Read to Understand as the between-subjects factor. Then, we use the Java code examples’ size (short, medium, long) as the within-subjects factor, where (1) the size of a short code example is between 35 and 70 lines of code (LOC); (2) the size of a medium code example is between 70 and 140 LOC; and (3) the size of a long code example is greater than 140 LOC. We exposed the participants in any group to all code example sizes, and applied the assigned inspection technique during each run.

We have two independent variables and two dependent variables. Our independent variables are the program comprehension technique, and the size of Java code examples (short, medium, long). Our dependent variables are the response accuracy, and the code example reviewing time.

To control extraneous factors, such as systematic performance differences (e.g., Java programming experience) between the treatment and control groups, we formed blocks of participants with similar levels of Java programming experience. Specifically, each group had 2 novice, 2 proficient, and 2 expert Java programmers. This setting assures two things. First, the overall performance of the two groups is expected to be equal. Second, each participant in a group has a counterpart with similar abilities in the other group.
<table>
<thead>
<tr>
<th>Size</th>
<th>Multistaging to Understand</th>
<th>Read to Understand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short</td>
<td>Group 1 – run 1</td>
<td>Group 2 – run 1</td>
</tr>
<tr>
<td>Medium</td>
<td>Group 1 – run 2</td>
<td>Group 2 – run 2</td>
</tr>
<tr>
<td>Long</td>
<td>Group 1 – run 3</td>
<td>Group 2 – run 3</td>
</tr>
</tbody>
</table>

Table 7.1: Crossed Factorial Design

7.2 Experimental Materials

7.2.1 Code Examples

Java was the language chosen for the code examples, being both the language with which the subjects had the most experience and the only language supported by *Vesperin*. We used 3 Java code examples:

1. [http://stackoverflow.com/q/26818478#26819260](http://stackoverflow.com/q/26818478#26819260)

2. [http://stackoverflow.com/q/14210307#14210519](http://stackoverflow.com/q/14210307#14210519)

3. [http://stackoverflow.com/q/5317329#5843759](http://stackoverflow.com/q/5317329#5843759)

The 3 Java code examples presented to the subjects varied in size. They were approximately 50 LOC, 135 LOC, and 200 LOC respectively, and needed to be inspected in 120 minutes.

7.2.2 Program Comprehension Questions

Our program comprehension questions for tasks 1, 2 and 3 are displayed in Figures 7.1, 7.2 and 7.3 respectively. These questions are also available online at
These questions are open-ended questions and are variants of closed-ended questions proposed by [Pennington (1987)](http://hsanchez.typeform.com/to/rzloop); one for each program comprehension aspect: (1) Function, (2) Operations, (3) Control Flow, (4) Data Flow, and (5) State. Tables 7.2 and 7.3 present a list of typical generic questions for evaluating the different abstractions of Pennington’s model, and their open-ended variants respectively.

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>Program Comprehension Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Flow</td>
<td>What is the sequence of execution? <em>Example:</em> Is the LOCAL-FILE read before the REMOTE-FILE is read?</td>
</tr>
<tr>
<td>State</td>
<td>What is the content of a data object at some point of execution? <em>Example:</em> What is the value of field y after exiting for-loop?</td>
</tr>
<tr>
<td>Operations</td>
<td>What does the code compute? <em>Example:</em> Is the NEGATIVE-INFINITY value compared to min.</td>
</tr>
<tr>
<td>Data flow</td>
<td>What does the code compute? <em>Example:</em> Is the detailed tax amount computed?</td>
</tr>
<tr>
<td>Function</td>
<td>What is the overall functionality? <em>Example:</em> Does the program accumulate account totals for each sales representative?</td>
</tr>
</tbody>
</table>

Table 7.2: Closed-ended program comprehension questions for evaluating abstractions.
Abstraction | Open-Ended Program Comprehension Questions
--- | ---
Control Flow | Describe a given sequence of execution using pseudo-code.
State | Describe accurately the composition of the data object at some point of execution.
Operations | Describe the need of data object in a sequence of execution.
Data flow | Describe when the value of a data object gets updated in a sequence of execution.
Function | Describe what functionality is implemented in a sequence of execution.

Table 7.3: Open-ended program comprehension questions for evaluating abstractions.

Some of the questions we asked in each task are implicitly dependent. That is, a question evaluating a single abstraction can implicitly evaluate other abstractions. For example, a question evaluating knowledge on the Control Flow abstraction can implicitly evaluate some knowledge on the Operations abstraction. Nonetheless, this dependency is benign, as it would not hinder the access of any abstraction already present in the participant’s mental representation.
General Questions

1 Which group did you get assigned to?*

☐ Group A
☐ Group B

2 How long did you spend on the task?*

Desired time format when answering this question: hr_min_sec

Comprehension Questions

3 [Data Flow] Describe what is added to (and removed from) response Array field in the `challengeReceived` method, and how this field contributes to the overall source code functionality.*

4 [Function] Describe the logic implemented in the `challengeReceived` method. *

5 [Operations] Describe the need for the variable parameters of type Map in the `challengeReceived` method. This variable can contain nullable values for certain keys *

6 [Control Flow] Using pseudo-code, write a summary of how a challenge object of type String gets received. *

*DO NOT simply refer to syntactical operations, but describe specific semantics of the overall code example.

Desired answer format: (1) text; (2) text; (2.1) text; (3) text; ...

7 [State] Look at the statement "return buf.tostring();" in method `convertToHex`. Describe the value of variable `buf` after its evaluation in this statement.*

Figure 7.1: Comprehension questions for task 1.
Figure 7.2: Comprehension questions for task 2.
General Questions

1. Which group did you get assigned to?*
   - Group A
   - Group B

2. How long did you spend on the task?*
   Desired time–format when answering this question: __hr___. __min__. __sec__

Comprehension Questions

3. [Function] Describe the overall functionality implemented in the code example. *

4. [Data Flow] Describe when the value of the field movieTitles is updated. *

5. [Control Flow] Using pseudo-code, write a summary of how the doInBackground method in class RequestTask performs its main functionality. *
   
   DO NOT simply refer to syntactical operations, but describe specific semantics of the doInBackground method.
   
   Desired answer format: (1) text; (2) text; (3) text ...

6. [State] Describe the composition of the value of variable movieTitles when used in the last statement of method onPostExecute in class RequestTask.*

7. [Operations] Describe the semantics of the onPostExecute found in the class RequestTask.*

Figure 7.3: Comprehension questions for task 3.
7.2.3 Response Accuracy Rating Scheme

We use Du Bois’s rating scheme (Du Bois et al., 2005) to evaluate answers to our open-ended questions. We chose this rating scheme because it could identify objective differences in response accuracy of open-ended questions. We rated the response accuracy of our open-ended questions in four categories:

- **Correct Answer.** The participant answered the question correctly.
- **Almost Correct Answer.** The participant answered the question almost correctly. Minor details were missed or minor details were wrong.
- **Right Idea.** The participant gave a half-baked answer. Important details were missed.
- **Wrong Answer.** The participant answered the question incorrectly.

There is a clear difference in distance among the above types of answers. Consequently, Du Bois’s rating scheme rated a correct answer as 10, an almost correct answer as 8, a right idea as 5, and a wrong answer as 0 respectively. Moreover, the rating scheme makes no distinction among responses in the same category as this could lead to highly subjective differences.

All participants were graded randomly and anonymously in order to minimize bias. Once all the assessments were completed, the results were sorted back into the respected groups.
7.3 Experimental Procedure

We evaluated our technique based on the 5 program comprehension aspects mentioned in Section 7.2.2.

Participants were introduced to our experimental procedure at the start of the experiment. We also gave them an overview of the goals and guidelines for their assigned comprehension technique. We asked them to spend their time wisely, stay only at the code example’s Q&A page, and stick to the assigned task. Tables 7.5 and 7.4 illustrate the goals and guidelines provided to each group. This information is also available at http://huascarsanchez.com/posts/thesiswork/tools/evaluate-answer and http://huascarsanchez.com/posts/thesiswork/tools/violette-multistage.
Goal:

- Gain understanding in
  - Control flow. What is the sequence of execution?
  - State. What is the content of a data object?
  - Operations. What does the code compute?
  - Data flow. Where does a data object get updated?
  - Function. What is the overall functionality?

Guidelines:

1. Spend your time wisely; stay only at the code example’s Q&A page.
2. Click on the stage button to generate an ordered set of code stages.
3. Review the name of the code stages in the order they appeared.
4. Iteratively click on any of the generated code stages to introduce a subset of the given Java code example’s behavior. E.g., Click stages in any order.
5. Review the introduced subset of behavior to make sure you understand what it is actually doing.
   1. Open any hidden code blocks as you go along, and then try to understand its elements

Table 7.4: Guidelines for the Multistaging to Understand experiment
Goal:

- Gain understanding in (without *Multistaging to Understand*)
  - Control flow. What is the sequence of execution?
  - State. What is the content of a data object?
  - Operations. What does the code compute?
  - Data flow. Where does a data object get updated?
  - Function. What is the overall functionality?

Guidelines:

1. Spend your time wisely; stay only at the code example’s Q&A page.
2. Gain understanding in the asked abstractions, such as control flow, function, data flow, state, and operations.

Table 7.5: Guidelines for the Reading to Understand experiment

The experiment comprises three program comprehension tasks. Each task was divided into two parts: A multistaging/code reading part and an answering comprehension questions part. After the introduction and overview, we asked the subjects to start the tasks.

7.3.1 Discussion of Results

We discuss in this section the results of our experiment. Specifically, we discuss the group differences with respect to our two research hypotheses.
7.3.1.1 $H_{0, accuracy, X, Y}$, Response Accuracy

Contained within Table 7.6 are the results of our analysis concerning the response accuracy for both groups. We use “number – number p-value=number” as the format to represent the results in each cell, where the first number represents the average response accuracy of the Multistaging to Understand group, the second number the average response accuracy of the Read to Understand group, and the third number the p-value for the one-sided paired t-test. Since our null hypotheses are directional, we use these t-tests to verify whether the Multistaging to Understand group show higher accuracy in their answers to our comprehension questions or shorter reviewing times than the Read to Understand group.

<table>
<thead>
<tr>
<th>Short</th>
<th>Medium</th>
<th>Long</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong></td>
<td>6.83 - 3.33</td>
<td>7.17 - 3.83</td>
</tr>
<tr>
<td>p=0.0037</td>
<td>p=0.0509</td>
<td>p=0.0534</td>
</tr>
<tr>
<td><strong>Control Flow</strong></td>
<td>8.50 - 6.83</td>
<td>7.17 - 4.33</td>
</tr>
<tr>
<td>p=0.0525</td>
<td>p=0.1984</td>
<td>p=0.0204</td>
</tr>
<tr>
<td><strong>Data Flow</strong></td>
<td>8.67 - 6.17</td>
<td>5.33 - 3.00</td>
</tr>
<tr>
<td>p=0.0462</td>
<td>p=0.2308</td>
<td>p=0.1199</td>
</tr>
<tr>
<td><strong>State</strong></td>
<td>8.67 - 7.00</td>
<td>7.67 - 5.67</td>
</tr>
<tr>
<td>p=0.0873</td>
<td>p=0.1594</td>
<td>p=0.0971</td>
</tr>
<tr>
<td><strong>Operations</strong></td>
<td>7.33 - 3.33</td>
<td>7.83 - 4.83</td>
</tr>
<tr>
<td>p=0.0595</td>
<td>p=0.0609</td>
<td>p=0.0549</td>
</tr>
</tbody>
</table>

Table 7.6: Average Response Accuracy

Table 7.6 shows notable differences in response accuracy averages between the groups, completely favoring the Multistaging to Understand group, but some of the $p$ – values were still high for the medium code example. Information gathered from subjects showed there were multiple wrong answers in both groups (for the medium code example). Moreover, it showed there was a higher frequency of wrong answers in
the control group than in the treatment group. This higher frequency of wrong answers in the control group caused a significant variation, resulting in these high \( p \)-values.

We investigated the obstacles subjects faced when inspecting the medium code example. By interviewing the subjects, we discovered that most of the experienced issues were related to deducing the intent of a few delocalized methods in the code example (Letovsky and Soloway 1986). The reason why this appears in the medium code example is likely by chance. Delocalization appears when a particular goal is implemented by lines of code that appear in spatially disparate areas of the program. It has been shown by Letovsky and Soloway (1986) that the likelihood of a programmer correctly recognizing a plan or intention in a program decreases as the lines of code that realize it are spread out or delocalized in the program. Reading code examples with delocalized plans or intentions can be difficult to understand as it is time consuming to find all the parts of the plan or intention and then figure out what they do. Consequently, understanding is attempted based on purely local information, resulting in confusion (Welty 1995).

Based on these observations, Table 7.7 illustrates the rejection of our null hypothesis. In that table, \( R \) stands for Rejected \((p < .10)\), and \( AR \) stands for Almost Rejected \((.10 < p < .12)\).

<table>
<thead>
<tr>
<th></th>
<th>Short</th>
<th>Medium</th>
<th>Long</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Control Flow</td>
<td>R</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>Data Flow</td>
<td>R</td>
<td>AR</td>
<td></td>
</tr>
<tr>
<td>State</td>
<td>R</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>Operations</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>

Table 7.7: Rejection of \( H_{0,accuracy,X,Y} \)
7.3.1.2  $H_{0, \text{reviewing-time}, Y}$, Reviewing Time

Differences in reviewing-time between both groups are reported in Table 7.8. We use the format mentioned in Section 7.3.1.1 to represent the results in each cell.

<table>
<thead>
<tr>
<th></th>
<th>Short</th>
<th>Medium</th>
<th>Long</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reviewing time (sec)</td>
<td>475 - 745</td>
<td>655 - 1022</td>
<td>465 - 912</td>
</tr>
<tr>
<td>p</td>
<td>0.0995</td>
<td>0.0446</td>
<td>0.0284</td>
</tr>
</tbody>
</table>

Table 7.8: Average Reviewing Time

Reviewing times were gathered automatically from two sources; Upwork and Violette. We used Upwork’s time tracker to collect reviewing times for those participants in the Read to Understand group, since these participants did not have Violette installed on their Google Chrome browser. We used Violette’s time tracker to collect reviewing times for those participants in the Multistaging to Understand group.

The results in Table 7.8 show shorter reviewing times (close to 50% average reduction) by the treatment group for all considered code examples. Conversely, the results show longer reviewing times by the control group. Based on these results we can reject the hypotheses $H_{0, \text{reviewing-time}, \text{Short}}$, $H_{0, \text{reviewing-time}, \text{Medium}}$, and $H_{0, \text{reviewing-time}, \text{Long}}$.

7.4 Threats to Validity

Threats to internal validity typically include the presence of confounding factors that comprise the integrity of the experimental results. We made every effort to minimize these factors, but possibilities in this case can occur:
• Selection effects. These can occur through variations in the natural performance of individual subjects. Subjects were split into two groups of equal abilities based on their Java experience in an attempt to minimize these effects.

• Maturation effects. These can occur when subjects react differently as time passes. Tasks were similar and were sorted in ascending order by their code example’s size. Consequently, positive (e.g., learning) and negative (e.g., fatigue) effects could not be ruled out. We introduced the tasks at the beginning of the experiment to try and counteract these effects.

• Loss on enthusiasm. Taking into account the length of the experiment, subjects were involved in the experiment for a total of two hours. It is possible that subjects found this action repetitive, and thus their interest dropped towards the end of the experiment. We informed subjects of the context of our research in advance in an attempt to minimize these effects.

Threats to external validity limit the ability to generalize any results from an experiment to a wider population. Our efforts to minimize threats to external validity were minimal, as the nature of our experiment was entirely exploratory. Consequently, possibilities of threats to external validity include:

• The subjects of the experiment (professional programmers recruited at Upwork) may not be representative of the general software engineering population.

• The Java code examples may not be representative (in complexity or length) of
the general Java code examples found on StackOverflow. Interested readers can accessed these examples online at http://stackoverflow.com/tags.

7.5 Conclusion

Our experimental results, discussed in this chapter, demonstrates that our technique is a valuable tool for understanding code examples where most of the code is non-localized, but has only minor benefits when code is partially or fully delocalized. The technique provides consistent speed improvements, irrespective of delocalization.
Chapter 8

Future Work

This chapter takes a step back and reflects on search driven development, how it influences the way we build software today, how it relates to Source Code Curation, and where it is going. We will start by looking at its current challenges, and then suggesting research questions that can help progress the field.

8.1 Reflecting on the Past

Search has always been a key component of software development. For instance, cut-and-paste usually involves search-and-replace; fast code navigation requires local indexing of method/class elements; and foraging for interesting code on StackOverflow or Github to reuse requires using their text-based search applications.

While there have been many attempts to foster Internet-scale code reuse, code reuse at that scale has generally fallen short for three reasons: location, quality, and consumption. These problems are not orthogonal, and thus influence each other.
Search technology can help solve the location problem; especially if the source code of interest is not already inside an existing codebase. Moreover, it can help determine the areas of a source code (part of a larger entity) that can be safely extracted for reuse. Nonetheless, search technology cannot help solve quality and consumption problems (entirely), as it has been postulated in this dissertation.

We can easily sum up the quality problem by the phrase “the quality of an online source code is inherently questionable.” The quality problem is realized by the challenges programmers face during their code foraging activities. This problem has strong implications for effective code foraging, as programmers might have trouble consuming the foreign source code they have found online.

Consumption of a piece of foreign source code covers both its understanding and integration. We can see it as the time and effort (cost) of understanding foreign source code and integrating it into one’s own. Quality-related challenges programmers face during code foraging can lead to a high consumption cost, as they can affect programmers’ decision making during code adaptation [Mackay 1991]. Consequently, if we want to minimize the consumption cost of a foreign source code, then we must address the challenging nature of understanding and adapting it upfront. This is the proposition that drives our Source Code Curation efforts.

This dissertation only scratches the surface of how we can address the challenging nature of online source code, and thus achieve our goal of making code foraging less ad-hoc and more thoughtful. In a world where a large percentage of software engineering knowledge is stored in the Web, there is a need for tools that can couple human
abilities with the latest available computational power. One example would be tools that assist programmers with crafting software solutions from the contributions of the many. Another example would be tools that automatically decompose the curation and consumption processes into small unit tasks that are verifiable by humans. The time is ripe to start both building these tools and running experiments that could finish laying the groundwork for effective and efficient code foraging.

8.2 Looking to the Future

Now we have stepped back and then reflected on the field of search driven development, we now can identify areas that could benefit from better tool support.

We break down these areas into two main themes: (1) Analyzing the retrieved code to make sure it is not introducing any bugs or security flaws, and (2) Coupling code foragers' expertise with the computational power of Codepacking.

8.2.1 Verifying Retrieved Code

In some cases, online source code may have been curated by some code foragers, and continue to be used by others. A familiar example of the latter arises when modifying one's existing source code to either extend, redesign, or replace its functionality. Sometimes, the task is to simply support an alternative application programming interface (API). Other times, foragers may just want to play with the curated code just for educational purposes.

While Source Code Curation might move the needle for code reuse, Source Code
Curation cannot automatically verify at the time of consumption whether a curated code constitutes a bug or a security flaw.

One way to verify whether the curated source is trustworthy is to leverage data mining and crowdsourced program verification techniques. The proliferation of open source software makes it increasingly easy to find large data sets of interesting code we can analyze. For example, we can detect, crowd check, and record common implementation patterns (Beck, 2007) used in popular object oriented languages. By using this knowledge, we can identify pieces of source code that are unlikely to occur in practice and may constitute bugs or security flaws.

Some of the research questions we can suggest to help unpack the above idea include:

- What information or properties should one go after for building a prediction model to detect “unlikely-to-occur-in-practice” source code?

- How can one automatically decompose the detection process into small unit tasks that are verifiable by humans?

- What kinds of assessments should one choose for detecting verification mistakes made by the human verifiers?

### 8.2.2 Keeping Code Foragers in the Loop

During a code example’s curation, sometimes is not possible to have available all the interfaces and its methods, APIs, etc., referenced in the example’s code. This
is expected, as code examples tend to be incomplete. Nonetheless, the curation system
should help in any way possible to bring the affected elements into the scratch space.

While Codepacking (See Chapter 5) helps place orphan code elements (e.g.,
classes, methods, fields, and keywords) within needed class/method declarations, or
suggests those package references that are expected in the current context, there is
always the chance of Codepacking not recommending the code element expected by its
user.

In the near future, a powerful improvement in Codepacking could come from
finding ways to keep code foragers “in the loop.” In other words, Codepacking can gener-
ate code element recommendations that use feedback based on code foragers’ preferences.
For example, code foragers can replace a recommendation generated by Codepacking
with one recommended by them. This feedback is taken into account for subsequent
recommendations. In this fashion, code foragers can refine automated recommendations
by passing indirect knowledge to Codepacking. In return, Codepacking learns the passed
knowledge and then uses it to tune subsequent recommendations.

This semi-automatic version of Codepacking offers a unique combination of
desirable features: incremental expansion of Codepacking’s working scope; Internet-
scale search for missing dependencies; retention of code foragers’ control over what code
elements are brought into the scratch space; and an adaptive feedback mechanism that
glues foragers’ expertise with the computational power of Codepacking.

Some of the possible research questions include:
• What types of feedback strategies and content choices should one include in the semi-automatic Codepacking?

• How specific do code foragers’ preferences have to be for the semi-automatic Codepacking technique to produce valid recommendations?

• What kinds of assessments should one choose to validate the semi-automatic Codepacking technique?
Chapter 9

Conclusion

The main contributions of this dissertation include: (1) an introduction of the notion of Source Code Curation, along with Vesperin, which is a system that provides curation support for programmers; (2) a practical and automated approach (Codepacking) for approximating the declarative completeness of Java code examples; and (3) a scalable approach (Multistaging) for revealing the prime subsets of behavior in Java code examples in an easy to understand sequence, along with a form of code inspection, called Multistaging to Understand, which is based on the Multistaging approach.

Our experiments show that code foragers using Vesperin felt they gained better understanding in unfamiliar code examples via curation. Moreover, they support our claim that Multistaging to Understand is a valuable tool for understanding code examples where most of the code is non-localized, but has only minor benefits when code is partially or fully delocalized. The technique provides consistent speed improvements, irrespective of delocalization. Moreover, our Codepacking mechanism provides consis-
tent support to Vesperin’s code transformations and Multistaging, thus allowing them to work with partial and non-compiling programs, not just complete ones.

After getting our hands dirty building systems, tools, and then running experiments, we took a step back and reflected on search driven development. We discussed how it influences the way we build software today, how it relates to Source Code Curation, and where it is going. This discussion helped us outline the challenges we face now, and suggest directions for future work. Examples include (1) analyzing source code (curated by others) to make sure that the source code to be reused by new users does not introduce bugs or security flaws; (2) generating Codepacking’s code element recommendations that take into account the feedback from code foragers; and (3) exploring different ways of interweaving code retrieved through curation into a programmer’s partially completed program.

All the work discussed above and in this dissertation is motivated by a grander vision of where code foraging practices involving Source Code Curation could take us. The time is ripe to start building Source Code Curation systems and running experiments that could finish laying the groundwork for efficient and effective code foraging.
Bibliography


145