

Automatically Describing Program Structure and Behavior

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Raymond P. L. Buse

May 2012

Abstract

Powerful development environments, data-rich project management systems, and ubiquitous software frameworks have fundamentally altered the way software is constructed and maintained. Today, professional developers spend less than 10% of their time actually writing new code and instead primarily try to understand existing software [1, 2]. Increasingly, programmers work by searching for examples or other documentation and then assembling pre-constructed components. Yet, code comprehension is poorly understood and documentation is often incomplete, incorrect, or unavailable [3, 4]. Moreover, few tools exist to support structured code search making the process of finding useful examples ad hoc, slow and error prone.

The overarching goal of this research is to *help humans better understand software* at many levels and in doing so improve development productivity and software quality. The contributions of this work are in three areas: readability, runtime behavior, and documentation.

We introduce the first **readability metric** for program source code. Our model for readability, which is internally based on logistic regression, was learned from data gathered from 120 study participants. Our model agrees with human annotators as much as they agree with each other. The importance of measuring code readability is highlighted by the observation that reading code is now the most time consuming part [5, 6, 7, 8] of the most expensive activity [9, 10] in the software development process.

Beyond surface-level readability, understanding software demands understanding the behavior of the running program. To that end, we contribute a model describing **runtime behavior** in terms of program paths. Our approach is based on two key insights. First, we observe a relationship between the relative frequency of a path and its effect on program state. Second, we carefully choose the right level of abstraction for considering program paths: interprocedural for precision, but limited to one class for scalability. Over several benchmarks, the top 5% of paths as ranked by our algorithm account for over half of program runtime.

Finally, we describe automated approaches for improving the understandability of software through **documentation synthesis**. A 2005 NASA survey found that the most significant barrier to code reuse is that software is too difficult to understand or is poorly documented [11]. As software systems grow ever larger and more complex, we believe that automated documentation tools will become indispensable. The key

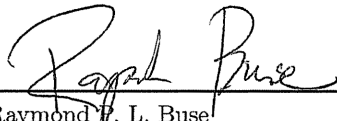
to our approach is adapting programming language techniques (e.g., symbolic execution) to create output that is directly comparable to existing human-written artifacts. This has two key advantages: (1) it simplifies evaluation by permitting objective comparisons to existing documentation, and (2) it enables tools to be used immediately — no significant change to the development process is required. We describe and evaluate documentation synthesis algorithms for exceptions, code changes, and APIs. In each case, we employ real humans to evaluate the output of our algorithms and compare against artifacts created by real developers — finding that our synthesized documentation is of similar quality to human-written artifacts.¹

Increasingly, software comprehension is a critical factor in modern development. However, software remains difficult to understand. This research seeks to improve that state by *automatically describing program structure and behavior*.

¹UVA IRB approval was obtained for all of the human studies and user evaluations described in this document.

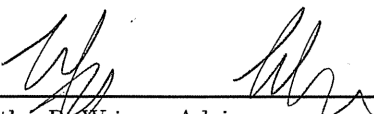
Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy (Computer Science)

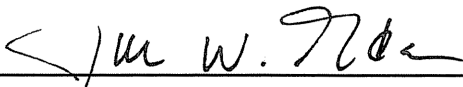


Raymond P. L. Buse

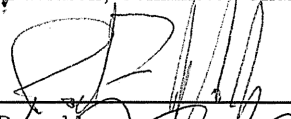
This dissertation has been read and approved by the Examining Committee:



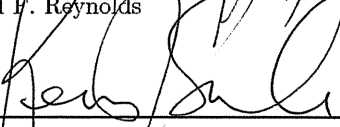
Westley R. Weimer, Advisor



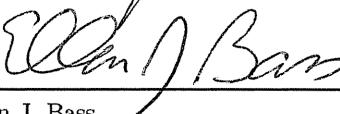
Jack W. Davidson, Committee Chair



Paul F. Reynolds

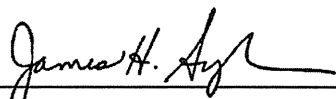


Kevin J. Sullivan



Ellen J. Bass

Accepted for the School of Engineering and Applied Science:



James H. Aylor, Dean, School of Engineering and Applied Science

May 2012

“It takes time to persuade men to do even what is for their own good.”

–Thomas Jefferson

Acknowledgments

I am in debt to the many great people who contributed to this work both directly and indirectly.

Wes Weimer, my advisor, offered many ideas and then permitted me to silently ignore numerous good ones. Me hermano, Pieter Hooimeer, always set a good example for how to avoid work. Krasimira Kapitanova always found more work for me to do.

Other friends and family helped make life both fun and interesting. Claire Le Goues taught me, for example, how, to use, commas and semi-colons; properly. Chris Gregg showed me how useful GPUs are, right before he showed me how impossible they are to use. Amy Verret prepared delicious scurvy-fighting meals.

Finally, I thank Aubry Verret, my wife and a software engineer, who once gave me the best encouragement I could hope to receive: “I really want that API-Doc tool.”

Ray Buse

May 2, 2012

Charlottesville, VA

Contents

Abstract	i
Acknowledgments	v
Contents	vi
List of Tables	ix
List of Figures	x
List of Terms	xiii
1 Introduction	1
1.1 Modeling Code Readability	2
1.2 Predicting Runtime Behavior	3
1.3 Synthesizing Documentation	4
1.3.1 Documentation for Exceptions	5
1.3.2 Documenting Code Changes	6
1.3.3 Documenting API Usage	7
1.4 Summary and Long-Term Vision	8
2 A Metric for Code Readability	9
2.1 Introduction	9
2.2 Study Methodology	12
2.2.1 Snippet Selection Policy	12
2.2.2 Readability Scoring	13
2.2.3 Study Participation	14
2.3 Study Results	15
2.4 Readability Model	16
2.4.1 Model Generation	17
2.4.2 Model Performance	19
2.5 Correlating Readability with Software Quality	21
2.5.1 Readability Correlations	23
2.5.2 Readability and Complexity	26
2.5.3 Software Lifecycle	27
2.6 Discussion	29
2.7 Threats to Validity	30
2.8 Conclusion	31
3 Estimating Path Execution Frequency	32
3.1 Introduction	32
3.2 Context and Motivation	34
3.2.1 Motivating Example	35
3.2.2 Example Client Applications	37
3.3 Path Enumeration	38

3.3.1	Static Path Enumeration	38
3.3.2	Dynamic Path Enumeration	39
3.4	Static Path Description Model	40
3.5	Model Evaluation	41
3.5.1	Experimental Setup	41
3.5.2	Classifier Training	42
3.5.3	F-score Analysis	43
3.5.4	Weighted Rank Analysis	44
3.6	Path Frequency and Running Time	46
3.7	Path Frequency for Branch Prediction	46
3.8	Model Implications	48
3.8.1	Feature Power Analysis	49
3.8.2	Threats To Validity	49
3.9	Conclusion	50
4	Documenting Exceptions	52
4.1	Introduction	52
4.2	Motivating Example	55
4.3	Exception Flow Analysis	56
4.4	Existing Exception Documentation	58
4.4.1	Complete and Consistent Documentation	59
4.4.2	Copy-and-Paste Documentation	60
4.4.3	Difficulty in Documenting Exceptions	61
4.4.4	Utility of Documentation	63
4.5	Algorithm for Generating Documentation	64
4.5.1	Documentation Post-Processing	67
4.6	Generated Documentation Quality	67
4.6.1	Comparing Automatic with Human Documentation	67
4.6.2	Evaluating Copy-and-Paste Documentation	70
4.7	Conclusion	71
5	Documenting Program Changes	73
5.1	Introduction	73
5.2	Motivating Example	75
5.3	Empirical Study of Log Messages	77
5.4	Algorithm Description	78
5.4.1	Obtaining Path Predicates	79
5.4.2	Generating Documentation	80
5.4.3	Summarization Transformations	81
5.5	Evaluation	84
5.5.1	Size Evaluation	85
5.5.2	Content Evaluation	86
5.5.3	Qualitative Analysis	90
5.5.4	Threats To Validity	91
5.6	Related Work	91
5.7	Conclusion	92
6	Synthesizing API Usage Examples	94
6.1	Introduction	94
6.2	Motivating Example	96
6.3	Human-Written Usage Examples	97
6.3.1	Properties of Human-Written Examples	98
6.3.2	Survey Results	100
6.4	Algorithm Description	102

6.4.1	Path Enumeration	102
6.4.2	Predicate Generation	103
6.4.3	Clustering and Abstraction	104
6.4.4	Emitting Documentation	105
6.4.5	Implementation Details	107
6.5	Indicative Examples	108
6.5.1	Naming	108
6.5.2	Patterns	109
6.5.3	Exceptions	109
6.6	Empirical Evaluation	110
6.6.1	Size Evaluation	111
6.6.2	Readability Evaluation	111
6.7	Human Study	111
6.7.1	Experimental Setup	111
6.7.2	Quantitative Results	112
6.8	Threats To Validity	113
6.9	Related Work	114
6.10	Conclusion	115
7	Conclusion	116
7.1	Summary of Results	116
7.2	Research Impact	116
7.3	Final Remarks	120
A	Study Instruments	121
	Bibliography	126

List of Tables

2.1	Correlation statistics; inter-annotator agreement	16
2.2	Readability features	18
3.1	Runtime frequency features.	41
3.2	Benchmarks used for evaluation of path frequency model.	42
3.3	Statistics from static intra-class path enumeration.	42
4.1	Documentation for exceptions benchmarks	58
4.2	Documentation for Exceptions Numerical Results	68
5.1	Benchmark set for change documentation evaluation.	76
5.2	Table of relational operators used for documentation comparison.	88
6.1	Corpus for example generation in this study.	108
7.1	Summary of Research Questions.	117
7.2	Publications supporting this dissertation.	120

List of Figures

2.1	Rendering of the readability dataset	14
2.2	Frequency distribution of the readability dataset	15
2.3	Readability scores distribution	17
2.4	Spearman's ρ across readability annotators	20
2.5	Readability annotator agreement by experience group	21
2.6	Readability features comparative predictive power	22
2.7	Readability benchmark set	23
2.8	Readability vs. External metrics – f-measure	24
2.9	Readability vs. External metrics – Continuous probabilities	25
2.10	Readability vs. Cyclomatic Complexity	26
2.11	Readability vs. defects – longitudinal	27
2.12	Readability vs. Project Maturity	28
3.1	The put method from <code>java.util.Hashtable</code>	36
3.2	Pseudocode for static path enumeration.	39
3.3	F -score for classifying high frequency paths	44
3.4	Kendall's tau performance for path frequency model.	45
3.5	Percentage of total run time covered by examining the top paths in each method.	47
3.6	Percentage of the total run time covered by examining the top paths of the entire program.	47
3.7	Static branch prediction hit rate.	48
3.8	Relative predictive power of features as established with a singleton analysis.	50
4.1	Algorithm for determining method exception information.	57
4.2	Completeness of exception documentation survey.	60
4.3	Overall documentation completeness (from JAVADOC <code>throws</code> clauses.	61
4.4	Exception documentation completeness as a function of exception propagation depth.	62
4.5	Correlation between the documentation rate and three proxies for the difficulty of comprehending a method.	63
4.6	Correlation between the rate at which exceptions are undocumented at invocation sites and whether the method underwent a defect repair in the past.	64
4.7	Algorithm for inferring exception documentation.	66
4.8	Exception documentation quality; tool vs. human.	69
4.9	Copy-and-Paste exception documentation.	71
5.1	Percent of changes modifying a given number of source code files.	77
5.2	Percent of changes with WHAT and WHY documentation.	78
5.3	High-level view of the architecture of DELTADOC	79
5.4	High-level pseudo-code for Documentation Generation.	80
5.5	Size comparison between <code>diff</code> , DELTADOC, and human generated documentation.	85
5.6	The cumulative effect of applying our set of transformations on the size of documentation.	85
5.7	Score metric across each benchmark.	89
6.1	ApiDoc algorithm visualization	98

6.2	Histogram of usage example sizes.	99
6.3	Feature importance for example documentation (survey results).	101
6.4	Size comparison of examples generated by our tool human-written	110
6.5	Readability comparison our tool vs human-written.	112
6.6	Human study results comparing the quality of our documentation with other tools and human-written.	113
7.1	Mock-up of an API-usage auto-completion tool.	118
7.2	Mock-up of an API-usage uncommon pattern detection tool.	119
A.1	Web-based tool for annotating the readability of code snippets.	122
A.2	Instructions for paper-based study for comparing change documentation (page 1/2).	123
A.3	Instructions for paper-based study for comparing change documentation (page 2/2).	124
A.4	Web-based tool for API example study.	125

List of Terms

accidental complexity — In the Brooks model [12], accidental complexity refers to the complexity in a program that is not intrinsic to the problem being solved (see **essential complexity**). Readability is one aspect of accidental complexity. [xiv](#), [10](#)

Application Program Interface (API) — A source code-based specification intended to be used as an interface by software components to communicate with each other. APIs are typically considered important points for documentation. [7](#), [94](#)

classifier — A mapping from unlabeled instances to (discrete) classes. Classifiers have a form (e.g., decision tree) plus an interpretation procedure (including how to handle unknowns, etc.). Some classifiers also provide probability estimates (scores), which can be thresholded to yield a discrete class decision thereby taking into account a utility function. [17](#), [42](#)

commit message — A commit or “check in” is the action of writing or merging the changes made in the working copy back to the repository. Developers typically associate with the change a free-form textual message known as a “commit message” or “log message.” A commit message typically describes **WHY** the change was made and **WHAT** the change is. [xv](#), [6](#), [73](#)

descriptive model — A mathematical model that can accurately predict future outcomes. Such a model represents a hypothesis for how the underlying system operates. In comparison, a normative model is a prescriptive model which suggests what ought to be done or how things should work according to an assumption or standard. [2](#), [10](#), [33](#)

documentation — Non-code text which describes some aspect of a software system to facilitate understanding. [4](#), [52](#)

essential complexity — In the Brooks model [12], essential complexity is a property of software arising from system requirements, and cannot be abstracted away. In comparison, **accidental complexity** can be reduced. [xiii](#), [10](#)

exception — A programming language construct (in other contexts also a hardware mechanism) designed to handle the occurrence of exceptional conditions: special conditions often changing the normal flow of program execution. In general, an exception is first raised (thrown) at one program point and then handled (caught) by switching the execution to a second program point (the exception handler). [5](#), [52](#)

feature — In the context of machine learning a feature (or attribute) is a quantity describing an object to be classified (i.e., an **instance**). A feature has a domain defined by the feature type (typically booleans, integers, or real numbers). Features may be nominal if there is no implied ordering between the values (e.g., colors) or ordinal if there is an ordering (e.g., low, medium, or high). [2](#), [10](#), [33](#), [40](#), [42](#)

human study — The use of human subjects in the scientific evaluation of a research hypothesis. The human studies reported in this document have each been approved by an Institutional Review Board. [3](#)

instance — In the context of machine learning an instance (also: example, case, record) is single object of the world from which a model will be learned, or on which a model will be used (e.g., for prediction). In most machine learning work, instances are described by feature vectors. [xiv](#), [18](#), [42](#)

metric — Formally, a function which defines a distance between elements of a set. In the software domain, a measure of some property of a piece of software or its specifications. The goal is obtaining objective, reproducible and quantifiable measurements, which may have valuable applications in schedule and budget planning, cost estimation, quality assurance testing, software debugging, software performance optimization, and optimal personnel task assignments. [3](#)

path enumeration — The process of identifying a set of program paths. Can be either static (in which case the number of acyclic paths is exponential in the number of program statements), or dynamic (i.e., based on observations or traces of the running program). [4](#), [38](#)

program expression — A combination of explicit values, constants, variables, operators, and functions that are interpreted according to the particular rules of precedence and of association for a particular programming language. [xv](#), [4](#), [79](#)

program path — A sequence of program statements representing the flow of control through an imperative program. [4](#), [32](#)

program statement — The smallest standalone element of an imperative programming language. A program is formed by a sequence of one or more statements. A statement comprises internal components (e.g., [program expressions](#)). [13](#)

readability — A human judgment of how easy a text is to understand. [2](#), [9](#)

static analysis — Analysis of computer software that is performed without actually executing programs built from that software (e.g., analysis of source code by an automated tool). [3](#)

version control system — Also known as revision control or source control is a management of changes to program source and other documents. Changes are usually identified by a number termed the “revision.” Each revision is associated with a time-stamp and the person making the change and a commit message. An instance of a version control system is typically called a “repository.” Common systems include CVS, SVN, Mercurial, and Git. [6](#), [73](#)

WHAT information — In the context of a [commit message](#), WHAT information refers to any text that describes the change itself, or the effect of the change on the runtime behavior of the program (see also [WHY information](#)). [xiii](#), [xv](#), [74](#)

WHY information — In the context of a [commit message](#), WHY information is any information that does NOT describe the change itself but instead describes why the change was made (e.g., to fix a bug), adds additional context, or consists of any other information that is not considered [WHAT information](#). [xiii](#), [xv](#), [74](#)

Chapter 1

Introduction

“I have a theory. Do you want to hear it?”

– *Fox Mulder*

SOFTWARE development is a large global industry, but software products continue to ship with known and unknown defects [13]. In the US, such defects cost firms many billions of dollars annually by compromising security, privacy, and functionality [14]. To mitigate this expense, recent research has focused on finding specific errors in code (e.g., [15, 16, 17, 18, 19, 20, 21, 22, 23, 24]). These important analyses hold out the possibility of identifying many types of implementation issues, but they fail to address a problem underlying all of them: software is difficult to understand.

Software understandability becomes increasingly important as the number and size of software projects grow: as complexity increases, it becomes paramount to comprehend software and use it correctly. Fred Brooks once noted that “the most radical possible solution for constructing software is not to construct it at all” [25], and instead assemble already-constructed pieces. Code reuse and composition are becoming increasingly important: a recent study found that a set of programs was comprised of 32% re-used code (not including libraries) [26], whereas a similar 1987 study estimated the figure at only 5% [27]. In a future where software engineering focus shifts from implementation to design and composition concerns, program understandability will become even more important.

In this dissertation we present a general and practical approach for *analyzing* program understandability from the perspective of real humans. In addition, we present novel algorithms for *synthesizing* documentation such that programs can be made easier to understand. We will focus on three key dimensions of program understandability: **readability**, a local judgment of how easy code is to understand; **runtime behavior**, a

characterization of what a program was designed to do; and **documentation**, non-code text that aids in program understanding. We make the following overarching claim:

Thesis: Program structure and behavior can be accurately modeled with semantically shallow features and documented automatically.

Two key technical insights underlie our approach. (1) We use machine learning and statistical methods to combine multiple surface features of code (e.g., *identifier length* or *number of assignment statements*) to characterize aspects of programs that lack precise semantics. The use of lightweight features permits our techniques to scale to large programs and generalize across multiple application domains. (2) We leverage programming language techniques — including symbolic execution, test case generation, and specification mining — to synthesize human-readable text that describes important aspects of program behavior. We generate output that is directly comparable to real-world human-created documentation. This is useful for evaluation, but also suggests that our tools could be readily integrated into current software engineering practice.

We begin by introducing five major research thrusts each comprising a chapter in this dissertation: Chapter 2: Modeling Code Readability, Chapter 3: Predicting Runtime Behavior, Chapter 4: Documentation for Exceptions, Chapter 5: Documentation for Code Changes, and Chapter 6: Documentation for APIs. We enumerate a total of twelve key research questions (**RQ 1** to **RQ 12**) and overview our approach to answering each.

1.1 Modeling Code Readability

We define **readability** as a human judgment of how easy a text is to understand. In the software domain, this is a critical determining factor of quality [28]. Perhaps everyone who has written code has an intuitive notion of the concept of software readability, and that program features such as indentation (e.g., as in Python [29]), choice of identifier names [30], and comments play a significant part. Dijkstra, for example, claimed that the readability of a program depends largely upon the simplicity of its sequencing control, and employed that notion to help motivate his top-down approach to system design [31].

Human notions of code readability arise from a potentially complex interaction of textual **features** (e.g., line length, whitespace, choice of identifier names, etc.). In Chapter 2, we present a **descriptive model** of readability based on such features. Our technique admits analysis (e.g., principle component) to determine the underlying factors which contribute most to a readability judgment. Understanding the relationship

between this feature space and the high-level notion of readability may influence both software engineering and language design.

RQ 1: To what extent do humans agree on code readability?

Before designing a model for readability, we must first characterize the extent to which humans agree on it. Notably, if humans have low agreement (i.e., personal preference dominates the judgment) then a generic readability model would likely be impractical.

We present a **human study** of code readability. A total of 120 students at *The University of Virginia* were asked to rate the readability of 100 short code snippets. Employing statistical methods, we find a high degree of inter-annotator agreement implying that a generic model for readability is feasible.

RQ 2: Is the proposed readability model accurate?

We next evaluate whether our model accurately describes human notions of readability. We measure the correlation between the output of the model and responses from the participants in our study. We employ cross-validation to mitigate the threat of over-fitting. Our main finding is that our model agrees with humans at least as well as they agree with each other (e.g., Spearman’s $\rho = 0.55$ humans, 0.71 model, $p < 0.0001$).

RQ 3: Is the proposed readability model correlated with external notions of software quality?

A readability model permits us to construct an automated **metric** suitable for use in software quality analysis and evaluation tasks. We evaluate our metric with respect to project stability, defect density, and other established quality indicators. A significant correlation of this type supports the practical utility of our approach.

1.2 Predicting Runtime Behavior

Runtime behavior refers to what a program will do, or is most likely to do, when executed — information that is typically unavailable for a **static analysis**. We claim that understanding runtime behavior is critical to understanding code. This conjecture is supported by the observation that runtime behavior information is a key aspect of documentation.

Much documentation implicitly requires code summarization or other filtering. For example, the Java standard library implementation of `Hashtable.put()`, is documented by describing its expected most common behavior, “Maps the specified key to the specified value” rather than describing what happens if the table must be expanded and all entries rehashed.

In Chapter 3, we present a model of static **program paths** suitable for estimating runtime frequency. Central to our technique is static **path enumeration**, whereby we enumerate all acyclic intra-class paths in a target program. To train our model and experimentally validate our technique, we also require a process for dynamic path enumeration that counts the number of times each path in a program is executed during an actual program run. The goal of this work is to produce a static model of dynamic execution frequency that agrees with the actual execution frequency observed on held-out indicative workloads.

RQ 4: What static code features are predictive of path execution frequency?

We investigate whether information about the runtime behavior of imperative programs, including their relative path execution frequency, is embedded into source code by developers in a predictable way. We hypothesize that paths that change a large amount of program state (e.g., update many variables, throw an exception and thus pop stack frames, etc.) are expected by programmers to be executed more rarely than paths that make small, incremental updates to program state (e.g., change just a few fields). This is consistent with the general strategy embodied by Amdahl’s law — make the common case fast.

Our proposed model can be trained on a set of surface-level features automatically. We investigate utility of the candidate feature set using Principle Component Analysis (PCA) and singleton feature analysis.

RQ 5: Is the proposed path frequency model accurate?

We present an extensive empirical evaluation of our model. We measure accuracy both in identifying hot paths and in ranking paths by frequency. We demonstrate the flexibility of our model by employing it to characterize the running time and dynamic branching behavior of several benchmarks.

1.3 Synthesizing Documentation

In addition to modeling code understandability, we propose to improve it by creating tools and algorithms capable of synthesizing **documentation**. David Parnas claims documentation is “the aspect of software engineering most neglected by both academic researchers and practitioners” [3].

Our general strategy is to mimic human documentation by *summarizing the effect* of a statement on the functional behavior and state of a program. Automatic documentation of this type holds out the promise of replacing much of human effort with artifacts that are more consistent, more accurate, and easier to maintain than traditional documentation.

Our technique is based on a combination of program analyses that can be used to summarize **program expressions**. Symbolic execution [32] and dataflow analysis [33] allow us to condense sequences of instructions

into single ones by evaluating and combining terms. We couple this with alias analysis (e.g., [34, 35, 36]) to resolve invocation targets. Additionally, we present new dynamic summarization heuristics. For example, we can re-arrange logical terms to condense their textual representation and we can select for relevancy and importance especially in consideration of runtime behavior metrics (e.g., prefer to document method calls instead of assignment statements).

We now describe three research thrusts based on the generation of documentation.

1.3.1 Documentation for Exceptions

Modern **exception** handling allows an error detected in one part of a program to be handled elsewhere depending on the context [37]. This construct produces a non-sequential control flow that is simultaneously convenient and problematic [38, 39]. Uncaught exceptions and poor support for exception handling are reported as major obstacles for large-scale and mission-critical systems (e.g., [40, 41, 42, 43]). Some have argued that the best defense against this class of problem is the complete and correct documentation of exceptions [44].

Unfortunately, the difficulty of understanding exceptions leads to many instances of incorrect, out-of-date, or inconsistent documentation [45]. An automated tool, by contrast, could provide specific and useful documentation that is synchronized with an evolving code base.

In Chapter 4 we present an algorithm that statically infers and characterizes exception-causing conditions in programs. We show that the output of the algorithm is usable as documentation of exceptional conditions. The proposed algorithm has two parts. First, we locate exception-throwing instructions and track the flow of exceptions through the program. Second, we symbolically execute control flow paths that lead to these exceptions. This symbolic execution generates boolean formulae over program variables that describe feasible paths. If the formula is satisfied at the time the method is invoked, then the exception can be raised. We output a string representation of the logical disjunction of these formula for each method/exception pair.

RQ 6: Is the proposed algorithm for exception documentation accurate?

Rice’s theorem implies that automated documentation cannot be perfect in all cases. Nonetheless, practitioners desire output that is simultaneously accurate (i.e., if the condition is met, the exception is thrown) and precise (i.e., the condition is as narrow as possible). Our primary criterion for success is that synthesized documentation be competitive with existing human-written documentation. We compare 951 instances of documentation generated by our tool with documentation written by real developers and present in real programs. We employ a rubric that facilitates objective and repeatable comparisons between human documentation and automated tool output.

RQ 7: Does the proposed algorithm for exception documentation scale to large programs (i.e., over 100k LOC)?

A key challenge is ensuring scalability in our fully inter-procedural analysis. In addition to call-graph preprocessing and infeasible path elimination, we propose to enumerate paths “backward” from exception throwing statements. Because these statements are relatively rare (as compared to method call sites) this allows us to significantly reduce the time needed for path enumeration. We find that our algorithm processes over 170k lines of code in under 25 minutes.

1.3.2 Documenting Code Changes

Much of software engineering can be viewed as the application of a sequence of modifications to a code base. **Version control systems** allow developers to associate a free-form textual **commit message** with each change they introduce [46]. The use of such commit messages is pervasive among development efforts that include version control systems [47]. Version control log messages can help developers validate changes, locate and triage defects, and generally understand modifications [48].

A typical commit message describes either or both what was changed and why it was changed. For example, revision 2565 of JABREF, a popular bibliography system, is documented with “Added Polish as language option.” Revision 3504 of PHEX, a file sharing program, is documented with “providing a disallow all robots.txt on request.” However, log messages are often less useful than this; they do not always describe changes in sufficient detail for other developers to fully understand them. Consider revision 3909 of iTEXT, a PDF library: “Changing the producer info.” Since there are many possible producers in iTEXT, the comment fails to explain what is being changed, under what conditions, or for what purpose.

The lack of high-quality documentation in practice is illustrated by the following indicative appeal from the MACPORTS development mailing list: “Going forward, could I ask you to be more descriptive in your commit messages? Ideally you should state what you’ve changed and also why (unless it’s obvious) ... I know you’re busy and this takes more time, but it will help anyone who looks through the log ...”¹

In Chapter 5 we present an algorithm suitable for supplementing human documentation of changes by describing the *effect* of a change on the runtime behavior of a program, including the conditions under which program behavior changes and what the new behavior is.

At a high level, our approach generates structured, hierarchical documentation of the form:

```
When calling A(), if X, do Y instead of Z.
```

¹<http://lists.macosforge.org/pipermail/macports-dev/2009-June/008881.html>

RQ 8: What qualities are necessary and sufficient for an effective commit message?

We present a study of existing commit messages. We characterize the information content of such messages as well as non-functional aspects like length and readability. We also present anecdotal information from practitioners about what makes commit messages effective.

RQ 9: Is the output of the proposed algorithm for change documentation accurate?

Change documentation must contain certain information to be useful. We conjecture that if the output of our algorithm contains the same information as human-written commit messages, without containing additional incorrect information, it will likely be effective in practice. We compare our algorithm’s output to 250 human-written messages from five widely-used open-source projects. We use a mechanical *score metric* to compare the information content of messages. We find that at least 82% of information in existing human-written commit messages is also contained in the output of the proposed algorithm. We supplement our quantitative analysis with subjective opinions from human annotators concerning the quality of the output of the proposed algorithm.

RQ 10: Is the output of the proposed algorithm for change documentation readable?

Evidence we have gathered suggests that high-quality commit messages are both short and easy to understand. Our algorithm optimizes for brevity and understandability by applying summarization transformations. We compare human-written documentation to the output of our algorithm. We measure length in terms of both the number of lines and number of characters. We measure readability by the automated metric introduced in Chapter 2. In both cases we find no statistically significant difference between our automated documentation and human-written.

1.3.3 Documenting API Usage

In studies of developers, usage examples of **Application Program Interfaces (APIs)** have been found to be a key learning resource [49, 50, 51, 52, 53]. Conceptually, documenting how to *use* an API is often preferable to simply documenting the function of each of its components.

One study found that the greatest obstacle to learning an API in practice is “insufficient or inadequate examples” [54]. We propose to design an algorithm that automatically generates documentation of this type. Given a corpus of usage examples (e.g., indicative uses mined from other programs), we propose to distill the most common use case and render it in a form suitable for use as documentation.

In Chapter 6 we present a technique for automatically synthesizing human-readable API usage examples which are well-typed and representative (i.e., corresponding to real-word use). We adapt techniques from

specification mining [55, 56, 57], the task of reverse engineering partial program specifications from a program implementation. We employ data-flow analysis to extract important details about each use including how return values are later used. We then gather up groups of such concrete uses and abstract them as generic examples. Because a single API may have multiple common use scenarios, we use clustering to discover and coalesce related usage patterns before expressing them as documentation.

RQ 11: What qualities are necessary and sufficient for an effective usage example?

Not all examples are equally useful for real developers. Examples obtained by search-based techniques (e.g., [58, 59]) often contain extraneous statements, lack important context, or are incomplete. To better understand the parameters of effective documentation we present a study of gold-standard human-written examples from the Java Standard Development Kit (SDK). Furthermore, we present results from a large survey on example quality and utility. This study informs our algorithm for example synthesis.

RQ 12: Is the output of the proposed algorithm for usage examples correct and useful?

We consider two key dimensions of correctness relevant to examples: syntactic correctness (i.e., the example constitutes well-formed code) and representativeness (i.e., the example faithfully abstracts actual use). We argue that our examples exhibit both of these properties by construction. We evaluate the utility of our algorithm with a human study. Over 150 participants compared the output of our tool to gold-standard API documentation from the Java SDK. Our analysis suggests that our tool could replace as many as 82% of human-written examples.

1.4 Summary and Long-Term Vision

The long-term goal of our work is to produce tools, models and algorithms to enhance code understanding. In general, evaluation takes three forms: directly, by analyzing a model’s accuracy with respect to training data; indirectly, by examining correspondence between the model’s output and traditional quality metrics; and against humans, by comparing produced documentation to human-written documentation.

We claim that program understandability is a critical to successful software engineering. We propose to provide developers with a better understanding of programs.

Chapter 2

A Metric for Code Readability

“Of course, you don’t have to take *my* word for it.”

– *LeVar Burton, Reading Rainbow*

2.1 Introduction

WE begin our investigation of automated approaches to program understanding by considering the most fundamental idea in the comprehension of text: **readability**. Today, rather than writing new code, developers spend the majority of their time composing and maintaining existing software [5, 7, 8]. Knuth asserts that readability is essential; it makes programs “more robust, more portable, [and] more easily maintained” [60].

Readability is often recognized as a first-class development concern. Elshoff and Marcotty, after finding that many commercial programs were much more difficult to read than necessary, proposed adding a development phase in which the program is made more readable [61]. Knight and Myers suggested that one phase of software inspection should be a check of the source code for readability [62] to ensure maintainability, portability, and reusability of the code. Basili *et al.* [63] proposed a series of reading techniques that are effective at improving defect detection during inspections. Haneef proposed adding a dedicated readability and documentation group to the development team, observing that, “without established and consistent guidelines for readability, individual reviewers may not be able to help much” [64]. Knuth emphasizes the importance of readability in his *Literate Programming* methodology [60]. He suggests that a program should be treated as “a piece of literature, addressed to human beings.”

In this chapter we present the first **descriptive model** of software readability. Our model is based on simple **features** that can be extracted automatically from programs. We show that this model correlates strongly with human annotators and also significantly with external, widely available, measures of software quality such as defect detectors and software changes.

Readability metrics are in widespread use in the context of natural languages. The Flesch-Kincaid Grade Level [65], the Gunning-Fog Index [66], the SMOG Index [67], and the Automated Readability Index [68] are just a few examples of readability metrics for ordinary text. These metrics are all based on simple factors such as average syllables per word and average sentence length. Despite this simplicity, they have each been shown to be quite useful in practice. Flesch-Kincaid, which has been in use for over 50 years, has not only been integrated into popular text editors including Microsoft Word, but has also become a United States governmental standard. Agencies, including the Department of Defense, require many documents and forms, internal and external, to meet have a Flesch readability grade of 10 or below (DOD MIL-M-38784B). Defense contractors also are often required to use it when they write technical manuals.

These metrics can help organizations gain some confidence that their documents meet goals for readability very cheaply, and have become ubiquitous for that reason. We believe that similar metrics, targeted specifically at source code and backed with empirical evidence for effectiveness, can serve an analogous purpose in the software domain. Readability metrics for the niche areas such as computer generated math [69], treemap layout [70], and hypertext [71] have been found useful. We describe the first general readability metric for source code.

Readability is not the same as complexity, for which some existing metrics have been empirically shown useful [72]. Brooks claims that complexity is an **essential** property of software; it arises from system requirements, and cannot be abstracted away [12]. In the Brooks model, readability is “accidental” because it is not determined by the problem statement. In principle, software engineers can only control **accidental complexity**: implying that readability can be addressed more easily than intrinsic difficulties.

While software complexity metrics typically take into account the size of classes and methods, and the extent of their interactions, the readability of code by our definition is based on local, line-by-line factors. Our notion of readability arises directly from the judgments of actual human annotators who do not have context for the code they are judging. Complexity factors, on the other hand, may have little relation to what makes code understandable to humans. Previous work [73] has shown that attempting to correlate artificial code complexity metrics directly to defects is difficult, but not impossible. Although both involve local factors such as indentation, readability is also distinct from coding standards (e.g., [74, 75, 76]), conventions primarily intended to facilitate collaboration by maintaining uniformity between code written by different developers.

In this study, we have chosen to target readability directly both because it is a concept that is independently valuable, and also because developers have great control over it. We show in Section 2.5 that there is indeed a significant correlation between readability and quality. The main contributions of this chapter are:

- A technique for the construction of an automatic software readability metric based on local code features.
- A survey of 120 human annotators on 100 code snippets that forms the basis of the metric presented and evaluated in this chapter. We are unaware of any published software readability study of comparable size (12,000 human judgments). We directly evaluate the performance of our model on this data set.
- A set of experiments which reveal significant correlations between our metric for readability and external notions of software quality including defect density. We evaluate on over two million lines of code.
- A discussion of the features involved in our metric and their relation to software engineering and programming language design.
- An experiment correlating our readability metric with explicit human mentions of bug repair. Using software version control repository information we coarsely separate changes made to address bugs from other changes. We find that low readability correlates more strongly with this direct notion of defect density than it does with the previous approach of using potential bugs reported by static analysis tools.
- An experiment which compares readability to cyclomatic complexity. This experiment serves to validate our claim that our notion of readability is largely independent from traditional measures of code complexity.
- A longitudinal experiment showing how changes in readability can correlate with changes in defect density as a program evolves. For ten versions of each of five programs we find that projects with unchanging readability have similarly unchanging defect densities, while a project that experienced a sharp drop in readability was subject to a corresponding rise in defect density.

The structure of this chapter is as follows. In Section 2.2 we present a study of readability involving 120 human annotators. We present the results of that study in Section 2.3, and in Section 2.4 we determine a small set of features that is sufficient to capture the notion of readability for a majority of annotators. In Section 2.5 we discuss the correlation between our readability metric and external notions of software quality. We discuss some of the implications of this work on programming language design in Section 2.6, discuss potential threats to validity in Section 2.7, and conclude in Section 2.8.

2.2 Study Methodology

A consensus exists that readability is an essential determining characteristic of code quality [28, 77, 5, 31, 61, 64, 73, 78, 7, 30, 8, 72], but not about which factors most contribute to human notions of software readability. A previous study by Tenny looked at readability by testing comprehension of several versions of a program [79]. However, such an experiment is not sufficiently fine-grained to extract precise features. In that study, the code samples were large, and thus the perceived readability arose from a complex interaction of many features, potentially including the purpose of the code. In contrast, we choose to measure the software readability of small (7.7 lines on average) selections of code. Using many short code selections increases our ability to tease apart which features are most predictive of readability. We now describe an experiment designed to extract a large number of readability judgments over short code samples from a group of human annotators.

Formally, we can characterize software readability as a mapping from a code sample to a finite score domain. In this experiment, we presented human annotators with a sequence of short code selections, called *snippets*, through a web interface. The annotators were asked to individually score each snippet based on their personal estimation of readability. We now discuss three important parameters: snippet selection policy (Section 2.2.1), snippet scoring (Section 2.2.2), and participation (Section 2.2.3).

2.2.1 Snippet Selection Policy

We claim that the readability of code is very different from that of natural languages. Code is highly structured and consists of elements serving different purposes, including design, documentation, and logic. These issues make the task of snippet selection an important concern. We have designed an automated policy-based tool that extracts snippets from Java programs.

First, snippets should be relatively short to aid feature discrimination. However, if snippets are too short, then they may obscure important readability considerations. Second, snippets should be logically coherent to allow annotators the context to appreciate their readability. We claim that they should not span multiple method bodies and that they should include adjacent comments that document the code in the snippet. Finally, we want to avoid generating snippets that are “trivial.” For example, we are uninterested in evaluating the readability of a snippet consisting entirely of boilerplate import statements or entirely of comments.

Consequently, an important tradeoff exists such that snippets must be as short as possible to adequately support analysis by humans, yet must be long enough to allow humans to make significant judgments on them. Note that it is *not* our intention to “simulate” the reading process, where context may be important

to understanding. Quite the contrary: we intend to eliminate context and complexity to a large extent and instead focus on the “low-level” details of readability. We do not imply that context is unimportant; we mean only to show that there exists a set of local features that, by themselves, have a strong impact on readability and, by extension, software quality.

With these considerations in mind, we restrict snippets for Java programs as follows. A snippet consists of precisely three consecutive simple **program statements** [80], the most basic unit of a Java program. Simple statements include field declarations, assignments, function calls, breaks, continues, throws and returns. We find by experience that snippets with fewer such instructions are sometimes too short for a meaningful evaluation of readability, but that three statements are generally both adequate to cover a large set of local features and sufficient for a fine-grained feature-based analysis.

A snippet *does* include preceding or in-between lines that are not simple statements, such as comments, function headers, blank lines, or headers of compound statements like **if-else**, **try-catch**, **while**, **switch**, and **for**. Furthermore, we do not allow snippets to cross scope boundaries. That is, a snippet neither spans multiple methods nor starts inside a compound statement and then extends outside it (however, we do permit snippets to start outside of a compound statement but end before the statement is complete). We find that with this set of policies, over 90% of statements in all of the programs we considered (see Section 2.5) are candidates for incorporation in some snippet. The few non-candidate lines are usually found in functions that have fewer than three statements. This snippet definition is specific to Java but extends to similar languages like C++ and C#. We find the size distribution (in number of characters) for the 100 snippets generated for this study to be approximately normal, but with a positive skew (mean of 278.94, median of 260, minimum of 92 and maximum of 577).

The snippets were generated from five open source projects (see Table 2.7). They were chosen to include varying levels of maturity and multiple application domains with the goal to keep the model generic and widely-applicable.

2.2.2 Readability Scoring

Prior to their participation, our volunteer human annotators were told that they would be asked to rate Java code on its readability, and that their participation would assist in a study of that aspect of software quality. Responses were collected using a web-based annotation tool (shown in Figure A.1) that users were permitted to access without time pressure. Participants were presented with a sequence of snippets and buttons labeled 1–5 [81]. Each participant was shown the same set of one hundred snippets in the same order. Participants were graphically reminded that they should select a number near five for “more readable”

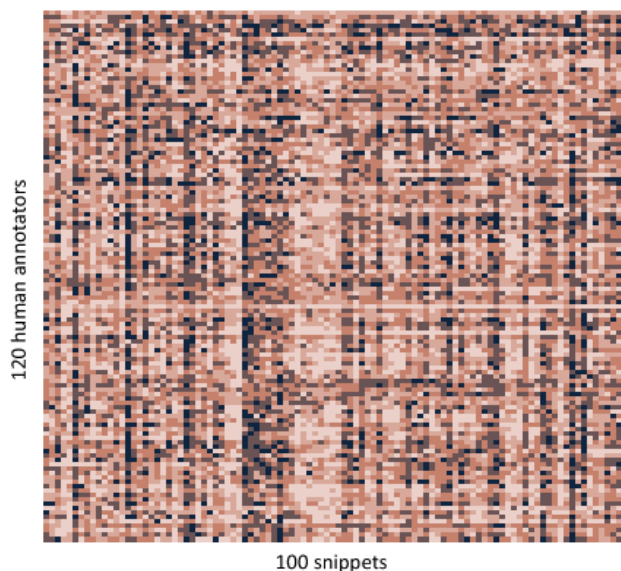


Figure 2.1: The complete data set obtained for this study. Each box corresponds to a judgment made by a human annotator. Darker colors correspond to lower readability scores (e.g., 1 and 2) the lighter ones correspond to higher scores. The vertical bands, which occur much more frequently here than in a figure with random coloration, indicate snippets that were judged similarly by many annotators. Our metric for readability is derived from these 12,000 judgments.

snippets and a number near one for “less readable” snippets, with a score of three indicating neutrality. Additionally, there was an option to skip the current snippet; however, it was used very infrequently (15 times in 12,000). Snippets were not modified from the source, but they were syntax highlighted to better simulate the way code is typically viewed.¹ Finally, clicking on a “help” link reminded users that they should score the snippets “based on [their] estimation of readability” and that “readability is [their] judgment about how easy a block of code is to understand.” Readability was intentionally left formally undefined in order to capture the unguided and intuitive notions of participants.

2.2.3 Study Participation

The study was advertised at several computer science courses at *The University of Virginia*. As such, participants had varying experience reading and writing code: 17 were taking first-year courses, 63 were taking second year courses, 30 were taking third or fourth year courses, and 10 were graduate students. In total, 120 students participated. The study ran from Oct 23 to Nov 2, 2008. Participants were told that all respondents would receive \$5 USD, but that the fifty people who *started* (not finished) the survey the earliest would receive \$10 USD. The use of start time instead of finish time encouraged participation without

¹Both syntax highlighting and the automatic formatting of certain IDEs (e.g., Eclipse) may change the perceived readability of code. The vast majority of editors feature syntax highlighting, however automatic formatting is not as universally used. We thus present snippets with syntax highlighting but without changes to formatting.

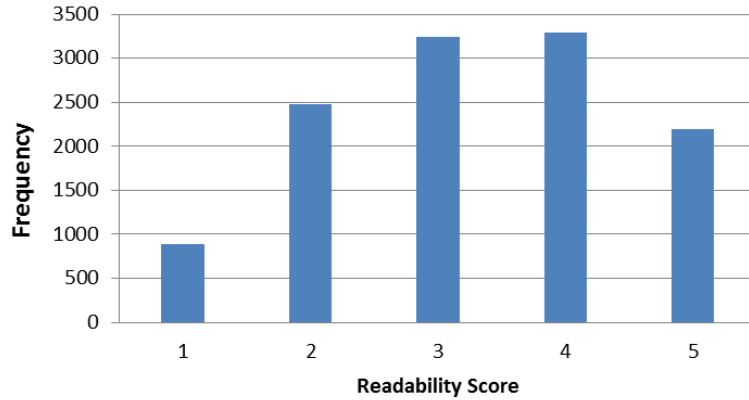


Figure 2.2: Distribution of readability scores made by 120 human annotators on code snippets taken from several open source projects (see Table 2.7).

placing any time pressure on the activity itself (e.g., there was no incentive to make rushed readability judgments); this protocol was made clear to participants. All collected data were kept carefully anonymous, and participants were aware of this fact: completing the survey yielded a randomly-generated code that could be monetarily redeemed. In Section 2.4.2 we discuss the effect of experience on readability judgments. In Section 2.7 we discuss the implications of our participant pool on experiment validity.

2.3 Study Results

Our 120 annotators each scored 100 snippets for a total of 12,000 distinct judgments. Figure 2.1 provides a graphical representation of this publicly-available data.² The distribution of scores can be seen in Figure 2.2.

First, we consider inter-annotator agreement, and evaluate whether we can extract a single coherent model from this data set. For the purpose of measuring agreement, we consider several correlation statistics. One possibility is Cohen’s κ which is often used as a measure of inter-rater agreement for categorical items. However, the fact that our judgment data is ordinal (i.e., there is a qualitative relationship and total order between categories) is an important statistical consideration. Since the annotators did not receive precise guidance on how to score snippets, absolute score differences are not as important as relative ones. If two annotators both gave snippet X a higher score than snippet Y , then we consider them to be in agreement with respect to those two snippets, even if the actual numerical score values differ. Thusly, in this study we tested a linear-weighted version of κ (which conceptually gives some credit for rankings that are “close”). In addition, we considered Kendall’s τ (i.e., the number of bubble-sort operations to order one list in the same way as a second), Pearson’s r (measures the degree of linear dependence) and Spearman’s ρ (the degree of

²The dataset and our tool are available at <http://www.cs.virginia.edu/~weimer/readability>

Statistic	Avg—Humans	Avg—Model
Cohen’s κ	0.18 (p=0.0531)	0.07 (p=0.0308)
Weighted κ	0.33 (p=0.0526)	0.27 (p=0.0526)
Kendall’s τ	0.44 (p=0.0090)	0.53 (p<0.0001)
Pearson’s r	0.56 (p=0.0075)	0.63 (p<0.0001)
Spearman’s ρ	0.55 (p=0.0089)	0.71 (p<0.0001)

Table 2.1: Five statistics for inter-annotator agreement. The “Avg—Humans” Column gives the average value of the statistic when applied between human annotator scores and the average human annotator score (or mode in the case of κ). The “Avg—Model” column show the value of the statistic between our model’s output and the average (or mode) human readability scores. In both cases we give the corresponding p-values for the null-hypothesis (the probability that the correlation is 0).

dependence with any arbitrary monotonic function, closely related to Pearson’s r) [82]. For these statistics, a correlation of 1 indicates perfect correlation, and 0 indicates no correlation (i.e., uniformly random scoring with only random instances of agreement). In the case of Pearson, a correlation of 0.5 would arise, for example, if two annotators scored half of the snippets exactly the same way, and then scored the other half randomly.

We can combine our large set of judgments into a single model simply by averaging them. Because each of the correlation statistics compares the judgments of two annotators at a time. We extend it by finding the average (in the case of κ and Weighted κ we use *mode* discrete values are expected) pairwise correlation between our unified model and each annotator. Our method is similar to previous work by Kashden *et al.* [83] who use pairwise Cohen’s κ to compute inter-rater reliability. Wasfi *et al.* [84] use pairwise Spearman’s ρ both to compute inter-rater agreement but also to evaluate their proposed model for sarcoidosis severity.

Correlation results are tabulated in Table 2.1. Translating these statistics into qualitative terms is difficult, but correlation greater than 0.5 (Pearson/Spearman) is typically considered to be moderate to strong for a human-related study [85]. We use this unified model in our subsequent experiments and analyses. We employ Spearman’s ρ throughout this study as our principle measure of agreement because it is the most general, and it appears to model the greatest degree of agreement. Figure 2.3 shows the range of agreements.

This analysis seems to confirm the widely-held belief that humans agree significantly on what readable code looks like, but not to an overwhelming extent. One implication is that there are, indeed, underlying factors that influence readability of code. By modeling the average score, we can capture most of these common factors, while simultaneously omitting those that arise largely from personal preference.

2.4 Readability Model

We have shown that there is significant agreement between our group of annotators on the relative readability of snippets. However, the processes that underlie this correlation are unclear. In this section, we explore the

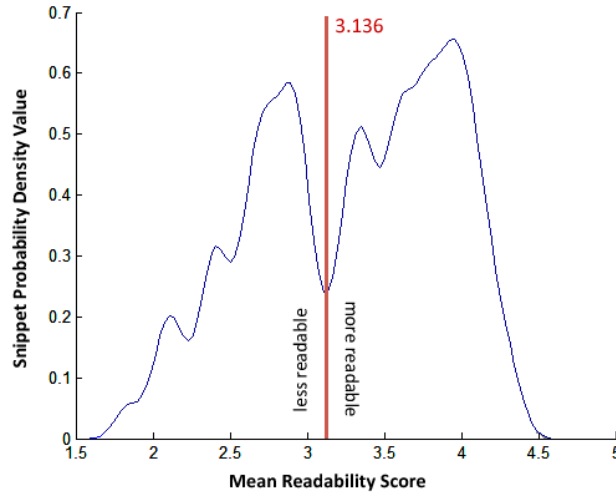


Figure 2.3: Distribution of the average readability scores across all the snippets. The bimodal distribution presents us with a natural cutoff point from which we can train a binary classifier. The curve is a probability-density representation of the distribution with a window size of 0.8.

extent to which we can mechanically predict human readability judgments. We endeavor to determine which code features are predictive of readability, and construct a model (i.e., an automated software readability metric) to analyze other code.

2.4.1 Model Generation

First, we form a set of features that can be detected statically from a snippet or other block of code. We have chosen features that are relatively simple, and that intuitively seem likely to have some effect on readability. They are factors related to structure, density, logical complexity, documentation, and so on. Importantly, to be consistent with our notion of readability as discussed in Section 2.2.1, each feature is independent of the size of a code block. Table 2.2 enumerates the set of code features that our metric considers when judging code readability. Each feature can be applied to an arbitrary sized block of Java source code, and each represents either an average value per line, or a maximum value for all lines. For example, we have a feature that represents the average number of identifiers in each line, and another that represents the maximum number in any one line. The last two features listed in Table 2.2 detect the character and identifier that occur most frequently in a snippet, and return the number of occurrences found. Together, these features create a mapping from snippets to vectors of real numbers suitable for analysis by a machine-learning algorithm.

Earlier, we suggested that human readability judgments may often arise from a complex interaction of features, and furthermore that the important features and values may be hard to locate. As a result, simple methods for establishing correlation may not be sufficient. Fortunately, there are a number of machine learning

Avg.	Max.	Feature Name
✓	✓	line length (# characters)
✓	✓	# identifiers
✓	✓	identifier length
✓	✓	indentation (preceding whitespace)
✓	✓	# keywords
✓	✓	# numbers
✓		# comments
✓		# periods
✓		# commas
✓		# spaces
✓		# parenthesis
✓		# arithmetic operators
✓		# comparison operators
✓		# assignments (=)
✓		# branches (if)
✓		# loops (for, while)
✓		# blank lines
	✓	# occurrences of any single character
	✓	# occurrences of any single identifier

Table 2.2: The set of features considered by our metric. Read “#” as “number of ...”

algorithms designed precisely for this situation. Such algorithms typically take the form of a **classifier** which operates on **instances** [86]. For our purposes, an instance is a feature vector extracted from a single snippet. In the training phase, we give a classifier a set of instances along with a labeled “correct answer” based on the readability data from our annotators. The labeled correct answer is a binary judgment partitioning the snippets into “more readable” and “less readable” based on the human annotator data. We designate snippets that received an average score below 3.14 to be “less readable” based on the natural cutoff from the bimodal distribution in Figure 2.3. We group the remaining snippets and consider them to be “more readable.” Furthermore, the use of binary classifications also allows us to take advantage of a wider variety of learning algorithms.

When the training is complete, we apply the classifier to an instance it has not seen before, obtaining an estimate of the probability that it belongs in the “more readable” or “less readable” class. This allows us to use the probability that the snippet is “more readable” as a score for readability. We used the Weka [87] machine learning toolbox.

We build a classifier based on a set of features that have predictive power with respect to readability. To help mitigate the danger of over-fitting (i.e., of constructing a model that fits only because it is very complex in comparison the amount of data), we use 10-fold cross validation [88] (in Section 2.4.2 we discuss the results of a principle component analysis designed to help us understand the true complexity of the model relative to the data). 10-fold cross validation consists of randomly partitioning the data set into 10 subsets, training on

9 of them and testing on the last one. This process is repeated 10 times, so that each of the 10 subsets is used as the test data exactly once. Finally, to mitigate any bias arising from the random partitioning, we repeat the entire 10-fold validation 10 times and average the results across all of the runs.

2.4.2 Model Performance

We now test the hypothesis that local textual surface features of code are sufficient to capture human notions of readability. Two relevant success metrics in an experiment of this type are recall and precision. Here, recall is the percentage of those snippets judged as “more readable” by the annotators that are classified as “more readable” by the model. Precision is the percentage of the snippets classified as “more readable” by the model that were also judged as “more readable” by the annotators. When considered independently, each of these metrics can be made perfect trivially (e.g., a degenerate model that always returns “more readable” has perfect recall). We thus weight them together using the f-measure statistic, the harmonic mean of precision and recall [89]. This, in a sense, reflects the accuracy of the classifier with respect to the “more readable” snippets. We also consider the overall accuracy of the classifier by finding the percentage of correctly classified snippets.

We performed this experiment on ten different classifiers. To establish a baseline, we trained each classifier on the set of snippets with randomly generated score labels. Guessing randomly yields an f-measure of 0.5 and serves as a baseline, while 1.0 represents a perfect upper bound. None of the classifiers were able to achieve an f-measure of more than 0.61 (note, however, that by always guessing ‘more readable’ it would actually be trivial to achieve an f-measure of 0.67). When trained on the average human data (i.e., when not trained randomly), several classifiers improved to over 0.8. Those models included the multilayer perceptron (a neural network), the Bayesian classifier (based on conditional probabilities of the features), a Logistic Regression, and the Voting Feature Interval approach (based on weighted “voting” among classifications made by each feature separately). On average, these three best classifiers each correctly classified between 75% and 80% of the snippets. We view a model that is well-captured by multiple learning techniques as an advantage: if only one classifier could agree with our training data, it would have suggested a lack of generality in our notion of readability.

While an 80% correct classification rate seems reasonable in absolute terms, it is perhaps simpler to appreciate in relative ones. When we compare continuous the output of the Bayesian classifier (i.e., we use a probability estimate of “more readable” rather a binary classification) to the average human score model it was trained against, we obtain a Spearman correlation of 0.71. As shown in Figure 2.4, that level of agreement is 16% greater than the average human in our study produced. Column 3 of Table 2.1 presents the results in

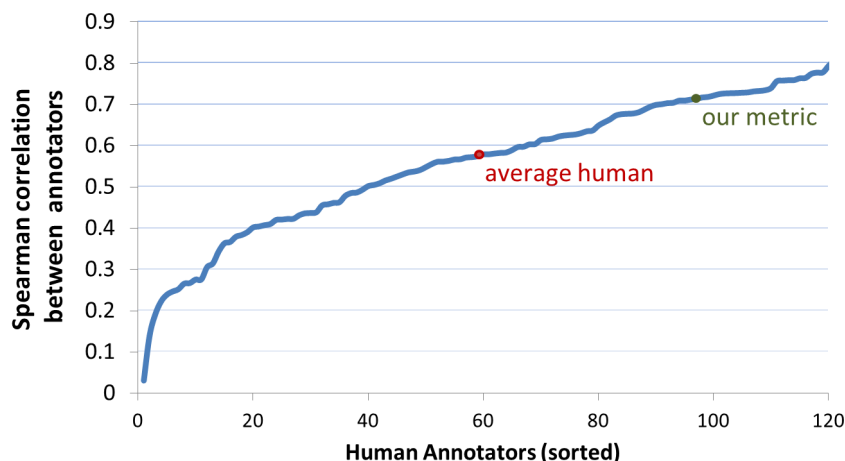


Figure 2.4: Inter-annotator agreement. For each annotator (in sorted order), the value of Spearman’s ρ between the annotator’s judgments and the average judgments of all the annotators (the model that we attempt to predict). Also plotted is Spearman’s ρ for our metric as compared to the average of the annotators.

terms of four other statistics. While we could attempt to employ more exotic classifiers or investigate more features to improve this result, it is not clear that the resulting model would be any “better” since the model is already well within the margin of error established by our human annotators. In other words, in a very real sense, this metric is “just as good” as a human. For performance we can thus select any classifier in that equivalence class, and we choose to adopt the Bayesian classifier for the experiments in this chapter because of its run-time efficiency.

We also repeated the experiment separately with each annotator experience group (e.g., first year CS students, second year CS students). Figure 2.5 shows the mean Spearman correlations. The dark bars on the left show the average agreement between humans and the average score vector for their group (i.e., inter-group agreement). For example, third and fourth year CS students agree with each other more often (Spearman correlation of approximately 0.6) than do first year CS students (correlation under 0.5). The light bar on the right indicates the correlation between our metric (trained on the annotator judgments for that group) and the average of all annotators in the group. Three interesting observations arise. First, for all groups, our automatic metric agrees with the human average more closely than the humans agree. Second, we see a gradual trend toward increased agreement with experience, except for graduate students. We suspect that the difference with respect to graduates may be a reflection of the more diverse background of the graduate student population, their more sophisticated opinions, or some other external factor. And third, the performance of our model is very consistent across all four groups, implying that to some degree it is robust to the source of training data.

We investigated which features have the most predictive power by re-running our all-annotators analysis using only one feature at a time. The relative magnitude of the performance of the classifier is indicative of

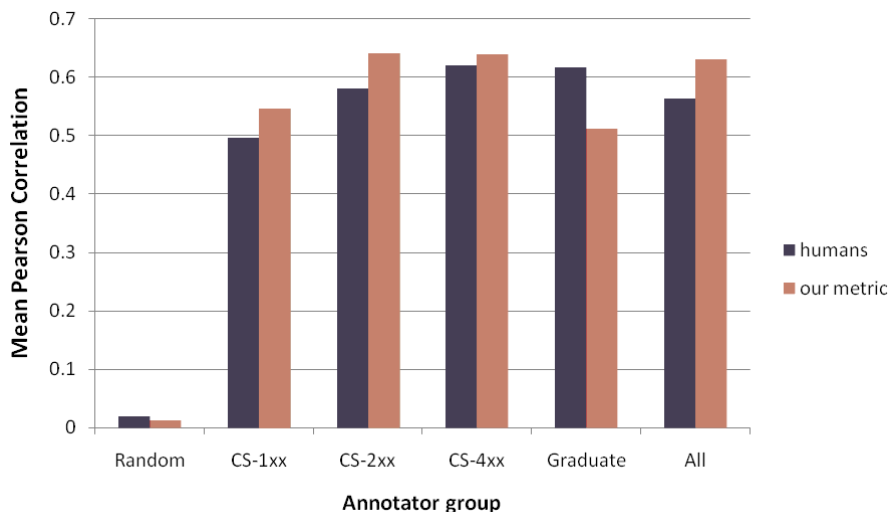


Figure 2.5: Annotator agreement by experience group. “Random” represents a baseline of uniformly distributed random annotations.

the comparative importance of each feature. Figure 2.6 shows the result of that analysis with the magnitudes normalized between zero and one.

We find, for example, that factors like ‘average line length’ and ‘average number of identifiers per line’ are very important to readability. Conversely, ‘average identifier length’ is not, in itself, a very predictive factor; neither are `if` constructs, loops, or comparison operators. Section 2.6 includes a discussion of some of the possible implications of this result. A few of these figures are similar to those used by automated complexity metrics [90]: in Section 2.5.2 we investigate the overlap between readability and complexity.

We prefer this singleton feature analysis to a leave-one-out analysis (which judges feature power based on decreases in classifier performance) that may be misleading due to significant feature overlap. This occurs when two or more features, though different, capture the same underlying phenomena. As a simple example, if there is exactly one space between every two words then a feature that counts words and a feature that counts spaces will capture essentially the same information and leaving one of them out is unlikely to decrease accuracy. A principal component analysis (PCA) indicates that 95% of the total variability can be explained by 8 principal components, thus implying that feature overlap is significant. The total cumulative variance explained by the first 8 principal components is as follows {41%, 60%, 74%, 81%, 87%, 91%, 93%, 95%}.

2.5 Correlating Readability with Software Quality

In the previous section we constructed an automated model of readability that mimics human judgments. We implemented our model in a tool that assesses the readability of programs. In this section we use that tool to

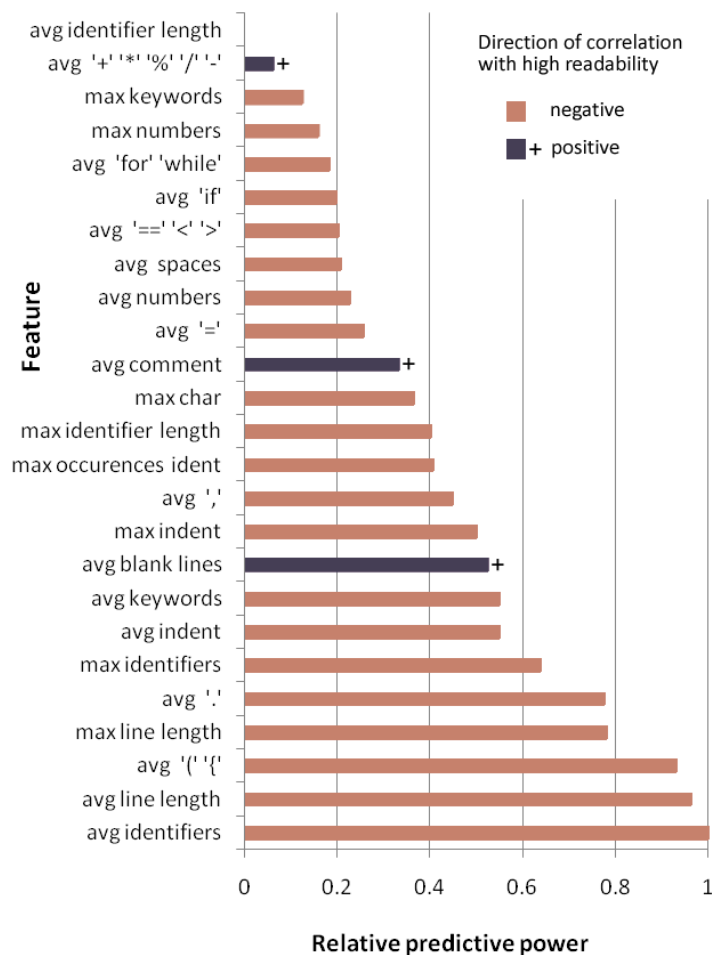


Figure 2.6: Relative power of features as determined by a singleton (one-feature-at-a-time) analysis. The direction of correlation for each is also indicated.

test the hypothesis that readability (as captured by our model) correlates with external conventional metrics of software quality. Specifically, we first test for a correlation between readability and FindBugs, a popular static bug-finding tool [91]. Second, we test for a similar correlation with changes to code between versions of several large open source projects. Third, we do the same for version control log messages indicating that a bug has been discovered and fixed. Then, we examine whether readability correlates with Cyclomatic complexity [90] to test our earlier claim that our notion of readability is largely independent of inherent complexity. Finally, we look for trends in code readability across software projects.

The set of open source Java programs we have employed as benchmarks can be found in Table 2.7. They were selected because of their relative popularity, diversity in terms of development maturity and application domain, and availability in multiple versions from *SourceForge*, an open source software repository. Maturity is self reported, and categorized by *SourceForge* into 1-planning, 2-pre-alpha, 3-alpha, 4-beta,

Project Name	KLOC	Maturity	Description
Azureus: Vuze 4.0.0.4	651	5	Internet File Sharing
JasperReports 2.04	269	6	Dynamic Content
Hibernate* 2.1.8	189	6	Database
jFreeChart* 1.0.9	181	5	Data representation
FreeCol* 0.7.3	167	3	Game
TV Browser 2.7.2	162	5	TV Guide
jEdit* 4.2	140	5	Text editor
Gantt Project 3.0	130	5	Scheduling
SoapUI 2.0.1	98	6	Web services
Data Crow 3.4.5	81	5	Data Management
Xholon 0.7	61	4	Simulation
Risk 1.0.9.2	34	4	Game
JSch 0.1.37	18	3	Security
jUnit* 4.4	7	5	Software development
jMencode 0.64	7	3	Video encoding

Figure 2.7: Benchmark programs used in our experiments. The “Maturity” column indicates a self-reported *SourceForge* project status. *Used as a snippet source.

5-production/stable, 6-mature, 7-inactive. Note that some projects present multiple releases at different maturity levels; in such cases we selected the release for the maturity level indicated.

The execution time of our readability tool (including feature detection and the readability judgment) was relatively short. For example, the 98K lines of code in SOAPUI took less than 16 seconds to process (about 6K LOC per second) on a machine with a 2GHz processor and disk with a maximum 150 MBytes/sec transfer rate.

2.5.1 Readability Correlations

Our first experiment tests for a correlation between defects detected by FINDBUGS with our readability metric at the function level. We first ran FINDBUGS on the benchmark, noting defect reports. Second, we extracted all of the functions and partitioned them into two sets: those containing at least one reported defect, and those containing none. To avoid bias between programs with varying numbers of reported defects, we normalized the function set sizes. We then ran the already-trained classifier on the set of functions, recording an f-measure for “contains a bug” with respect to the classifier judgment of “less readable.” The purpose of this experiment is to investigate the extent to which our model correlates with an external notion of code quality in aggregate.

Our second experiment tests for a similar correlation between “code churn” and readability. Version-to-version changes capture another important aspect of code quality. This experiment used the same setup as the first, but used readability to predict which functions will be modified between two successive releases of a program. For this experiment, “successive release” means the two most recent stable versions. In other words,

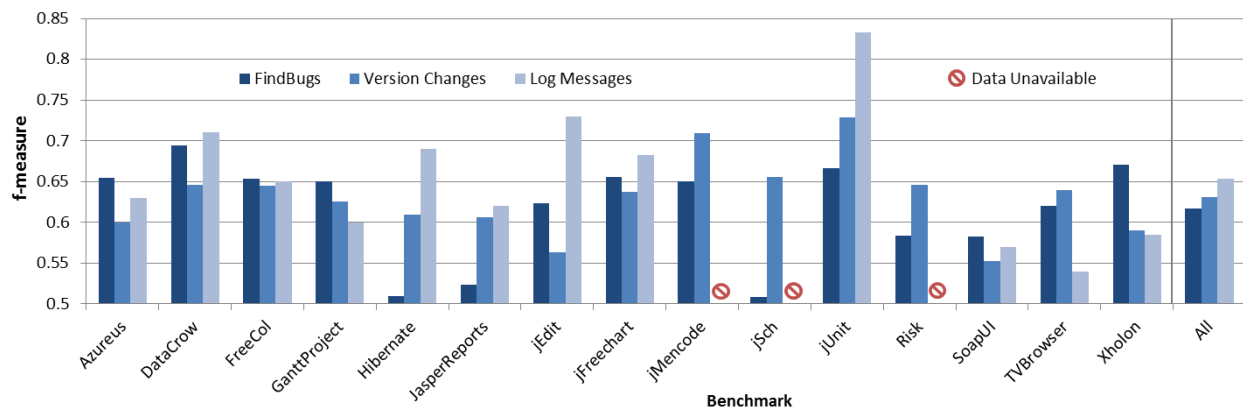


Figure 2.8: f-measure for using readability to predict functions that: show a FINDBUGS defect, have change between releases, and have a defect referenced in a version control log message. For log messages, “Data Unavailable” indicates that not enough version control information was available to conduct the experiment on that benchmark.

instead of “contains a bug” we attempt to predict “is going to change soon.” We consider a function to have changed in any case where the text is not exactly the same, including changes to whitespace. Whitespace is normally ignored in program studies, but since we are specifically focusing on readability we deem it relevant.

While our first experiment looks at output from a bug finder, such output may not constitute true defects in code. Our third experiment investigates the relationship between readability and defects that have actually been noted by developers. It has become standard software engineering practice to use version control repositories to manage modifications to a code base. In such a system, when a change is made, the developer typically includes a log message describing the change. Such log messages may describe a software feature addition, an optimization, or any number of other potential changes. In some cases log messages even include a reference to a “bug” — often, but not always, using the associated bug tracking number from the project’s bug database. In this third experiment we use our metric for readability to predict whether a function has been modified with a log message that mentions a bug.

Twelve of our fifteen benchmarks feature publicly accessible version control systems with at least 500 revisions (most have thousands). For the most recent version of each program, for each line of code, we determine the last revision to have changed that line (e.g., this is similar to the common `cvs blame` tool). We restrict our evaluation to the most recent 1000 revisions of each benchmark and find 5087 functions with reported bugs out of 111,670 total. We then inspect the log messages for the word “bug” to determine whether a change was made to address a defect or not. We partition all functions into two sets: those where at least one line was last changed to deal with a bug, and those where no lines were last changed to deal with a bug.

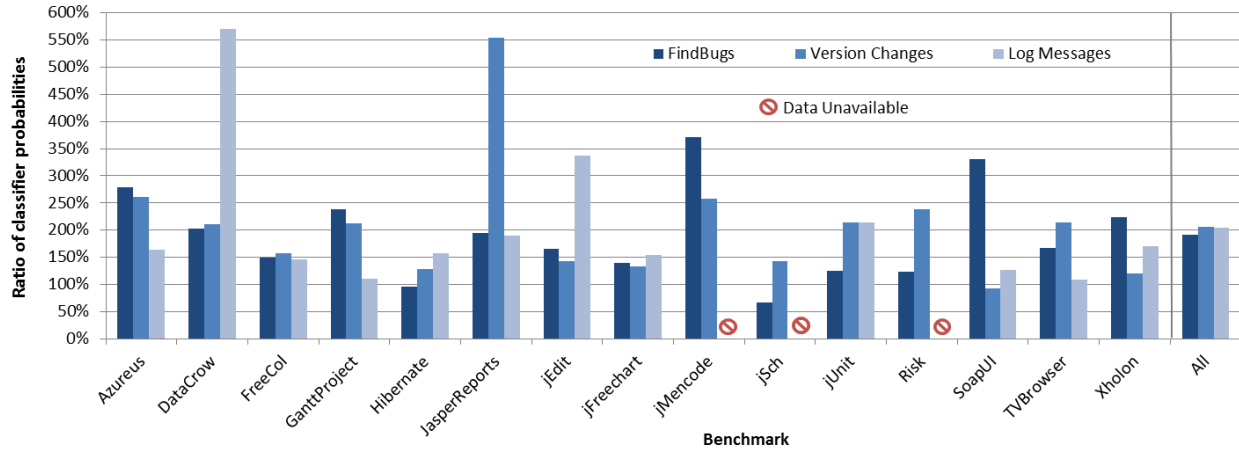


Figure 2.9: Mean ratio of the classifier probabilities (predicting ‘less readable’) assigned to functions that contained a FINDBUGS defect, that will change in the next version, or that had a defect reported in a version control log. For example, FREECOL functions with FINDBUGS errors were assigned a probability of ‘less readable’ that was nearly 150% greater on average than the probabilities assigned to functions without such defects.

Figure 2.8 summarizes the results of these three experiments. The average f-measure over our benchmarks for the FINDBUGS correlation is 0.62 (precision=0.90, recall=0.48), for version changes it is 0.63 (precision=0.89, recall=0.49), and for log messages indicating a bug fix in the twelve applicable benchmarks, the average f-measure is 0.65 (precision=0.92, recall=0.51). It is important to note that our goal is *not* perfect correlation with FINDBUGS or any other source of defect reports: projects can run FINDBUGS directly rather than using our metric to predict its output. Instead, we argue that our readability metric has general utility and is correlated with multiple notions of software quality. This is best shown by our strong performance when correlating with bug-mentioning log messages: code with low readability is significantly more likely to be changed by developers later for the purposes of fixing bugs.

A second important consideration is the *magnitude* of the difference. We claim that classifier probabilities (i.e., continuous output versus discrete classifications) are useful in evaluating readability. Figure 2.9 presents this data in the form of a ratio, the mean probability assigned by the classifier to functions positive for FINDBUGS defects or version changes to functions without these features. A ratio over 1 (i.e., > 100%) for many of the projects indicates that the functions with these features tend to have lower readability scores than functions without them. For example, in the jMencode and SoapUI projects, functions judged less readable by our metric were dramatically more likely to contain FINDBUGS defect reports, and in the JasperReports project less-readable methods were very likely to change in the next version.

As a brief note: we intentionally do not report standard deviations for readability distributions. Both the underlying score distribution that our metric is based on (see Figure 2.3) and the output of our tool itself are

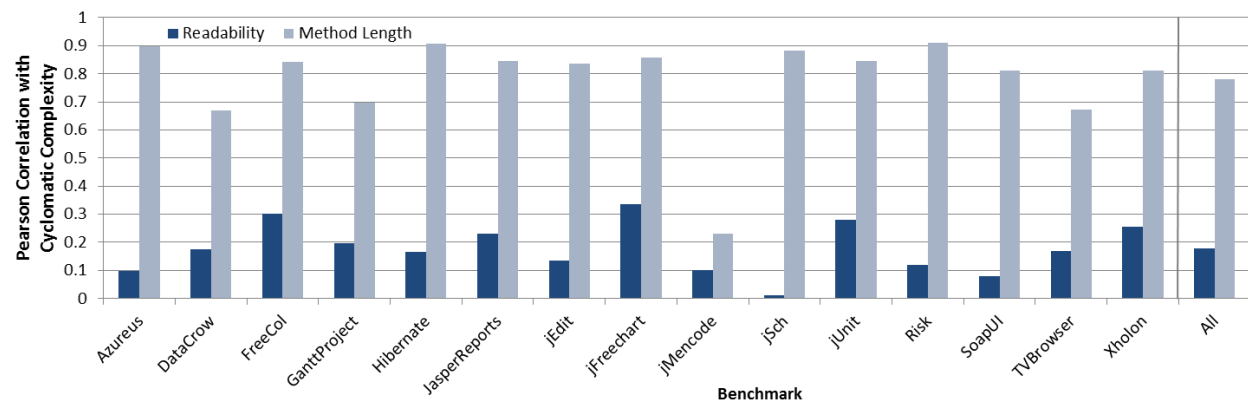


Figure 2.10: Pearson Product moment correlation between Cyclomatic complexity and readability as well as between Cyclomatic complexity and method length (number of statements in the method). Readability is at most weakly correlated with complexity in an absolute sense. In a relative sense, compared to method length, readability is effectively uncorrelated with complexity. These results imply that readability captures an aspect of code that is not well modeled by a traditional complexity measure.

bimodal. In fact, the tool output on our benchmarks more closely approximates a bathtub or uniform random distribution than a normal one. As a result, standard inferences about the implications of the standard deviation do not apply. However, the mean of such a distribution does well-characterize the ratio of methods from the lower half of the distribution to the upper half (i.e., it characterizes the population of 'less-readable' methods compared to 'more-readable' ones).

For each of these three external quality indicators we found that our tool exhibits a substantial degree of correlation. Predicting based on our readability metric yields an f-measure over 0.8 in some case. Again, our goal is not a perfect correlation with version changes and code churn. These moderate correlations do, however, add support to the hypothesis that a substantial connection exists between code readability, as described by our model, and defects and upcoming code changes

2.5.2 Readability and Complexity

In this chapter, we defined readability as the “accidental” component of code understandability, referring to the idea that it is an artifact of writing code and not closely related to the complexity of the problem that the code is meant to solve. Our short snippet selection policy masks complexity, helping to tease it apart from readability. We also selected a set of surface features designed to be agnostic to the length of any selection of code. Nonetheless, some of the features still may capture some amount of complexity. We now test the hypothesis that readability and complexity are *not* closely related.

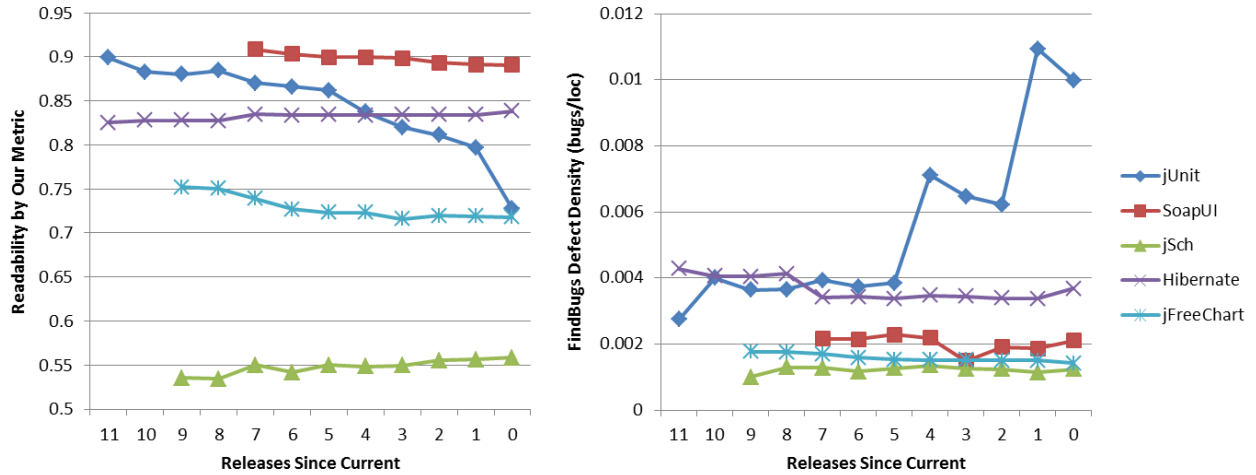


Figure 2.11: The left graph shows the average readability metric of all functions in a project as a function of project lifetime. The right graph shows the FINDBUGS defect density in a project as a function of project lifetime for the same projects and releases. Note that the projects with more constant readabilities have corresponding flat defect densities, while JUNIT (described in text) becomes significantly less readable and significantly more buggy starting at release 5.

We use McCabe’s Cyclomatic measure [90] as an indicative example of an automated model of code complexity. We measured the Cyclomatic complexity and readability of each method in each benchmark. Since both quantities are ordinal (as opposed to binary as in our three previous experiments), we computed Pearson’s r between them. We also computed, for use as a baseline, the correlation between Cyclomatic complexity and method length. Figure 2.10 shows that readability is not closely related to this traditional notion of complexity. Method length, on the other hand, is much more strongly related to complexity. We thus conclude that while our notion of readability is not orthogonal to complexity, it is in large part modeling a distinct phenomena.

2.5.3 Software Lifecycle

To further explore the relation of our readability metric to external factors, we investigate changes over long periods of time. We hypothesize that an observed trend in the readability of a project will manifest as a similar trend in project defects. Figure 2.11 shows a longitudinal study of how readability and defect rates tends to change over the lifetime of a project. To construct this figure we selected several projects with rich version histories and calculated the average readability level over all of the functions in each. Each of the projects can be modeled with a linear relationship at a statistical significance level (p-value) of less than 0.05 except for defect density with JFREECHART.

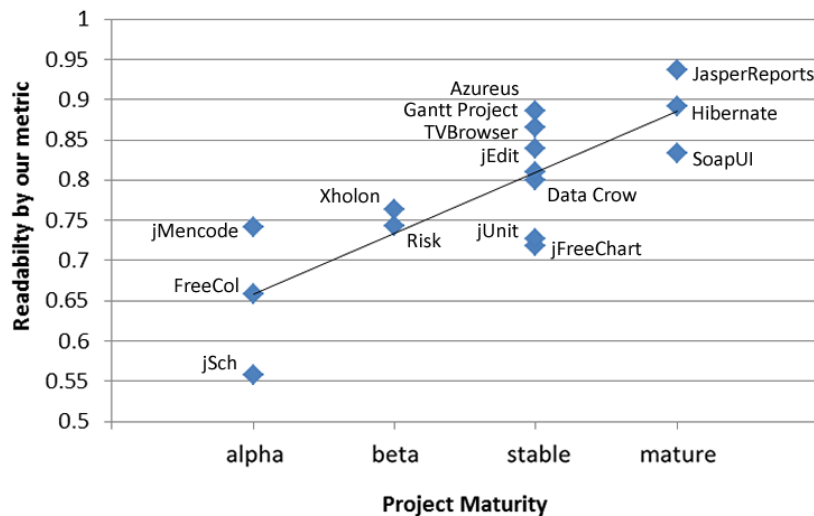


Figure 2.12: Average readability metric of all functions in a project as a function of self-reported project maturity with best fit linear trend line. Note that projects of greater maturity tend to exhibit greater readability.

Note that newly-released versions for open source projects are not always more stable than their predecessors. Projects often undergo major overhauls or add additional cross cutting features. Consider JUNIT, which has recently adopted a “completely different API ... [that] depends on new features of Java 5.0 (annotations, static import...)” [92].

The rightmost graph in Figure 2.11 plots FINDBUGS-reported defect density over multiple project releases. Most projects, such as HIBERNATE and SOAPUI, show a relatively flat readability profile. These projects experience a similarly flat defect density. The JUNIT project, however, shows a sharp decline in readability from releases 5 to 0 (as the new API is introduced) and a corresponding increase in defect density from releases 5 to 0. For JUNIT, we observe that 58.9% of functions have readability below 0.5 in the earliest version we tested, and 71.4% in the most recent (an increase of 12.5%). Of the remaining four systems, SOAPUI had the greatest change: 44.3% below 0.5 to 48.1% (an increase of 3.8%). In this case study, our readability metric correlates strongly with software quality metrics both between different projects and also between different releases of the same project.

We conducted one additional study to measure readability against maturity and stability. Figure 2.12 plots project readability against project maturity, as self-reported by developers. The data shows a noisy, but statistically significant (Pearson’s $r = 0.80$ with $p = 0.00031$), upward trend implying that projects that reach maturity tend to be more readable. For example, every “6-mature” project is more readable than every “3-alpha” project.

2.6 Discussion

This work uses multiple methods to investigate the relationship between local code features and readability. We believe that this information may have implications for the way code should be written and evaluated, and for the design of programming languages. However, we caution that this data may only be truly relevant to our annotators; it should not be interpreted to represent a comprehensive or universal model for readability. Furthermore, by the nature of a descriptive model, it may not be suitable for directly prescribing coding practices. However, we believe it can be useful to identify aspects of code readability which should be more carefully considered.

The length of identifier names had no significant relationship to human annotator readability scores. This observation fails to support the common belief that “single-character identifiers ... [make the] ... maintenance task much harder” [62]. An observation which perhaps contributed to a significant movement toward “self documenting code” which is often characterized by long and descriptive identifier names and few abbreviations. The movement has had particular influence on the Java community. Furthermore, naming conventions, like the “Hungarian” notation which seeks to encode typing information into identifier names, should be considered carefully [93]. In our study, the average identifier length had near zero predictive power, while maximum identifier length was much more useful as a negative predictor. While we did not include any features to detect encoded type information or other variable naming conventions, paradigms that result in longer identifiers without conveying additional information may negatively impact readability.

Unlike identifiers, comments are a very direct way of communicating intent. One might expect their presence to increase readability dramatically. However, we found that comments were only moderately well-correlated with our annotators’ notion of readability (33% relative power). One conclusion may be that while comments can enhance readability, they are typically used in code segments that started out less readable: the comment and the unreadable code effectively balance out. The net effect would appear to be that comments are not always, in and of themselves, indicative of high or low readability.

The number of identifiers and characters per line has a strong influence on our readability metric (100% and 96% relative power respectively). It would appear that just as long sentences are more difficult to understand, so are long lines of code. Our findings support the conventional wisdom that programmers should keep their lines short, even if it means breaking up a statement across multiple lines.

When designing programming languages, readability is an important concern. Languages might be designed to force or encourage improved readability by considering the implications of various design and language features on this metric. For example, Python enforces a specific indentation scheme in order to aid comprehension [94, 29]. In our experiments, the importance of character count per line suggests that

languages should favor the use of constructs, such as switch statements and pre- and post-increment, that encourage short lines.

It is worth noting that our model of readability is *descriptive* rather than *normative* or *prescriptive*. That is, while it can be used to predict human readability judgments for existing software, it cannot be directly interpreted to prescribe changes that will improve readability. For example, while “average number of blank lines” is a powerful feature in our metric that is positively correlated with high readability, merely inserting five blank lines after every existing line of code need not improve human judgments of that code’s readability. Similarly, long identifiers contribute to lower readability scores in our model, but replacing all identifiers with random two-letter sequences is unlikely to help. We might tease apart such relationships and refine our model by applying model-prescribed changes to pieces of code and then evaluating their actual change in readability via a second human study: differences between predicted changes and observed changes will help illuminate confounding variables and imprecise features in our current model. Learning such a normative model of software readability remains as future work.

Finally, as language designers consider new language features, it might be useful to conduct studies of the impact of such features on readability. The techniques presented in this chapter offer a framework for conducting such experiments.

2.7 Threats to Validity

One potential threat to validity concerns the pool of participants for our study. In particular, our participants were taken largely from introductory and intermediate computer science courses, implying that they have had little previous experience reading or writing code as compared to industrial practitioners. Furthermore, they may possess some amount of bias to certain coding practices or idioms resulting from a uniform instruction at the same institution.

Because our model is built only upon the opinions of computer science students at *The University of Virginia*, it is only a model of readability according to them. Nonetheless, the metric we present shows significant correlation with three separate external notions of software quality. Furthermore, our study shows that even graduate students, who have widely varying educational and professional backgrounds, show a strong level of agreement with each other and with our metric. This indicates that annotator bias of this type may be small; however, we did not study it directly in this chapter. Finally, rather than presenting a final or otherwise definitive metric for readability, we present an initial metric coupled with a methodology for constructing further metrics from an arbitrary population.

We also consider our methodology for scoring snippets as a potential threat to validity. To capture an intuitive notion of readability, rather than a constrained one, we did not provide specific guidance to our participants on how to judge readability. It is possible that inter-annotator agreement would be much greater in a similar study that included a precise definition of readability (e.g., a list of factors to consider). It is unclear how our modeling technique would perform under those circumstances. Similarly, both practice and fatigue factors may affect the level of agreement or bias the model. However, we do not observe a significant trend in the variance of the score data across the snippet set (a linear regression on score variance has a slope of 0.000 and R-squared value of 0.001, suggesting that practice effects were minimal).

2.8 Conclusion

In this chapter we have presented a technique for modeling code readability based on the judgments of human annotators. In a study involving 120 computer science students, we have shown that it is possible to create a metric that agrees with these annotators as much as they agree with each other by only considering a relatively simple set of low-level code features. In addition, we have seen that readability, as described by this metric, exhibits a significant level of correlation with more conventional metrics of software quality, such as defects, code churn, and self-reported stability. Furthermore, we have discussed how considering the factors that influence readability has potential for improving the programming language design and engineering practice with respect to this important dimension of software quality. Finally, it is important to note that the metric described in this chapter is not intended as the final or universal model of readability.

Chapter 3

Estimating Path Execution Frequency

“Two roads diverged in a wood, and I,
I took the one less traveled by . . .”
– *Robert Frost, 'The Road Not Taken'*

3.1 Introduction

COMPREHENSION of software requires not only parsing the program text, but also interpreting how the running program will behave. While we have shown that a surface-level idea like readability can be powerful, it is strictly limited by not considering the meaning of the code. The importance of understanding program behavior is evidenced by the empirical observation that most or all of the execution time of a typical program is spent along a small percentage of **program paths**. That is, certain sequences of instructions tend to repeat often. Such paths are known as *hot paths* [95, 96, 97, 98].

Characterizing the runtime behavior of software artifacts is an important concern for tasks far beyond program comprehension; for propositions as diverse as code optimization [99, 100], maintenance [101] and general data-flow analysis [102]. As a result, program profiling remains a vibrant area of research (e.g., [96, 103, 104]) even after many years of treatment from the community. Unfortunately, profiling is severely limited by practical concerns: the frequent lack of appropriate workloads for programs, the questionable degree to which they are indicative of actual usage, and the inability of such tools evaluate program modules or individual paths in isolation.

In a static context, without workloads, there is no clear notion of the relative execution frequency of control-flow paths. The frequency with which any given block of code will be executed is unknown. Without

additional information all paths can only be assumed equally likely. This practice is questionable, particularly when an analysis is designed to measure or affect performance. It is reported that large amounts of code exists to handle “exceptional” cases only, with 46%–66% of code related to error handling and 8%–16% of all methods containing error handling constructs (see [105, p. 4] for a survey). Such error handling code is common, but does not significantly impact real performance [106]. Therefore, the assumption that all paths are equally likely or that input space is uniform leads to inaccurate or otherwise suboptimal results in program analysis.

We propose a fully static approach to the problem of hot path identification. Our approach is based on two key insights. First, we observe a relationship between the relative frequency of a path and its effect on program state. Second, we carefully choose the right level of abstraction for considering program paths: interprocedural for precision, but limited to one class for scalability.

Program State. We observe that there are static source-level characteristics that can indicate whether a program path was “intended” to be common, or alternatively, if it was designed to be rare or “exceptional.” We claim that infrequent cases in software typically take one of two forms. The first involves error detection followed by a divergence from the current path, such as returning an error code as a response to an invalid argument or raising an exception after an assertion failure. The second involves significant reconfiguration of program state, such as reorganizing a data structure, reloading data from a file or resizing a hash table. Conversely, common cases typically entail relatively small state modifications, and tend to gradually increase available context, subtly increasing system entropy. We hypothesize that paths which exhibit only small impacts on program state, both in terms of global variables and in terms of context and stack frames, are more likely to be hot paths.

Program Paths. When predicting relative path execution frequency, the definition of “path” is critical. In their seminal work on `gprof`, Graham *et al.* note, “to be useful, an execution profiler must attribute execution time in a way that is significant for the logical structure of a program as well as for its textual decomposition.” [103] Flat profiles are not as helpful to programmers or as accurate as interprocedural call-graph based profiles. We consider method invocations between methods of the same class to be part of one uninterrupted path; this choice allows us to be very precise in common object-oriented code. For the purposes of our analysis we split paths when control flow crosses a class boundary; this choice allows us to scale to large programs.

These two insights form the heart of our static approach to predicting dynamic execution frequencies. In this chapter we enumerate a set of static source-level features on control flow paths. Our **features** are based on structure, data flow, and effect on program state. We employ these features to build a **descriptive models** for runtime path frequency. Our approach reports, for each path, a numerical estimate of runtime

frequency relative either to the containing method, or to the entire program. We evaluate our prototype implementation on the SPECjvm98 benchmarks, using their supplied workloads as the ground truth. We also discuss and evaluate the relative importance of our features.

The main contributions of this chapter are:

- A technique for statically estimating the runtime frequency of program paths. Our results can be used to support or improve many types of static analyses.
- An empirical evaluation of our technique. We measure accuracy both in selecting hot paths and in ranking paths by frequency. We demonstrate the flexibility of our model by employing it to characterize the running time and dynamic branching behavior of several benchmarks. The top 5% of paths as reported by our metric account for over half of program runtime; a static branch predictor based on our technique has a 69% hit rate compared to the 65% of previous work.

The structure of this chapter is as follows. We discuss related work, present a motivating example, and enumerate several potential uses of our technique in Section 3.2. After describing our process for program path enumeration in Section 3.3, we discuss the features that make up our model in Section 3.4. We perform a direct model evaluation in Section 3.5. We then perform a timing based experiment (Section 3.6) and a branch prediction experiment (Section 3.7). Finally, we extract and discuss possible implications of our model (Section 3.8) and then conclude.

3.2 Context and Motivation

In this section, we present key related work, review a simple example that demonstrates the application and output of our technique, and then highlight a number of tasks and analyses that could benefit from a static prediction of dynamic path frequencies.

In recent years multiple research efforts have explored the problem of predicting program behavior. Bowring *et al.* [107] present a technique that models program executions as Markov models, and a clustering method for Markov models that aggregates multiple program executions into effective behavior classifiers. Classifiers built by their technique are good predictors of program behavior, however, their technique relies on actual prior program runs where ours is purely static.

Wall *et al.* [108] presents a technique for estimating profile information from simple heuristics. For example, the number of times each basic block will be executed is estimated by counting the number of static calls to the block's containing procedure. Wall found that real profiles from different runs worked much better than the estimated profiles for optimization tasks, however. We propose to address a similar problem, but use a much more sophisticated model based on machine learning to provide more accurate predictions.

Monsifrot *et al.* [109] use decision trees to learn heuristics for loop unrolling. Their proposed model is based on surface-level compile time features (e.g., *number of statements*). Monsifrot’s results indicate that machine learning can provide significantly better choices than existing compiler heuristics for at least one class of optimizations. We extend this basic idea but instead to model a much broader notion of runtime behavior.

The work most similar to ours is *static branch prediction*, a problem first explored by Ball and Larus [110]. They showed that simple heuristics are useful for predicting the frequency at which branches are taken. The heuristics typically involve surface-level features such as, “if a branch compares a pointer against null or compares two pointers, predict the branch on false condition as taken.” Extensions to this work have achieved modest performance improvements by employing a larger feature set and neural network learning [111].

We have chosen to focus on paths instead of branches for two reasons. First, we claim that path-based frequency can be more useful in certain static analysis tasks; see Ball *et al.* [112] for an in-depth discussion. Second, paths contain much more information than individual branches, and thus are much more amenable to the process of formal modeling and prediction.

Automatic *workload generation* is one strategy for coping with a lack of program traces. This general technique has been adapted to many domains [113, 114, 115]. While workload generation is useful for stress testing software and for achieving high path coverage, it has not been shown suitable for the creation of indicative workloads. In contrast, we attempt to model indicative execution frequencies.

More recent work in *concolic testing* [116, 117] explores all execution paths of a program with systematically selected inputs to find subtle bugs. The technique interleaves static symbolic execution to generate test inputs and concrete execution on those inputs. Symbolic values that are too complex for the static analysis to handle are replaced by concrete values from the execution. Our approach makes static predictions about dynamic path execution frequency, but does not focus on bug-finding.

3.2.1 Motivating Example

We hypothesize that information about the expected runtime behavior of imperative programs, including their relative path execution frequency, is often embedded into source code by developers in an implicit, but nonetheless predictable, way. We now present an intuition for how this information is manifested.

Figure 3.1 shows a typical example of the problem of hot path identification in a real-world algorithm implementations. In this case there are three paths of interest. The path corresponding to insertion of the new entry on lines 26–27 is presumably the “common case.” However, there is also the case in which the

```

1  /**
2  * Maps the specified key to the specified
3  * value in this hashtable ...
4  */
5  public synchronized V put(K key, V value) {
6
7      // Make sure the value is not null
8      if (value == null) {
9          throw new NullPointerException();
10     }
11
12     ...
13
14     modCount++;
15     if (count >= threshold) {
16         // Rehash the table if the
17         // threshold is exceeded
18         rehash();
19
20         tab = table;
21         index = (hash & 0x7FFFFFFF) % tab.length;
22     }
23
24     // Creates the new entry.
25     Entry<K,V> e = tab[index];
26     tab[index] =
27         new Entry<K,V>(hash, key, value, e);
28     count++;
29     return null;
30 }

```

Figure 3.1: The `put` method from the Java SDK version 1.6’s `java.util.Hashtable` class. Some code has been omitted for illustrative simplicity.

`value` parameter is `null` and the function terminates abruptly on line 9, as well as the case where the `rehash` method is invoked on line 18, an operation likely to be computationally expensive.

This example helps to motivate our decision to follow paths within class boundaries. We leverage the object-oriented system paradigm whereby code and state information are encapsulated together. Following paths within classes enables our technique to discover that `rehash` modifies a large amount of program state. Not tracing all external method invocations allows it to scale.

Our technique identifies path features, such as “throws an exception” or “reads many class fields”, that we find are indicative of runtime path frequency. The algorithm we present successfully discovers that the path that merely creates a new entry is more common than the `rehash` path or the exceptional path. We now describe some applications that might make use of such information.

3.2.2 Example Client Applications

Profile-guided optimization refers to the practice of optimizing a compiled binary subsequent to observing its runtime behavior on some workload (e.g., [118, 100, 99]). Specific optimizations include the selection of variables to promote to registers [119], placement of code to improve cache behavior [120, 121], and prediction of common control paths for optimizations including loop unrolling [122]. Our technique has the potential to make classes of such optimization more accessible; first, by eliminating the need for workloads, and second by removing the time required to run them and record profile information. A static model of relative path frequency could help make profile-guided compiler optimizations more mainstream.

Current work in computational *complexity estimation* has been limited to run-time analyses [123]. At a high level, the task of estimating the complexity of the `put` procedure in Figure 3.1 requires identifying the paths that represent the common case and thus dominate the long-run runtime behavior. This is critical, because mistakenly assuming that the path along which `rehash` is invoked is common, will result in a significant over-estimate of the cost of `put`. Similarly, focusing on the path corresponding to the error condition on line 9 will underestimate the cost.

Static *specification mining* [124, 56], which seeks to infer formal descriptions of correct program behavior, can be viewed as the problem of distinguishing correct paths from erroneous ones in programs that contain both. Behavior that occurs on many paths is assumed to be correct, and deviations may indicate bugs [18]. In practice, static specification miners use static path counts as a proxy for actual execution behavior; a static model of path frequency would improve the precision of such techniques.

Many static analyses produce false alarms, and *statistical ranking* is often used to sort error reports by placing likely bugs at the top [125]. These orderings often use static proportion counts and *z*-rankings based on the assumption that all static paths are equally likely. A static prediction of path frequency would allow bug reports on infrequent paths, for example, to be filtered to the bottom of the list.

In general any program analysis or transformation that combines static and dynamic information could benefit from a model of path frequency. Areas as disparate as *program specialization* [126], where frequency predictions could reduce the annotation and analysis burden for indicating what to specialize, and *memory bank conflict detection* in GPU programs [127], where frequency predictions could reduce the heavy simulation and deployment costs of static analysis, are potential clients. In the next sections we describe the details of our analysis.

3.3 Path Enumeration

In an imperative language, a method is comprised of a set of control flow paths, only one of which will be executed on any actual invocation. Our goal is to statically quantify the relative runtime frequency with which each path through a method will be taken. Our analysis works by analyzing features along each path.

Our algorithm operates on a per-class-declaration basis. This enables it to consider interactions between the methods of a single class that are likely to assist in indicating path frequencies, but avoids the large explosion of paths that would result if it were fully context sensitive.

Central to our technique is *static path enumeration*, whereby we enumerate all acyclic intra-class paths in a target program. To experimentally validate our technique, we also require a process for *dynamic* path enumeration that counts the number of times each path in a program is executed during an actual program run.

3.3.1 Static Path Enumeration

Static intra-class path enumeration begins at each member function of each concrete program class. A *static intra-class path* is a flat, acyclic whole-program path that starts at a public method of a class T and contains only statements within T and only method invocations to methods outside of T . We first enumerate the paths in each method separately; we then combine them to obtain intra-class paths.

For a given method, we construct the control flow graph and consider possible paths in turn. A method with loops may have an unbounded number of potential paths; we do not follow back edges to ensure that the path count is finite. For the purposes of our analysis, a path through a loop represents all paths that take the loop one or more times. We find that this level of description is adequate for many client analyses, but higher bounds on the number of loop iterations are also possible. Our path definition is similar to the one used in efficient path profiling [96], but richer in that we consider certain interprocedural paths. We process, record and store information about each path as we enumerate it.

Once all intra-method paths are enumerated they can be merged, or “flattened”, into intra-class paths. We use a fix-point worklist algorithm. We iterate through each path, and at intra-class invocation sites we “splice-in” the paths corresponding to the invoked method, cloning the caller path for each invoked callee path. The process terminates when all such invocation sites in all paths have been resolved. Since we do not allow a statement to appear more than once on any single path and the number of invocation sites is finite, this process always terminates.

Figure 3.2 shows the pseudocode for our static path enumeration algorithm. While our static intra-class path definition is intuitive, we show the enumeration process because our path frequency estimates are given

Input: Concrete Class T .
Output: Mapping $P : \text{method} \rightarrow \text{static intra-class path set}$

```

1: for all methods  $meth$  in  $T$  do
2:    $P(meth) \leftarrow$  acyclic paths in  $meth$ 
3: end for
4: let  $Worklist \leftarrow$  methods in  $T$ 
5: while  $Worklist$  is not empty do
6:   let  $meth \leftarrow$  remove next from  $Worklist$ 
7:   for all paths  $path$  in  $P(meth)$  do
8:     let  $changed \leftarrow \text{false}$ 
9:     for all method invocations  $callee()$  in  $path$  do
10:      if  $callee \in T$  then
11:        for all paths  $path_c$  in  $P(callee)$  do
12:          let  $path' \leftarrow$  copy of  $path$ 
13:           $path' \leftarrow path'[callee() \mapsto path_c]$ 
14:           $P(meth) \leftarrow P(meth) \cup \{path'\} - \{path\}$ 
15:           $changed \leftarrow \text{true}$ 
16:        end for
17:      end if
18:    end for
19:    if  $changed$  then
20:       $Worklist \leftarrow Worklist \cup \{meth\}$ 
21:    end if
22:  end for
23: end while

```

Figure 3.2: High-level pseudocode for enumerating static intra-class paths for a concrete class T .

only for static paths we enumerate and our rankings of path frequencies are given only relative to other static paths. Our intra-class path definition thus influences the results of our analysis.

3.3.2 Dynamic Path Enumeration

To train and evaluate our model of path execution frequency we require “ground truth” for how often a given static path is actually executed on a typical program run. Our dynamic path enumeration approach involves two steps. First, we record each statement visited on a program run; before executing a statement we record its unique identifier, including the enclosing class and method. One run of the program will yield one single list of statements.

Second, we partition that list into static paths by processing it in order while simulating the run-time call stack. We split out intra-class paths when the flow of control leaves the currently enclosing class or when the same statement is revisited in a loop. Once the list of statements has been partitioned into static intra-class paths, we count the number of times each path occurs. This *dynamic count* is our ground truth and the prediction target for our static analysis.

3.4 Static Path Description Model

We hypothesize that there is sufficient information embedded in the code of a path to estimate the relative frequency with which it will be taken at runtime. We characterize paths using a set of **features** that can be used to render this estimate. With respect to this feature set, each path can be viewed as a vector of numeric entries: the *feature values* for that path. Our path frequency estimation model takes as input a feature vector representing a path and outputs the predicted frequency of that path.

In choosing a set of features we have two primary concerns: predictive power and efficiency. For power, we base our feature set on our earlier intuition of the role that program state transformations play in software engineering practice. To scale to large programs, we choose features that can be efficiently computed in linear time.

The features we consider for this study are enumerated in Table 3.1; they are largely designed to capture the state-changing behavior that we hypothesize is related to path frequency. We formulate these features for Java, but note that they could easily be extended to other object-oriented languages. Many of our features consist of a static count of some source-level feature, such as the number of `if` statements. Others measure the percentage of some possible occurrences that appear along the path. For example, “field coverage” is the percentage of the total class fields that appear along the path, while “fields written coverage” is the percentage of the non-`final` fields of that class that are updated along the path. “Invoked method statements” refers to statements only from the method where the path originates. Our count features are discrete, non-negative and unbounded; our coverage features are continuous in the closed interval $[0, 1]$.

Given vectors of numeric features, we can use any number of machine learning algorithms to model and estimate path frequency. In our experiments, accurate estimates of path frequency arise from a complex interaction of features. We view the choice of features, the hypothesis behind them, and the final performance of a model based on them as more important than the particular machine learning technique used. In fact, we view it as an advantage that multiple learning techniques that make use of our features perform similarly. In our experiments, for example, Bayesian and Perceptron techniques performed almost equally well; this suggests that the features, and not some quirk of the learning algorithm, are responsible for accurate predictions. In the experiments in this chapter we use logistic regression because its accuracy was among the best we tried, and because the probability estimates it produced showed a high degree of numerical separation (i.e., the probability estimates showed few collisions). This property proved important in creating a ranked output for a large number of paths.

Count	Coverage	Feature
✓		<code>==</code>
✓		<code>new</code>
✓		<code>this</code>
✓		all variables
✓		assignments
✓		dereferences
✓	✓	fields
✓	✓	fields written
✓	✓	invoked method statements
✓		<code>goto</code> stmts
✓		<code>if</code> stmts
✓		local invocations
✓	✓	local variables
✓		non-local invocations
✓	✓	parameters
✓		<code>return</code> stmts
✓		statements
✓		<code>throw</code> stmts

Table 3.1: The set of features we consider. A “Count” is a raw static tally of feature occurrences. A “Coverage” is a percentage measuring the fraction of the possible activity that occurs along the path.

3.5 Model Evaluation

We performed two experiments to evaluate the predictive power of our model with respect to the SPECJVM98 benchmarks. The first experiment evaluates our model as a binary classifier, labeling paths as either “high frequency” or “low frequency.” The second experiment evaluates the ability of our model to correctly sort lists of paths by frequency.

3.5.1 Experimental Setup

We have implemented a prototype version of the algorithm described in Section 3.4 for the Java language. We use the Soot [128] toolkit for parsing and instrumenting. We use the Weka [87] toolkit for machine learning. We characterize our set of benchmarks in Table 3.2. Note that the JAVAC program is significantly larger and, in our view, more realistic than the other benchmarks; we see it as a more indicative test of our model in terms of program structure. However, all of the benchmarks have been specifically designed to be indicative of real programs in terms of runtime behavior.

Table 3.3 details the behavior of our static path enumeration algorithm (see Section 3.3.1) on our benchmarks. The JAVAC and JACK benchmarks had the largest number of static paths. Despite its relatively large size in lines of code, JESS actually had very few static intra-class paths because of its relatively sparse call graph. The JACK and DB benchmarks had the highest number of static paths per method because of

Name	Description	LOC	Class	Meth
check	check VM features	1627	14	107
compress	file compression	778	12	44
db	data management	779	3	34
jack	parser generator	7329 ^a	52	304
javac	java compiler	56645	174	1183
jess	expert system shell	8885	5	44
mtrt	ray tracer	3295	25	174
Total		79338	275	1620

^ano source code was given for jack; we report lines from a decompiled version. Our analysis runs on Java bytecode.

Table 3.2: The set of SPECJVM98 benchmark programs used. “Class” counts the number of classes, “Meth” counts the number of methods.

Name	Paths	Paths/Method	Runtime
check	1269	11.9	4.2s
compress	491	11.2	2.91s
db	807	23.7	2.8s
jack	8692	28.6	16.9s
javac	13136	11.1	21.4s
jess	147	3.3	3.12s
mtrt	1573	9.04	6.17s
Total or Avg	26131	12.6	59s

Table 3.3: Statistics from static intra-class path enumeration.

their more complicated object structures. In total we were able to enumerate over 26,000 static paths in under a minute on a 2GHz dual-core architecture; the process was largely disk bound to 150 MBytes/sec.

3.5.2 Classifier Training

We formulate the problem of hot-path identification as a classification task: given a path, does it belong to the set of “high frequency” or “low frequency” paths? We can phrase the problem alternatively as: with what probability is this a “high frequency” path? In our case, each **instance** is a **feature** vector of numerical values describing a specific static path, and the **classifier** used is based on logistic regression.

Such classifiers are supervised learning algorithms that involve two phases: training and evaluation. The training phase generates a classifier from a set of instances along with a labeled correct answer derived from the path counts obtained by dynamic path enumeration (Section 3.3.2). This answer is a binary judgment partitioning the paths into low frequency and high frequency. We consider two ways of distinguishing between these categories: an “intra-method” approach and an “inter-method” approach. The preferred scheme necessarily depends on the client application.

In an intra-method experiment we first normalize path counts for each method so that the sum of all path counts for that method becomes 1.0. We label paths as high frequency if they have a normalized intra-method frequency greater than 0.5 (i.e., they have a frequency greater than all other paths in the method combined). In an inter-method experiment, we consider a path as high frequency if it was taken more than 10 times. Our approach is largely insensitive to the threshold chosen; we achieve very similar results for thresholds between two and twenty.

After training, we apply the classifier to an instance it has not seen before, obtaining an estimate of the probability that it belongs in the high frequency class. This is our estimate for the runtime frequency of the path.

To help mitigate the danger of over-fitting (i.e., of constructing a model that fits only because it is very complex in comparison to the amount of data), we always evaluate on one benchmark at a time, using the other benchmarks as training data. We thus avoid training and testing on the same dataset.

3.5.3 F-score Analysis

First, we quantify the accuracy of our approach as an information retrieval task. In doing so we evaluate only discrete (nominal) classifier output. Two relevant success metrics in an experiment of this type are *recall* and *precision*. Here, recall is the percentage of high frequency paths that were correctly classified as high frequency by the model. Precision is the fraction of the paths classified as high frequency by the model that were actually high frequency according to our dynamic path count. When considered independently, each of these metrics can be made perfect trivially (e.g., a degenerate model that always returns high frequency has perfect recall). We thus weight them together using the *F*-score statistic, the harmonic mean of precision and recall [89]. This reflects the accuracy of the classifier with respect to the high frequency paths.

For each benchmark X , we first train on the dynamic paths from all of the other benchmarks. We then consider all of the high frequency paths in X , as determined by dynamic path count. We also consider an equal number of low frequency paths from X , chosen at random. We then use our model to classify each of the selected paths. Using an equal number of high- and low-frequency paths during evaluation allows us to compare accuracy in a uniform way across benchmarks. The *F*-scores for each benchmark, as well as the overall average, are shown in Figure 3.3.

In our case, an *F*-score of 0.67 serves as a baseline, corresponding to a uniform classification (e.g., predicting all paths to be high frequency). Perfect classification yields an *F*-score of 1.0, which our model achieves in 3 cases. On average we obtain an *F*-score of 0.8 for intra-method hot paths and 0.9 for inter-method ones.

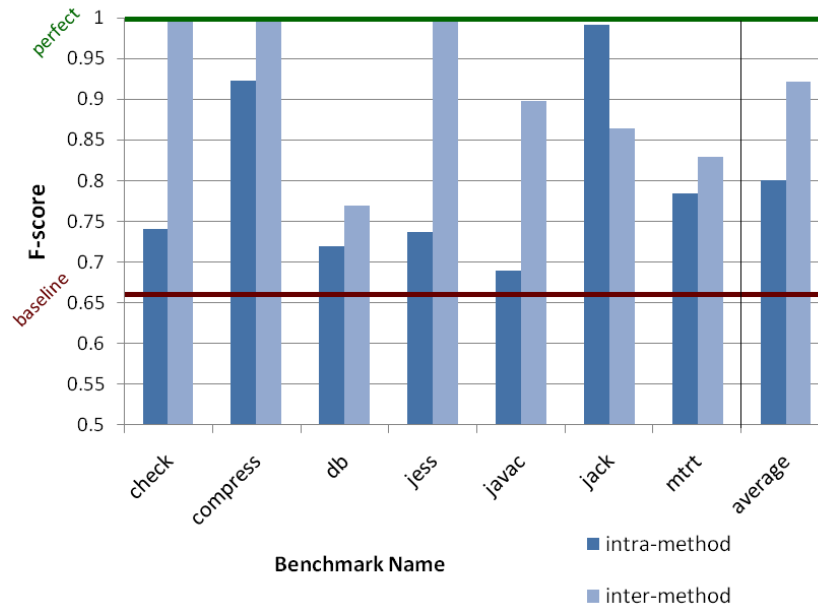


Figure 3.3: F -score of our performance at classifying high frequency paths. 0.67 is baseline; 1.0 is perfect.

Our model is thus accurate at classifying hot paths. One example of an application that would benefit from such a classifier is static specification mining. A specification miner might treat frequent and infrequent paths differently, in the same way that previous miners have treated error paths and normal paths differently for greater accuracy [56]. However, for other analysis tasks it is important to be able to discriminately rank or sort paths based on predicted frequency. Our next experiment extends our model beyond binary classification to do so.

3.5.4 Weighted Rank Analysis

In this experiment we evaluate our model’s ability to rank, sort and compare relative execution frequencies among paths. Rather than a binary judgment of “high frequency” or “low frequency”, we use the classifier probability estimates for high frequency to induce an ordering on paths that estimates the real ordering given by the dynamic path counts.

We use the same method for data set construction, training and testing as before, but evaluation requires a new metric. Intuitively, if the dynamic path counts indicate that four paths occur in frequency order $A > B > C > D$, then the prediction $A > B > D > C$ is more accurate than the prediction $D > C > B > A$. Kendall’s tau [129] is a distance metric between ranked orderings that formalizes this intuition; we use it as our evaluation criterion.

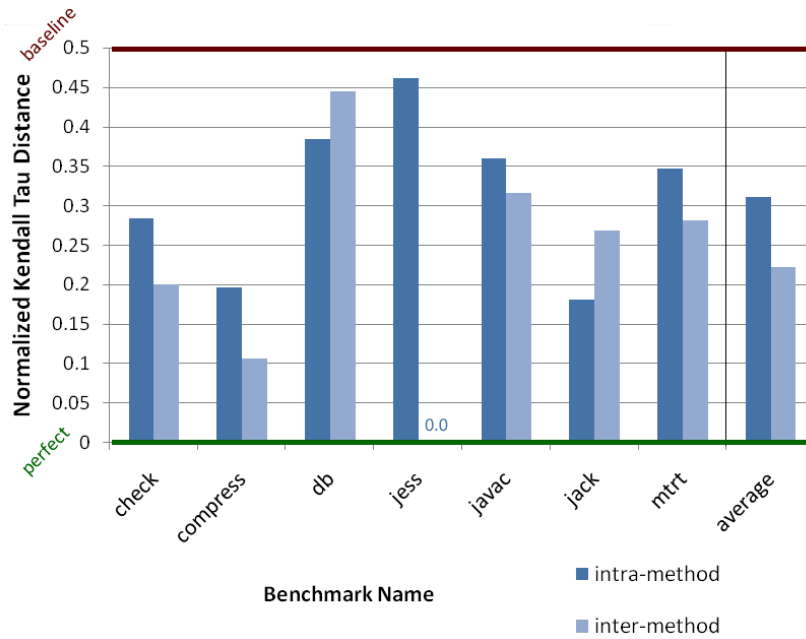


Figure 3.4: Kendall’s tau performance of our model when used to rank order paths based on predicted dynamic frequency. 0.0 is perfect; 0.25 is a strong correlation; 0.5 is random chance.

Kendall’s tau is equivalent to the number of bubble sort operations or swaps needed to put one list in the same order as a second normalized to the size of the lists. Kendall’s tau is a distance metric for ordered lists, and smaller numbers indicate a stronger correlation. We use a normalized Kendall’s tau to facilitate comparison between lists of different sizes. Lists in reverse order (i.e., exactly wrong) would score 1.0, a random permutation would score 0.5 on average, and perfect agreement (i.e., the same order) yields 0.0.

Figure 3.4 shows the Kendall’s tau values for each benchmark individually as well as the overall average. For the particular case of inter-method hot paths on the `jess` benchmark we obtain a perfect score of 0.0 (i.e., exactly predicting the real relative order). On average, our model yields a tau value of 0.30 for intra-method and 0.23 for inter-method paths. We view numbers in this range as indicative of a strong correlation. Note that small perturbations to the ordering, especially among the low frequency paths, are not likely to represent a significant concern for most clients. Static branch prediction [110], program specialization [126], and error report ranking [125] are examples of client analyses that could make use of ranked paths. In the next section we show how selecting “top paths” from this ranking is sufficient to quickly characterize a large percentage of run time program behavior.

3.6 Path Frequency and Running Time

We have shown that our static model can accurately predict runtime path frequency. In this section, we use our model for hot paths to characterize program behavior in terms of actual running time. In essence, we evaluate our model as a “temporal coverage metric.”

To acquire timing data we re-instrument each benchmark to record entry and exit time stamps for each method to nanosecond resolution. This adds an average of 19.1% overhead to benchmark run time, as compared to over 1000% for our full dynamic path enumeration. We combine timing data with path enumeration and calculate the total time spent executing each path.

We again use our model to sort static program paths by predicted frequency. We then select the X most highly ranked paths and calculate the percentage of the total runtime spent along the selected paths. We experiment with selecting X paths from each method in Figure 3.5, and X paths from the entire program in Figure 3.6.

Our results indicate a strong correlation between our static model for path frequency and run time program behavior. In the intra-method case, selecting the single “hottest path” from each method is sufficient to capture nearly 95% of program run time. Recall that, on average, these benchmarks have over 12 paths per method. Furthermore, in the inter-method case selecting 5% of program paths is sufficient for over 50% temporal coverage on average. This analysis indicates that our model is capable of characterizing real program run time behavior, with accuracy that is likely to be sufficient for improving or enabling many types of program analyses. For example, a feedback-directed compiler optimization that is too expensive to be applied to all paths could be applied to the single hottest path predicted by our model and have a high probability of improving common-case performance.

3.7 Path Frequency for Branch Prediction

In this section we demonstrate the robustness of our approach by coercing our static path frequency prediction model into a static branch predictor. A static branch predictor is one that formulates all predictions prior to running the program. Intuitively, to make a branch prediction, we ask our model to rank the paths corresponding to the branch targets and we choose the one with the greatest frequency estimate.

To achieve this comparison, we start by constructing the control flow graph for each method. At the method entry node, we enumerate the list of paths for the method and sort that list according to our intra-method frequency estimate. We then traverse the control flow graph (CFG) top-down, at every node maintaining a set of static paths that could pass through that node.

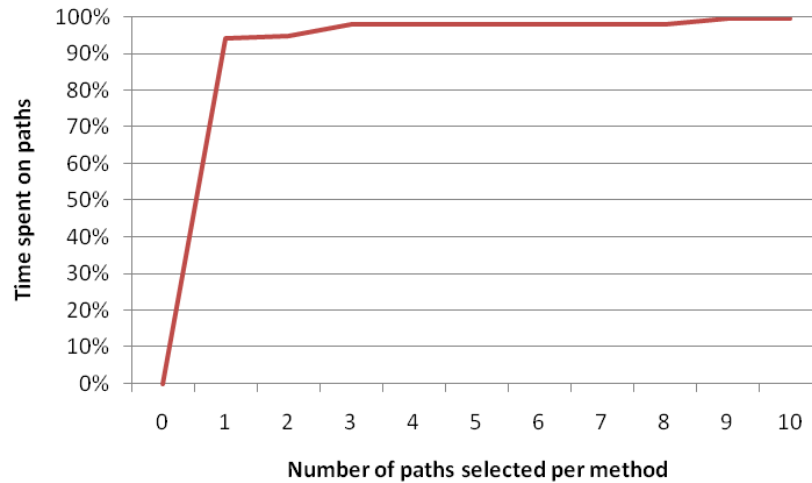


Figure 3.5: Percentage of total run time covered by examining the top paths in each method.

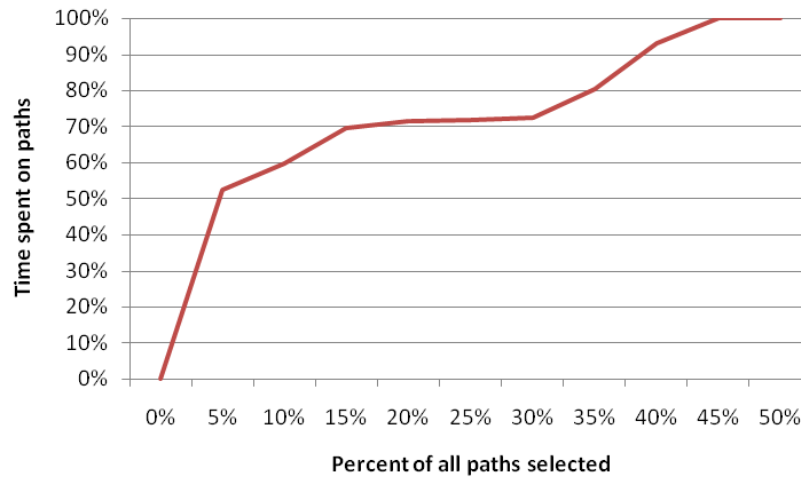


Figure 3.6: Percentage of the total run time covered by examining the top paths of the entire program.

Consider, for example, the code fragment below:

```

1  a = b;
2  c = d;
3  if (a < c) {      /* choice point */
4      e = f;
5  } else {
6      g = h;
7  }

```

If execution is currently at the node corresponding to line 2, the future execution might proceed along the path 1-2-3-4 or along the path 1-2-3-6. At each branching node we perform two actions. First, we partition the path set entering the node into two sets corresponding to the paths that conform to each side of the branch. Second, we record the prediction for that branch to be the side with the highest frequency path

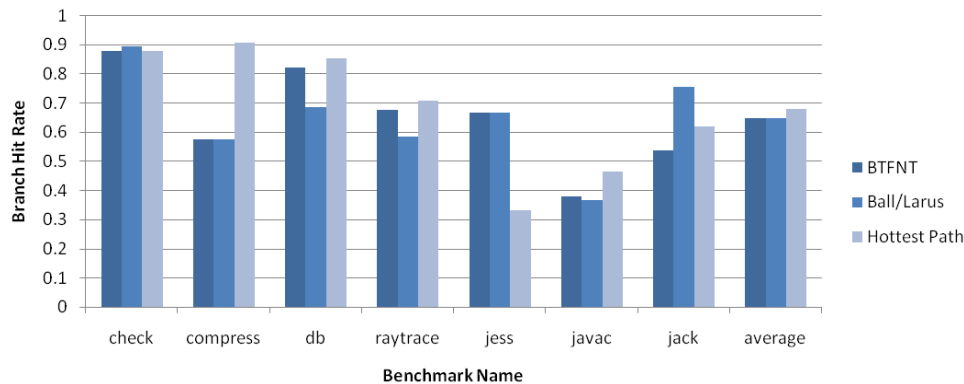


Figure 3.7: Static branch prediction hit rate. BTFNT is a baseline heuristic; Ball/Larus represents the state-of-the-art; Hottest Path corresponds to using our static path frequency prediction to predict branches.

available. In the code fragment above, we will query our model about the relative ordering of 1-2-3-4 and 1-2-3-6; if 1-2-3-6 were ranked higher we would record the prediction “not taken” for the branch at line 3. Finally, because our path model is acyclic, at branches with back edges we follow the standard “backward taken / forward not taken” (BTFNT) heuristic, always predicting the back edge will be followed. This scheme is based on the observation that back edges often indicate loops, which are typically taken more than once.

We compare our branch prediction strategy to a baseline BTFNT scheme and to an approach proposed by Ball and Larus [110] based on a set of nine heuristics such as “if a branch checks an integer for less than zero... predict the branch on false condition.” We compare the predictions from these three strategies against the ground truth of running each benchmark. We measure success in terms of the branch “hit rate” (i.e., the fraction of time the static prediction was correct). Figure 3.7 reports the hit rate for each benchmark individually as well as the overall average.

Static branch prediction is a difficult task. Even with an optimal strategy a 90% hit rate is typically maximal [110], and no one strategy is perfect for all cases. For example, Ball and Larus perform well on JACK because their heuristics can be frequently employed. Our technique performs quite well on COMPRESS because of its relatively large number of branches with two forward targets. On average, however, a static branch predictor based on our static model of path frequencies outperforms common heuristics and previously-published algorithms by about 3%. This result serves to indicate that our frequency model can be robust to multiple use cases.

3.8 Model Implications

We have shown that our model is useful for estimating the runtime frequency of program paths based solely on a set of static features. In this section we present a brief qualitative analysis of our model and discuss

some of its possible implications. We conduct a singleton feature power analysis to evaluate individual feature importance as well as a principle component analysis (PCA) to access the degree of feature overlap.

3.8.1 Feature Power Analysis

In Figure 3.8 we rank the relative predictive power of each feature and indicate the direction of correlation with runtime frequency. For example, the single most predictive feature in our set is the number of statements in a path. On average, more statements imply the path is *less* likely to be taken.

This data is gathered by re-running our F -score analysis on each benchmark as before, but training on only a single feature and measuring the predictive power of the resulting model. We then normalize the data between 0 and 1. For correlation direction, we compare the average value of a feature in the high and low frequency classes. We prefer a singleton analysis to a “leave-one-out” analysis (where feature power is measured by the amount of performance degradation resulting from omitting the feature from the feature set) due to the high degree of feature overlap indicated by a PCA, where we find that 92% of the variance can be explained by 5 principle components.

The singleton feature power analysis, while not a perfect indicator of the importance of each feature, does offer some qualitative insight into our model. For example, the relatively high predictive power of statement counts, particularly assignment statements, would seem to support our hypothesis that paths with large changes to program state are uncommon at runtime. We observe much weaker predictive power when training only on class field metrics, suggesting that updates to local variables are also important for uncommon paths. On the other hand, local variable coverage correlates with frequent paths: since variable updates correlate with uncommon paths, our model bears out our intuition that common paths read many local variables but do not update them. Features characterizing path structure, such as “if” and “return” count, show modest predictive power compared to features about state updates.

3.8.2 Threats To Validity

One threat to the validity of these experiments is overfitting — the potential to learn a model that is very complicated with respect to the data and thus fails to generalize. We mitigate that threat by never testing and training on the same data; to test one program we train on all others. Our results may fail to generalize beyond the benchmarks that we presented; we chose the SPEC benchmarks as indicative examples. Finally, our results may not be useful if our notion of a static intra-class path does not capture enough essential program behavior. Since our output is formulated on these static paths, a client analysis must make queries in terms of them. Our success at building a branch predictor on top of our path frequency analysis helps to

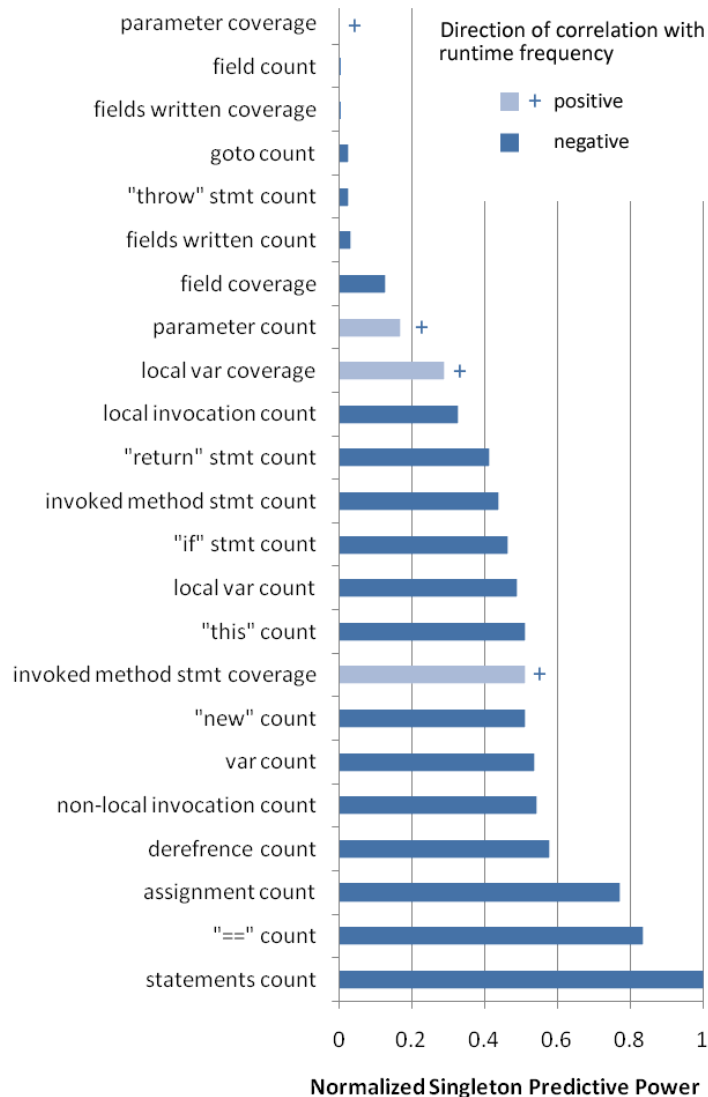


Figure 3.8: Relative predictive power of features as established with a singleton analysis.

mitigate that danger; while some program behavior is not captured by our notion of static paths, in general they form a natural interface.

3.9 Conclusion

We have presented a descriptive statistical model of path frequency based on features that can be readily obtained from program source code. Our static model eliminates some of the need for profiling by predicting runtime behavior without requiring indicative workloads or testcases. Our model is over 90% accurate with respect to our benchmarks, and is sufficient to select the 5% of paths that account for over half of the total runtime of a program. We also demonstrate our technique's robustness by measuring its performance as a

static branch predictor, finding it to be more accurate on average than previous approaches. Finally, the qualitative analysis of our model supports our original hypothesis: that the number of modifications to program state described by a path is indicative of its runtime frequency. We believe that there are many program analyses and transformations that could directly benefit from a static model of dynamic execution frequencies, and this work represents a first step in that direction.

Chapter 4

Documenting Exceptions

“How glorious it is — and also how painful — to be an exception.”

– *Alfred de Musset*

4.1 Introduction

PREVIOUS chapters have described automated models for program readability and runtime behavior. Equipped with these new insights on code understandability, we now propose to improve it with algorithms capable of synthesizing **documentation**. We begin by documenting a key aspect of modern programs that are often difficult to reason about correctly: exceptions.

Modern **exception** handling allows an error or unusual condition detected in one part of a program to be handled elsewhere depending on the context. A method may not have enough information to handle “exceptional” conditions. In such cases the method “raises” or “throws” an exception to a parent method farther up the call stack where sufficient context may exist to properly handle the event. Most language-level exception schemes are based on this *replacement model* [37, 130]. The resulting non-sequential control flow is simultaneously convenient and problematic [38, 39]. Uncaught exceptions and poor support for exception handling are reported as major obstacles for large-scale and mission-critical systems (e.g., [40, 41, 42, 43]). Some have argued that the best defense against this class of problem is the complete and correct documentation of exceptions [44]. The use of so-called “checked” exceptions, which force developers to declare the presence of uncaught exceptions, is a partial solution. However, in practice, many developers have found this to be burdensome requirement. Often exceptions are caught trivially (i.e., no action is taken to resolve the underlying error [131]) or the mechanism is purposely circumvented [132, 133].

In this chapter we investigate current practice in the documentation of exceptions. In general, we find this documentation to be largely incomplete, especially in circumstances where it is most difficult for a human to generate. We also find it to be often out-of-date or inaccurate and frequently the result of copy-and-pasting documentation from elsewhere in the program. Despite this, we show that the presence of documentation correlates significantly with code quality. In particular, documenting exceptions more completely can measurably reduce defect rates in client software.

Reasoning about programs that use exceptions is difficult for humans as well as for automatic tools and analyses (e.g., [134, 135, 136, 137, 138]). We propose to relieve part of that burden. We present an algorithm for inferring conditions that may cause a given method to raise an exception. Our automatic approach is based on symbolic execution and inter-procedural dataflow analysis. It alerts developers to the presence of “leaked” exceptions they may not be aware of, as well as to the causes of those exceptions. It also makes plain the concrete types of exceptions that have been masked by subsumption and subtyping; designing an exception handler often requires precise exception type information [44, 132]. Finally, but importantly, the tool can be used to automatically generate documentation for the benefit of the developers, maintainers, and users of a system.

Software maintenance, traditionally defined as any modification made to a system after its delivery, is a dominant activity in software engineering. Some reports place maintenance at 90% of the total cost of a typical software project [9, 10]. One of the main difficulties in software maintenance is a lack of up-to-date documentation [139]. As a result, some studies indicate that 40% to 60% of maintenance activity is spent simply studying existing software (e.g., [2] p. 475, [9] p. 35). Improving software documentation is thus of paramount importance, and in many cases our proposed algorithm does just that. In addition, our algorithm is fully automatic and quite efficient, making it reasonable to run it nightly and thus prevent drift between the maintained program and the documentation.

Many tools exist that allow for the automatic extraction of API-level documentation from source-code annotations. JAVADOC, a popular tool for Java, has been studied in the past (e.g., [140]) and is in common use among both commercial and open source Java projects. JAVADOC allows programmers to document the conditions under which methods throw exceptions, but in many existing projects this documentation is incomplete or inconsistent [141, 142, 143]. Users report that they prefer documentation that, among other properties, “provides explanations at an appropriate level of detail”, is “complete” and is “correct” [45]. The documentation produced by our prototype tool integrates directly with JAVADOC and we evaluate it in terms of its completeness, correctness and inclusion of potentially-inappropriate information.

The main contributions of this chapter are:

- A study of the documentation of exceptions in several existing open source projects. We correlate the presence and quality of documentation with multiple software complexity and quality metrics, and we account for and analyze the presence of documentation created by copy-and-paste.
- An automatic algorithm for determining conditions sufficient to cause an exception to be thrown. Those conditions are used to generate human-readable documentation for explicitly-thrown and implicitly-propagated exceptions.
- Experimental evidence that our tool generates documentation that is at least as accurate as what was generated by humans in 85% of cases.
- A series of experiments comparing the documentation rate of exceptions with proxy metrics for analysis difficulty. In particular, we investigate method length, cyclomatic complexity, and readability. In general, we observe fewer exceptions documented in methods that are more difficult for humans to analyze. We claim that our automated tool can reduce the documentation bottleneck caused by human analysis difficulties.
- An investigation and characterization of copy-and-paste documentation. We find that, in total, about 50% of the documentation instances in our benchmarks are exact duplicates of documentation found elsewhere. In our study, we confirm that copy-and-paste documentation is frequently worse than unique documentation. We claim that a tool for automatic documentation can be useful not only where no human documentation is present, but also in cases where human documentation is of low quality.
- An experiment correlating exception documentation frequency, the fraction of exceptions that could be documented and actually are, with a more natural notion of software quality and defect density: explicit human mentions of bug repair. Using version control repository information we coarsely separate changes made to address bugs from other changes. We find that the rate at which exceptions are documented exhibits a significant negative correlation with the appearance of bugs, implying that documentation of this type may be useful for preventing software defects.

The structure of this chapter is as follows. In Section 4.2 we present a motivating example related to exception documentation inference. We present our algorithm for tracking exceptions in Section 4.3. In Section 4.4 we empirically study existing programs and motivate the need for automatic exception documentation. We present our algorithm for documenting exceptions in Section 4.5. Section 4.6 presents experimental results about the efficacy of our algorithm on off-the-shelf programs. Section 4.7 concludes.

4.2 Motivating Example

In this section we illustrate some of the challenges in documenting exceptions with a simple example drawn from `FreeCol`, an open-source game. The class in question is called `Unit` and is intended to be sub-classed many times.

```

1  /**
2   * Moves this unit to america.
3   *
4   * @throws IllegalStateException If the move is illegal.
5   */
6  public void moveToAmerica() {
7      if (!(getLocation() instanceof Europe)) {
8          throw new IllegalStateException("A unit can only be moved"
9              + " to america from europe.");
10     }
11     setState(TO_AMERICA);
12     // Clear the alreadyOnHighSea flag:
13     alreadyOnHighSea = false;
14 }

```

Our goal is to determine what exceptions `moveToAmerica()` can raise, and what conditions are sufficient to cause them to be raised. On the surface, this seems fairly simple: if the result of a call to `getLocation()` returns an object that is not of type `Europe` an `IllegalStateException` will be thrown on line 9.

There is more to consider, however. Any method invocation, including `getLocation()` (line 8) and `setState()` (line 13), has the potential to throw an exception as well. We thus need to inspect those methods to see which exceptions they may throw. Additionally, because this class is likely to be extended and method invocation is handled by dynamic dispatch, the call to either method may resolve to an implementation for any subclass of `Unit`. Any of those methods may contain method invocations of their own. Finding all relevant exception sources can thus require significant tracing, and the more complex the code gets, the harder it becomes to track what must be true to reach those `throw` statements. Java addresses part of this problem by requiring “throws clause” annotations for checked exceptions and limiting what checked exceptions can be declared for subtypes. However, for unchecked exceptions in Java (or all exceptions in C#, etc.) the problem remains.

Software evolution and maintenance present additional major concerns. If a `throw` statement is added to any method reachable from `moveToAmerica`, the documentation of it, and potentially many other methods, including any method that calls `moveToAmerica`, would have to be amended. Notice also that the human provided JAVADOC documentation for this method is “If the move is illegal”. This hides what constitutes an illegal move, which might be desirable if we expect that the implementation might change later. However, an automated tool could provide specific and useful documentation that would be easy to keep synchronized

with an evolving code base. The algorithm we present in this chapter generates “`IllegalStateException` thrown when `getLocation()` is not a Europe.”

We also note that the documentation string “If the move is illegal” is used in two other locations in the `FreeCol` source tree. Such duplicated documentation leads us to surmise that “copy-and-paste” documentation may be of lower precision and utility. Copied documentation may not have been specialized to its new context and may thus not provide sufficient detail to be useful.

4.3 Exception Flow Analysis

We now present an algorithm that determines the runtime circumstances sufficient to cause a method to throw an exception. Our algorithm locates exception-throwing instructions and tracks the flow of exceptions through the program.

The algorithm is essentially a refinement of JEX [144, 132]. For each method we generate a set of possible exception types that could be thrown by (and thus escape) that method. Our algorithm contains two improvements over previous work: a pre-processing of the call graph for increased speed, and a more precise treatment of throw statements to ensure soundness. This algorithm takes as input a program, its call graph, and the results of a receiver-class analysis on dynamic dispatches, and produces as output a mapping from methods to information about thrown exceptions.

Figure 4.1 describes this algorithm formally. For each method, we call each separate exception that can escape that method and be propagated to its callers an *exception instance*. Each exception instance is an opportunity for documentation. An exception instance can be thrown directly from the method in question or thrown (but not caught) in a called method. For each exception instance, we trace the statement that raises it to each of the methods that can propagate it. We produce a set of exception instances for each method in the input program.

We require the program call graph C and a mapping R from method invocations to sets of concrete methods that could be invoked there at runtime. Our approach takes the form of a fixpoint worklist algorithm that considers and processes methods in turn. Since exceptions can be propagated through method invocations, we first process leaf methods that make no such invocations. We process a method only when we have precise information about all of methods it may invoke. This process terminates because the updates (line 14) are monotonic (i.e., we can learn additional exceptions that a method may raise, but we never subtract information) and the underlying lattice is of finite height (i.e., at worst a method can raise all possible exceptions mentioned in the program).

Input: Receiver-class information R .
Input: Program call graph C (built using R).
Input: Maximum desired propagation $depth$.
Output: Mapping $M : \text{methods} \rightarrow \text{exception info}$.

```

1: for all all methods  $m \in C$  do
2:    $M(m) \leftarrow \emptyset$ 
3: end for
4:  $Worklist \leftarrow \text{methods in } C$ 
5: Topological sort  $Worklist$ 
6: while  $Worklist$  is not empty do
7:    $c \leftarrow \text{remove next cycle of methods from } Worklist$ 
8:   for all  $m \in c$  do
9:      $Explicit \leftarrow \{(e, l, 0) \mid m \text{ has } \text{throw } e \text{ at } l\}$ 
10:     $Propa \leftarrow \{(e, l, d + 1) \mid m \text{ has a method call at } l$ 
       $\wedge m' \in R(l) \wedge (e, l', d) \in M(m')\}$ 
11:    for all  $(e, l, d) \in Explicit \cup Propa$  do
12:      if  $l$  not enclosed by  $\text{catch } e'$  with  $e \leq e'$  then
13:        if  $d \leq depth$  then
14:           $M(m) \leftarrow M(m) \cup \{(e, l, d)\}$ 
15:        end if
16:      end if
17:    end for
18:    if  $M$  changed then
19:      Add  $m$  and methods calling  $m$  to  $Worklist$ 
20:    end if
21:  end for
22: end while

```

Figure 4.1: Algorithm for determining method exception information. If $M(m)$ is (e, l, d) then m can propagate (leak) exception e from location l , with e having already propagated through d other methods.

Processing a method consists of determining the set of exception instances that method can raise. We consider both explicit **throw** statements (line 9) and also method invocations (line 10) as possible sources of exceptions. We associate a new exception instance with the given method if a thrown exception is not caught by an enclosing **catch** clause (line 12) and thus may propagate to the caller. With respect to sources of exceptions that we consider, this analysis is conservative and may report exceptions that could never be thrown at run-time (e.g., `if (1 == 2) throw Exception();`).

We do *not* consider non-throw statements as possible sources of exceptions (e.g., division raising a divide-by-zero exception). While there may be some utility in a system that attempts to exhaustively track *all* exceptions, those exceptions that are implicitly-thrown are generally indicative of programming errors rather than design choices [131], and their inclusion in an API-level documentation might often be considered inappropriate [141]. Furthermore, estimating all implicitly-thrown exceptions is likely to result in many false positives (as noted by [132], strictly speaking any statement can throw any exception).

This algorithm is a refinement of JEX [144, 132]. The primary differences are that we topologically sort the call graph and also that we maintain completeness by considering all **throw** statements, instead of just those

Name	Version	Domain	kLOC	methods	throws	documented
Azureus	2.5	Internet File Sharing	470	14678	655	35.42%
DrJava	20070828	Development	131	2215	121	19.01%
FindBugs	1.2.1	Program Analysis	142	2965	245	7.76%
FreeCol	0.7.2	Game	103	3606	330	75.15%
hsqldb	1.8.0	Database	154	2383	315	26.03%
jEdit	4.2	Text Editor	138	1110	53	15.09%
jFreeChart	1.0.6	Data Presentation	181	41	7	42.86%
Risk	1.0.9	Game	34	366	15	40.00%
tvBrowser	2.5.3	TV guide	155	3306	21	57.14%
Weka	3.5.6	Machine Learning	436	127	9	88.89%
Total or Mean			1944	30797	1771	40.73%

Table 4.1: The set of Benchmark programs used in this study. The “methods” column gives the total number of methods reachable from `main()`. The “throws” column gives the number of `throw` statements contained in those methods. The “documented” column gives the percentage of those exceptions that are documented with a non-empty JAVADOC comment.

where the operand is a new object allocation. We also track additional information, such as the exception instance depth. We define *depth* to be the minimum number of methods an exception must propagate through before it reaches the method in question. For example, if an exception is explicitly thrown in the method body, its depth is 0. If `foo` calls `bar` and `bar` raises an exception, that exception has depth 1 in `foo`. In section Section 4.4 we will relate depth to the likelihood of human-written documentation.

Practically speaking, the primary implementation decision that may affect the precision of this algorithm (as well as the algorithm we present in Section 4.5) is choice of a call graph. The problem of constructing an accurate and precise call graph for dynamic languages typically involves receiver class analysis or alias analysis. Many off-the-shelf solutions exist. In fact, this problem has become one of the most heavily-researched topics in program analysis [145]. For the experiments presented in this chapter we employed SPARK [146] to produce call graphs. SPARK is reasonably fast (terminating in less than one hour for all of our benchmarks) and is integrated into SOOT [128], the Java bytecode analysis framework we used to parse input programs.

4.4 Existing Exception Documentation

In this section we explore the state-of-the-art in exception documentation. We selected ten popular open source Java programs totaling about 2 million lines of code and enumerated in Table 4.1. We present four studies of the quality and quantity of existing exception documentation.

Our first analysis focuses on documentation completeness (Section 4.4.1). We show that documentation is rarely present in a complete and consistent manner. Second, we show that even when documentation is present, it was often created by cut-and-paste and not specialized to the current context (Section 4.4.2).

Third, we show that there is a negative correlation between the presence of documentation and complexity of the surrounding method (Section 4.4.3). Finally, we show that despite the flawed nature of documentation in practice, it may be useful in preventing software defects: the presence of documentation is negatively correlated with software defects (Section 4.4.4).

4.4.1 Complete and Consistent Documentation

We begin by investigating how much exception documentation is actually present. For a given program P and a given exception type T , we define the *completeness* of the documentation for T to be:

$$\frac{\text{number of methods that throw } T \text{ and document it}}{\text{number of methods that throw } T}$$

We claim that complete documentation of exceptions is important to software developers because incompleteness can lead to problems with security, reliability, and encapsulation, as well as to difficulties in software maintenance.

In this subsection we will provide evidence to demonstrate that the conditions under which real world programs can raise exceptions are often not completely documented. We use the analysis from Figure 4.1 to determine all possible methods that can throw a given exception. Note that we are not yet considering documentation quality, only documentation presence. We return to the issue of quality later.

We note that not every exception type must be completely documented in every project. For a given program P and a given exception type T , we say that the documentation is *consistent* if the completeness is either one or zero. Intuitively, we expect consistent documentation to either include every instance where a particular exception type can be raised, or no instance. That is, we hypothesize that if `SecurityException` is deemed worth documenting some of the time, it should be worth documenting all of the time. This is similar to the assumption used by Engler et al. [18] to find bugs from inconsistencies in systems code. This notion of consistency allows for the possibility that the developers may consciously decide not to document certain kinds of exceptions, but views partial documentation of an exception type as a mistake.

Experimentally, we found many examples of inconsistent documentation in our benchmarks. Figure 4.2 presents some of the most frequently documented, yet sometimes neglected, exception types. For each benchmark chosen, the two most commonly documented exception types are listed. For each exception type we list the completeness of its documentation: the static count of methods documenting that exception as a percentage of methods that can throw that exception. We do not consider “stub” annotations, which indicate an exception type without additional comment (e.g., just `@throws Exception`), to be actual documentations

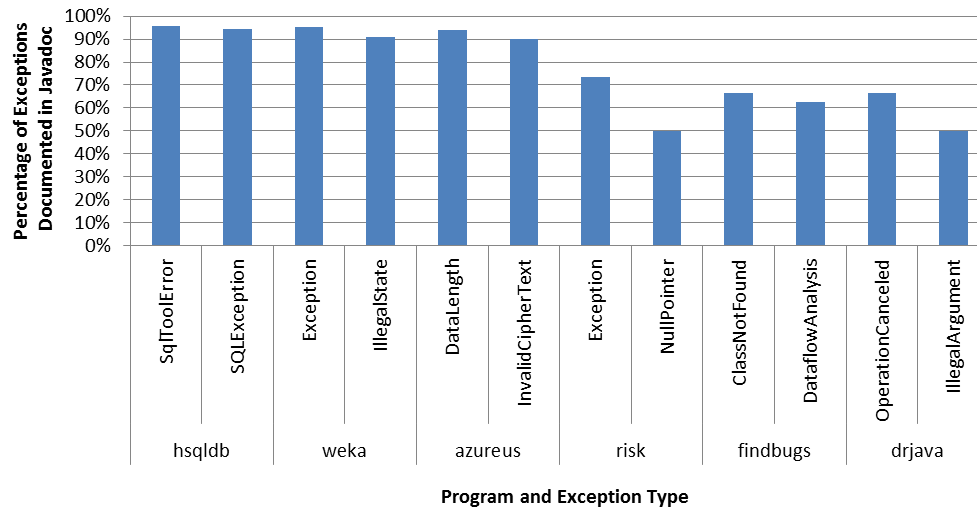


Figure 4.2: Completeness of exception documentation. Each column represents methods that document the given exception expressed as a percentage of methods that can throw the exception. For each benchmark, the two most frequently documented exceptions where the rate is below 100% are listed.

since they do not convey any information regarding the triggering of the exception, and may have been automatically inserted by a development environment.

We conjecture it is unlikely that documentation for a particular exception type is needed or appropriate in over 90% of methods that raise it, but somehow not needed or desired in the other 10%. It is interesting to note that some commonly-documented exceptions are associated with standard library exceptions.

4.4.2 Copy-and-Paste Documentation

Many instances of documentation are not unique: by convention, either intentional or otherwise, many documentation strings are repeated within a project. Figure 4.3 shows the total documentation completeness, broken down into unique and copy-and-paste documentation. In our observations, as much as 88%, and 55% on average, of exception documentation instances in our benchmark programs consist of a textual string that has been used more than once in the program to document an exception.

In Section 4.6 we measure the relative quality of duplicated documentation, but for now we simply note that a significant portion of exceptions have a copy-and-paste or “cookie cutter” nature. Some verbatim examples of duplicated documentation follow:

From AZUREUS

```
10x "inappropriate."
3x "on error"
```

From HSQLDB

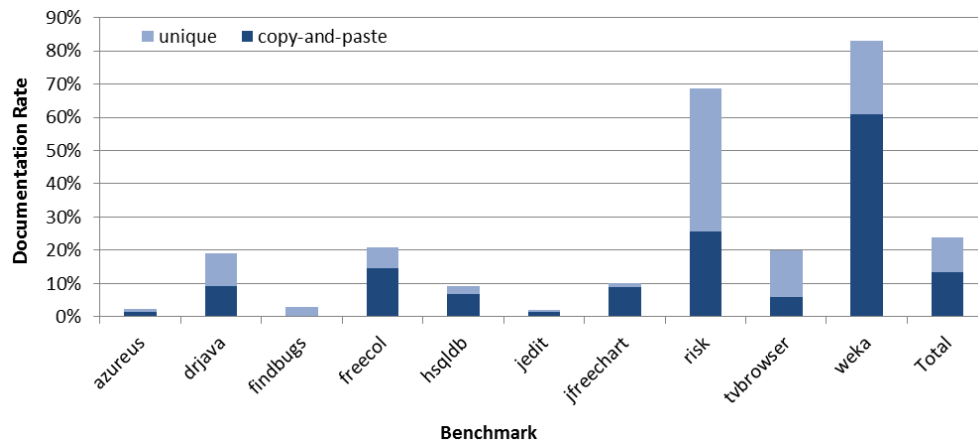


Figure 4.3: Overall documentation completeness (from JAVADOC `throws` clauses, totaled over all exception types in each program) in benchmark programs. The fraction of documentation that is duplicated somewhere else in the benchmark is labeled as “copy-and-paste.”

3x "Description of the Exception"

From JEDIT

12x "if an I/O error occurs"

From JFREECHART

23x "if there is a problem."

11x "for errors"

6x "should not happen."

From TVBROWSER

15x "Thrown if something goes wrong."

3x "possible Error"

This implies that such documentation instances may be not have been accurately specialized to the context of the method they are being used to describe, and may thus be unhelpful or misleading. Similarly, copy-and-paste source code has been shown to be “less trustworthy” than other code on average [57].

4.4.3 Difficulty in Documenting Exceptions

We also hypothesize that some exception instances may not be documented because of the difficulty of, or lack of programmer interest in (e.g., [9, p.45] and [147]), manually modeling exception propagation. Figure 4.4 indicates that the propagation depth of an exception, in practice, has a strong inverse correlation with the probability that it will be documented. In particular, it is quite rare for documentation to appear for any exception not explicitly raised in the current method (i.e., for any exception instance at depth > 0). For propagation depths larger than three, documentation becomes almost non-existent.

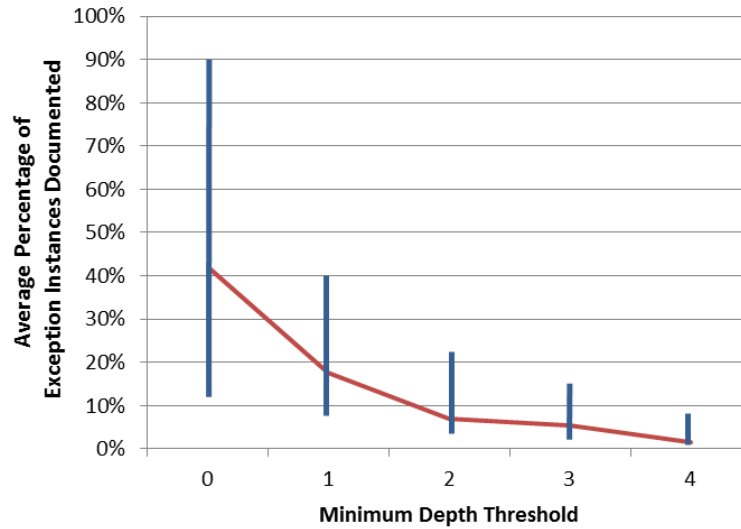


Figure 4.4: Exception documentation completeness as a function of exception propagation depth. The line represents the average over all exception instances within all of our benchmarks. The vertical bars indicate the maximum and minimum observed value. All of our benchmarks are included. An exception instance explicitly thrown within the method in question has depth zero; higher depths indicate the number of method invocations through which the exception must be propagated.

As additional analogues for difficulty, we also consider the complexity and readability of methods that might document an exception. For each method in our benchmarks we calculate a *documentation rate* — the number of thrown exceptions for that method that are documented as a fraction of those that might be documented. For the purpose of software engineering, we desire that complex or unreadable methods be more fully documented. In practice, however, we hypothesize that methods that are more difficult to understand will be less well documented. That is, documentation will be missing from the methods where it is needed most.

To measure readability, we use the metric described in Chapter 2. To measure complexity, we employ the well-known “cyclomatic complexity” metric originally proposed by McCabe [90]. Finally, as a baseline, we measure the length of each method as the number of statements it contains. We then calculate the Pearson Product Moment Correlation Coefficient between the rate at which exceptions are not documented (i.e., 1 - documentation rate) and each of these three proxies for understandability.

The results of this study are plotted in Figure 4.5. We observe a significant correlation between the difficulty of the documentation task and the lack of success at it, in terms of documentation rate. For Pearson Product Moment, 0.0 indicates no correlation and 1.0 indicates perfect correlation. On average we found our metrics for comprehension difficulty to correlate at a level near 0.2. These results support the intuition that difficulty in interpreting code has a significant negative influence on whether it will be documented.

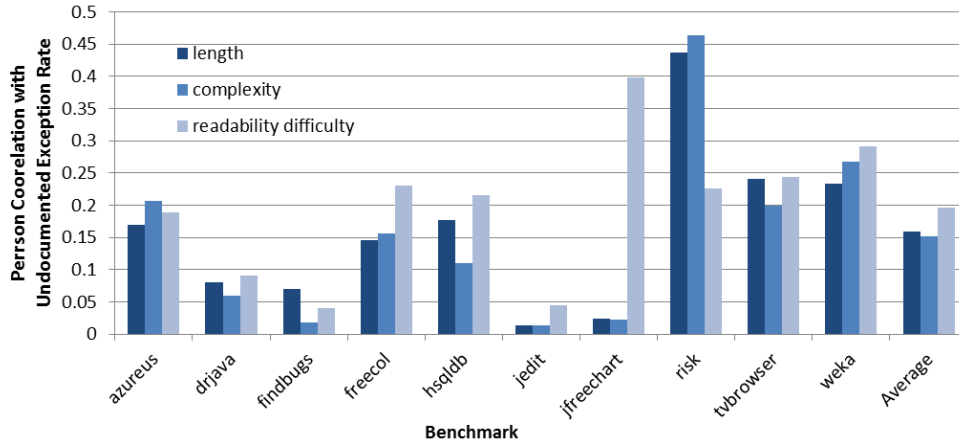


Figure 4.5: Pearson Product Moment Correlation between the rate at which exceptions remain undocumented (1 - documentation rate) and three proxies for the difficulty of comprehending a method. In all cases the computation is conducted on per-method basis. Length refers to the number of simple statements in the method. Complexity refers to Cyclomatic Complexity as per [90]. Readability difficulty is from Chapter 2.

4.4.4 Utility of Documentation

We have shown that documentation of exceptions is often incomplete, and that this incompleteness can be at least partially explained by the difficulty involved in modeling and understanding source code. Despite its incompleteness, the presence of documentation has some utility related to code quality. The documentation for an interface is often all that a programmer has available when writing client code that uses that interface. Intuitively, we claim that incomplete documentation makes it more likely that client code will contain bugs. Formally, we hypothesize that there is a significant correlation between whether a (client) method contains a defect and the rate at which the (interface) methods it calls document their thrown exceptions.

To measure a correlation between documentation rate and defects, we must be able to detect the presence of defects. Rather than using bug-finding tools to report potential defects, we restrict attention to defects reported by developers. Seven of our ten benchmarks feature publicly accessible version control systems, each with thousands of revisions. For the most recent version of each program, for each line of code, we determine the last revision to have changed that line (i.e., similar to the common `cvs blame` tool). We then inspect the log messages for the word “bug” to determine whether a change was made to address a defect or not. We partition all functions into two sets: those where at least one line was last changed to deal with a bug, and those where no lines were last changed to deal with a bug.

Next, for each method we compute the *exception documentation ratio* of called methods that it may invoke:

$$\frac{\# \text{ of documented exceptions raised by called methods}}{\# \text{ of exceptions raised by called methods}}$$

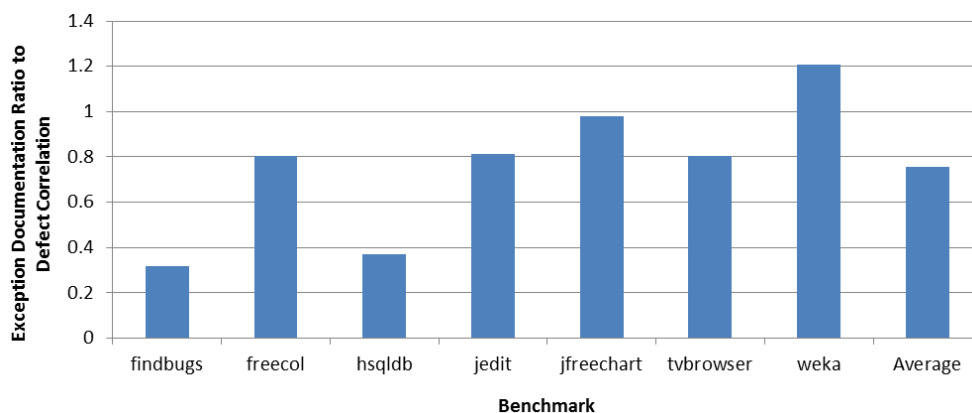


Figure 4.6: Characterizes the correlation between the rate at which exceptions are undocumented at invocation sites and whether the method underwent a defect repair in the past. A repair is indicated by a version control log message containing the word “bug.” 1.0, for example, indicates that for methods without bugs, exceptions in invoked methods are documented twice as often.

This ratio is an indicator of the completeness of documentation in invoked methods. If the ratio is low for a client method M , then the developer for M was “flying blind” with minimal indication of when subroutines might raise exceptions.

Figure 4.6 shows the average percent *difference* in the exception documentation ratio of methods that have been identified as having a bug to all other methods in the program. Thus, 1.0 indicates that for client methods without bugs, exceptions in invoked interface methods are documented 100% more frequently (i.e., twice as often). On average, we observe this statistic to be about 0.8, indicating a strong and consistent relationship between the rate at which exceptions go undocumented, and defect rates.

We thus conclude that there is significant reason to believe that exception documentation is useful for avoiding defects. While such documentation is often incomplete, inconsistent, or created by copy-and-paste, on average it makes a very measurable impact on software quality.

These investigations lead us to conclude that the lack of exception documentation in practice follows from the difficulty in developing it and not from a lack of need for it. These observations serve as evidence that the presence of JAVADOCS for exceptions is inconsistent and incomplete, and motivate the need for tool assistance for developing documentation.

4.5 Algorithm for Generating Documentation

We have shown in Section 4.4 that exception documentation can be useful, but we have also demonstrated that it tends to be incomplete in practice. These observations lead us to propose a tool that documents

exceptions in a fully-automatic and complete way. In this section we describe formally the algorithm that underlies our proposed tool. In Section 4.6 we will evaluate it on our set of benchmarks.

We now present an algorithm that generates documentation characterizing the runtime circumstances sufficient to cause a method to throw an exception. The algorithm starts with the exception information generated by our exception tracking algorithm (Section 4.3). It then symbolically executes control flow paths that lead to exceptions. This symbolic execution generates predicates describing feasible paths yielding a boolean formula over program variables. If the formula is satisfied at the time the method is invoked, then the exception can be raised. Via a final post-processing step, those formulae become the documentation describing the conditions under which the exception is thrown.

In Section 4.3 we derived *where* exceptions can be thrown. We will now use that information to discover, for each exception, *why* it might be thrown. Specifically, we use inter-procedural symbolic execution to discover predicates over program variables that would be sufficient to trigger a **throw** statement. In some cases we are able to prune infeasible (i.e., over-constrained) paths. Figure 4.7 describes this algorithm formally. The algorithm produces a mapping D ; if $D(m, e) = doc$ then m can raise exception e when doc is true.

The algorithm is a fixed point computation using a worklist of methods. Each method is processed in turn (line 8) and a fixed point is reached when there is no change in the inferred documentation (line 21). Processing a method involves determining path predicates that describe conditions under which exceptions may be propagated from that method.

To process a method, we first enumerate all of the control flow paths (line 10) that can lead from the method head to the statements that we have previously determined can raise exceptions. We construct these control flow paths by starting at the exception-throwing statement and working backward through the control flow graph of the method until we reach the method entry point; we ignore forward edges during this traversal and thus obtain loop-free paths. When a statement has two predecessors, we duplicate the dataflow information and explore both: our analysis is thus path-sensitive and potentially exponential.

Each full control flow path is then symbolically executed (line 11, as in, e.g., [17]). We track conditional statements (e.g., **if** and **while**) and evaluate their guarding predicates using the current symbolic values for variables. We collect the resulting predicates; conjuncted together they form a constraint called a *path predicate* that describes conditions under which that path can be taken [96, 148, 149, 150]. An off-the-shelf path predicate analysis could also be used; we interleave path predicate generation with fixed-point method processing for efficiency.

If the exception-throwing statement is a dynamic dispatch method invocation, we process the dataflow information for each concrete method that may be invoked (line 12). We use an off-the-shelf conservative receiver class analysis [146] to obtain this information (line 13). A method invocation is thus handled by

Input: Map M : method \rightarrow exception information.
Input: Receiver-class information R .
Input: Program call graph C (built using R).
Output Map D : method \times exception \rightarrow predicate.

```

1: for all all methods  $m \in C$  and all exceptions  $e$  do
2:    $D(m, e) \leftarrow \text{false}$ 
3: end for
4:  $Worklist \leftarrow$  methods in  $C$ 
5: Topological sort  $Worklist$ 
6: while  $Worklist$  is not empty do
7:    $c \leftarrow$  remove next cycle of methods from  $Worklist$ 
8:   for all  $m \in c$  do
9:     for all  $(e, l, d) \in M(m)$  do
10:      for all loop-free paths  $p$  from  $start$  to  $l$  do
11:         $Preds \leftarrow$  symbolically execute  $p$ 
12:        if  $l$  is a method call then
13:          for all methods  $m'$  in  $R(l)$  do
14:             $D(m, e) \leftarrow D(m, e) \vee (Preds \wedge D(m', e))$ 
15:          end for
16:        else
17:           $D(m, e) \leftarrow D(m, e) \vee Preds$ 
18:        end if
19:      end for
20:    end for
21:    if  $D$  changed then
22:      Add  $m$  and methods calling  $m$  to  $Worklist$ 
23:    end if
24:  end for
25: end while

```

Figure 4.7: Algorithm for inferring exception documentation that explains *why* an exception is thrown. If $D(m, e) = P$ then method m can raise exception e when P is true; the predicate P is the documentation.

assigning the actual arguments to the formal parameters and including the body of the callee. We thus model execution paths that extend all of the way to the original **throw** statement. If the callee method m' raises an exception e under conditions $D(m', e)$ and the caller m can reach that point with path predicate $Preds$ then the propagated exception occurs on condition $D(m', e) \wedge Preds$ (line 14).

To guarantee termination we only enumerate and explore paths that do not contain loops involving backwards branches. That is, we only consider paths in which loop bodies are evaluated at most once. Because we do not filter out paths with infeasible path predicates, employing this common heuristic [19] will *not* cause us to miss exception-throwing statements. Instead, it will cause us to generate potentially-inaccurate predicates, and thus potentially-inaccurate documentation, in some cases. We make this trade off favoring speed over precision because exception-throwing conditions typically do not depend on particular values of loop induction variables (see Section 4.6). We are interested in documenting exceptions in terms of API-level variables and not in terms of local loop induction variables.

After a fixed point is reached, the path predicate becomes our documentation for the exception instance. If one exception type can be thrown via multiple different paths, those path predicates are combined with a logical disjunction to form a single path predicate. The result is, for each method and for each exception raised by that method, a boolean expression over program variables which, if satisfied at runtime, is sufficient to cause the exception to be raised.

4.5.1 Documentation Post-Processing

We post-process and pretty-print the resulting path predicates to produce the final human-readable documentation output of our tool. We employ a small number of recursive pattern-matching translations to phrase common Java idioms more succinctly. For example:

- `true` becomes `always`
- `false` \vee `x` becomes `x`
- `x != null` becomes “`x` is not null”
- `x instanceof T` becomes “`x` is a `T`”
- `x.hasNext()` becomes “`x` is nonempty”
- `x.iterator().next()` becomes “`x`.{some element}”

We also apply some basic boolean simplifications (e.g., replacing `x && not(x)` with “false”). The most complicated transformation we use replaces `(x && y) || (x && z)` with `x && (y || z)`. An off-the-shelf term rewriting engine or algebraic simplification tool could easily be employed; in our experiments the generated exception documentation did not involve terms that could be simplified arithmetically.

4.6 Generated Documentation Quality

The purpose of our algorithm is the automatic construction of useful documentation for exceptions. In this section we compare the documentation produced by our prototype implementation with pre-existing documentation already present in programs.

4.6.1 Comparing Automatic with Human Documentation

Having ran our tool on each of the benchmarks in Table 4.2 and generated documentation for each exception, for comparison purposes we now restrict attention to only those instances for which a human-written JAVADOC comment exists. For our benchmarks there were 951 human-documented exception instances. We paired each existing piece of documentation with the documentation suggested by our tool and then manually reviewed

Name	Paths	Instances	Mean depth	RCA	EFA	DocInf	Total
Azureus	93,052	17,296	3.30	703.2s	2.32s	504.56s	1,210s
DrJava	1,288	317	1.66	606.0s	0.22s	46.33s	653s
FindBugs	9,280	1,160	1.99	431.0s	0.27s	62.00s	493s
FreeCol	79,620	6,635	4.18	454.5s	2.49s	135.80s	593s
hsqldb	30,569	2,779	2.98	153.6s	0.84s	323.77s	478s
jEdit	6,216	588	4.80	430.1s	0.16s	54.40s	485s
jFreeChart	17	9	1.11	368.7s	0.01s	0.02s	369s
Risk	74	33	0.91	374.0s	0.09s	10.90s	385s
tvBrowser	1,557	283	5.37	490.6s	1.30s	43.30s	535s
Weka	18	14	0.57	458.0s	0.08s	55.34s	513s
Total or Mean	221,691	29,114	2.69	4,470s	7.79s	1,236s	5,714s

Table 4.2: Measurements from running our tool on each benchmark, generating documentation for every exception. “Paths” is the number of control flow paths enumerated and symbolically executed. “Instances” is the number of documentations generated. “Mean Depth” is the mean distance between the method where the documentation is generated, and the original throw statement. “RCA” is runtime for receiver class analysis (we did not implement this) and includes the time for loading and parsing the program files. “EFA” is the runtime for Exception Flow Analysis. “DocInf” is runtime for documentation generation and includes all post processing. All experiments were conducted on a 2GHz dual-core system.

each documentation pair. We categorized each according whether the tool-generated documentation was *better*, *worse*, or about the *same* as the human-created version present in the code.

We consider the tool-generated documentation to be better when it is more precise. For example:

```
Worse:  if inappropriate
(Us) Better: params not a KeyParameter
```

We also consider our documentation to be better when it contains more complete information or otherwise includes cases that were forgotten in the human-written documentation. For example:

```
Worse:  id == null
(Us) Better: id is null or id.equals("")
```

Often, both pieces of documentation conveyed the condition. For example:

```
Same: has an insufficient amount
      of gold.
(Us) Same: getPriceForBuilding() >
          getOwner().getGold()
```

In all other cases, we considered the tool-generated documentation to be worse. This typically happened when human created documentation contained special insight into the relationship between program variables, or expressed high-level information about the program state that could not be readily inferred from a set of predicates without additional knowledge. For example:

```
Better: the queue is empty
(Us) Worse: m_Head is null
```

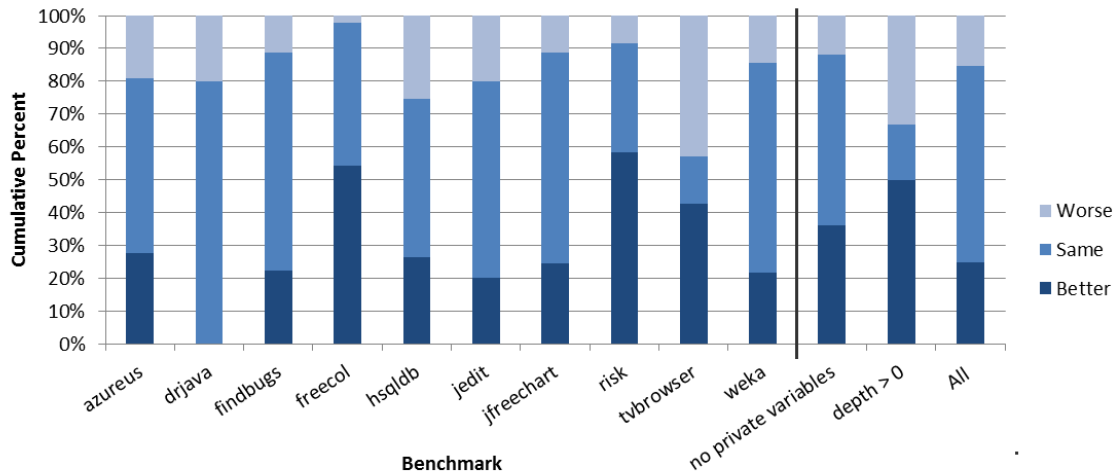


Figure 4.8: The relative quality of the documentation produced by our tool for 951 exception instances in 1.9MLOC that were already human-documented.

All ties or difficult decisions were resolved in favor of the existing human-written documentation. The total time taken to evaluate all 951 documentation pairs was 3.5 hours; the determination was usually quite direct.

Figure 4.8 shows relative quality of the documentation produced by our tool for the 951 exception instances in our benchmarks. The ten columns on the left give breakdowns for individual benchmarks. The three rightmost columns give overall performance; we describe them in more detail below.

We also measured whether the generated conditions could be expressed strictly in terms of non-private API-level interface variables. This property is generally beyond our control, although techniques involving Craig interpolants can be used to express predicates in terms of certain variables instead of others [151]. Predicates over private or local variables might not be useful because their meanings might not be clear at the API-level, and because users of the system cannot directly affect them. Furthermore, documentation involving private variables has the potential to expose implementation details that were intended to be hidden.

In our experiments 29% of the documentation instances involved private or local variables; 71% involved parameters and public variables only. Surprisingly, however, documentation instances involving private variables were nearly as good (83% vs 88% same as or better than existing) as those that were constructed strictly in terms of public variables. This follows largely from the descriptiveness of private variable names in our benchmarks. Consider this indicative example from FINDBUGS:

```
Same: the class couldn't be found
(Us) Same: classesThatCantBeFound.
        contains(className)
```

Even though `classesThatCantBeFound` is a private field, and we may not even know what type of object it is, this generated documentation is still as good as the human-written one. Not all private variables were descriptive, as this example from WEKA shows:

```
Better: Matrix must be square
(Us) Worse: m == n
```

Without knowing that `m` and `n` are matrix dimensions, our documentation is not useful. In general, we need not always reject generated documentation that contains a private reference. The decision to accept or reject such documentations depends on several factors: the extent to which encapsulation is a concern, the relative readability of private vs. public data, and on the intended purpose of the documentation (e.g., internal or external use).

Finally, note that in this section we have provided experimental evidence of the utility of our tool primarily with respect to low-depth (i.e., < 4) exceptions: those for which we have a significant baseline for comparison. While higher-depth exceptions will, in general, be represented by more complex predicates, it is not clear at what point they become too complex to be useful to human readers as documentation. In any case, we can see in Table 4.2 that our inference algorithm does scale to large programs, where nearly 100,000 control flow paths are enumerated and average depths sometimes exceed five. In practice, the determination of which exception types and depths should be documented is likely to depend on program-specific concerns (e.g., the meaning of the exception) as well as usage (e.g., whether the documentation is to be employed as a debugging aid or an API-level documentation supplement, etc.).

4.6.2 Evaluating Copy-and-Paste Documentation

As a final experiment, we take advantage of our framework for documentation inference to investigate whether the large amount of copy-and-paste documentation discovered in Section 4.4.1 is of similar quality to the rest of the documentation in the program. As a proxy for quality, we use a comparison against our tool-generated output. If, on average, copy-and-paste documentation is worse than what our tool would provide and unique documentation is better than what our tool would provide, then we claim that copy-and-paste documentation is worse than unique documentation.

For this experiment we re-use the documentation comparison data from before, but split the set of documentation instances into two sets for each benchmark. One set contains documentation instances that are identical to some other documentation instance in the program (i.e., copy-and-paste documentation). The other set contains those that are unique. We then score each human-written documentation instance as follows:

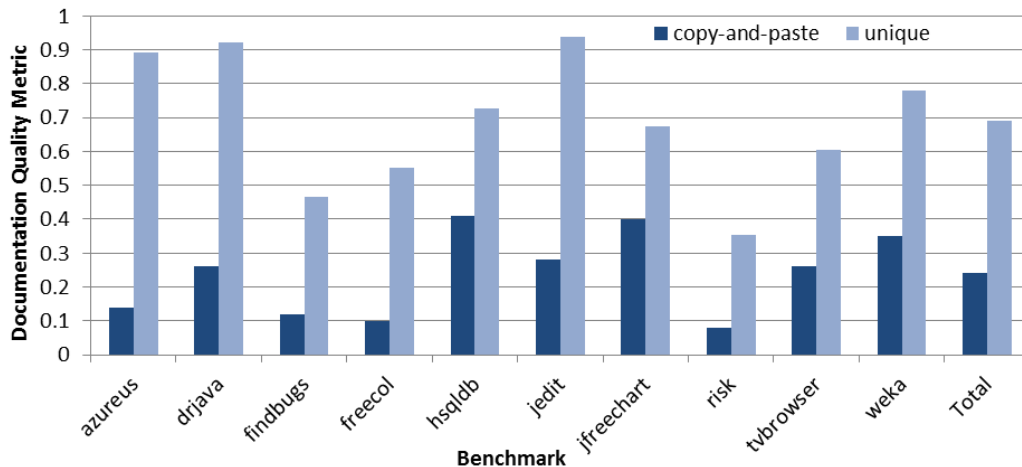


Figure 4.9: Comparison of human-written unique exception documentation strings with human-written strings that are used at least twice (i.e., copy-and-paste documentation).

Comparison	Points
worse	0
same	0.5
better	1

Note that this experiment is not a measurement of the quality of our tool’s output; instead, we use our tool’s output as a baseline for comparing existing human-written documentation instances. We normalize the score for each set to facilitate this comparison. Figure 4.9 shows the results of this analysis. On average, copy-and-paste documentation is only about 35% as good as unique documentation.

4.7 Conclusion

In languages such as JAVA and C#, thrown exceptions are documented at the API level. We have studied exception documentation in about 2 million lines of code and discovered that it is neither complete nor consistent. Instead, documentation is missing when it is difficult for humans to model exception propagation depth, read the code, or understand its complexity. Over half of existing documentation was created by copy-and-paste.

We present an exception flow analysis that determines which exceptions may be thrown or propagated by which methods; this algorithm is a slight refinement of previous work. The analysis is conservative; it will not miss exceptions but may report false positives. We use the results of that analysis in the primary contribution of this work: an algorithm that characterizes the conditions, or path predicates, under which exceptions may be thrown. This analysis is also conservative; it may generate poor predicates for exceptions that depend on loops or where the call graph is imprecise. We convert these predicates into human-readable

documentation. We are able to generate documentation involving only public and API-level variables in 71% of 951 instances associated with 1.9M lines of code.

The documentation instances we generated are at least as accurate as what was created by humans in 85% of the instances, and are strictly better in 25% of them.

Our study of existing documentation suggests that many exception instances remain undocumented in practice. This is especially true when exceptions are propagated through methods. Our algorithm is completely automatic and handles both propagated and local exceptions. Similarly, our experiments confirm that copy-and-paste documentation is of lower quality than documentation that has been specialized to its context: our algorithm could be used to refine or augment such duplicated documentation. Our approach is efficient enough to be used nightly, taking 95 minutes for two million lines of code, thus reducing drift between an implementation and its documentation.

The time costs are low, no annotations are required, and the potential benefits are large. We believe this work is a solid step toward making automatic documentation generation for exceptions a reality.

Chapter 5

Documenting Program Changes

“It is not the strongest of the species that survive, nor the most intelligent,
but the one most responsive to change.”

– *Charles Darwin*

5.1 Introduction

IN the last chapter we argued that documentation is useful for helping a developer understand code. Of course, developers do not generally work in isolation; it is not unusual for hundreds or thousands of engineers to collaborate on a single code base. Thus, not only is it important to understand an existing program, but it is also important to understand changes to that program, particularly changes made by others. Increasingly, documentation that can aide inter-developer communication is critical [45].

In this chapter, we focus on the documentation of code changes. Much of software engineering can be viewed as the application of a sequence of modifications to a code base. Modifications can be numerous; the linux kernel, which is generally considered to be “stable” changes 5.5 times per hour [152]. To help developers validate changes, triage and locate defects, and generally understand modifications **version control systems** permit the association of a free-form textual **commit message** with each change. The use of such messages is pervasive among development efforts with versioning systems [47]. Accurate and up-to-date commit messages are desired [153], but on average, for the benchmarks we investigate in this chapter, only two-thirds of changes are documented with a commit message that actually describes the change.

The lack of high-quality documentation in practice is illustrated by the following indicative appeal from the **MacPorts** development mailing list: “Going forward, could I ask you to be more descriptive in your

commit messages? Ideally you should state what you’ve changed and also why (unless it’s obvious) ... I know you’re busy and this takes more time, but it will help anyone who looks through the log ...”¹

In practice, the goal of a log message may be to (A) summarize *what* happened in the change itself (e.g., “Replaced a warning with an `IllegalArgumentException`”) and/or (B) place the change in context to explain *why* it was made (e.g., “Fixed Bug #14235”). We refer to these as **WHAT information** and **WHY information**, respectively. While most WHY context information may be difficult to generate automatically, we hypothesize that it is possible to mechanically generate WHAT documentation suitable for replacing or supplementing most human-written summarizations of code change effects.

We observe that over 66% of log messages (250 of 375; random sample) from five large open source projects contain WHAT summarization information, implying that tools like `diff`, which compute the precise textual difference between two versions of a file, cannot replace human documentation effort — if `diff` were sufficient, no human-written summarizations would be necessary. We conjecture that there are two reasons raw `diffs` are inadequate: they are too long, and too confusing. To make change descriptions clearer and more concise, we propose a semantically-rich summarization algorithm that describes the effects of a change on program behavior, rather than simply showing what text changed.

Our proposed algorithm, which we refer to as DELTADOC, summarizes the runtime conditions necessary for control flow to reach the changed statements, the effect of the change on functional behavior and program state, and what the program used to do under those conditions. At a high level this produces structured, hierarchical documentation of the form:

```
1 When calling A(), If X, do Y Instead of Z.
```

Additionally, as a key component, we introduce a set of iterative transformations for brevity and readability. These transformations allow us to create explanations that are 80% shorter than standard `diffs`.

The primary goal of our approach is to reduce human effort in documenting program changes. Experiments indicate that DELTADOC is suitable for replacing the WHAT content in as much as 89% of existing version control log messages. In 23.8% of cases, we find DELTADOC contains more information than existing log messages, typically because it is more precise or more accurate. Moreover, our tool can be used when human documentation is not available, including the case of machine-generated patches or repairs [154].

The main contributions of this chapter are:

- An empirical, qualitative study of the use of version control log messages in several large open source software systems. The study analyzes 1000 messages, finding that their use is commonplace and that they are comprised of both WHAT and WHY documentation.

¹<http://lists.macosforge.org/pipermail/macports-dev/2009-June/008881.html>

- An algorithm (DELTADOC) for describing changes and the conditions under which they occur, coupled with a set of transformation heuristics for change summarization. Taken together, these techniques automatically generate human-readable descriptions of code changes.
- A novel process for objectively quantifying and comparing the information content of program documentation.
- A prototype implementation of the algorithm, and a comparison of its output to 250 human-written messages from five projects. Our experiments, backed by a human study, suggest DELTADOC could replace over 89% of human-generated WHAT log messages.

We begin with an example highlighting our approach.

5.2 Motivating Example

Projects with multiple developers need descriptive log messages, as this quote from the CAYENNE project's development list suggests: "Sorry to be a pain in the neck about this, but could we please use more descriptive commit messages? I do try to read the commit emails, but since the vast majority of comments are CAY-XYZ, I can't really tell what's going on unless I then look it up."² Consider revision 3909 of iTEXT, a PDF library written in Java. Its entire commit log message is "Changing the producer info." This documentation is indicative of many human-written log messages that do not describe changes in sufficient detail for others to fully understand them. For example, neither the type of "the producer" nor the nature of the change to the "info" are clear. Now consider the DELTADOC for the same revision:

```

1 When calling PdfStamperImp close()
2 If indexOf(Document.getProduct()) != -1
3     set producer = producer + "; modified using " + Document.getVersion()
4 Instead of
5     set producer = Document.getVersion() +
6         "(originally created with: " + producer + ")"

```

With this documentation it is possible to determine that the change influences the format by which version information is added to the `producer` string. Our documentation is designed to provide sufficient context for the effect of a change to be understood. Furthermore, we claim that `diff`, a commonly-used tool for viewing the text of code changes, often fails to clarify their effects. Consider revision 3066 to the JABREF bibliography manager also written in Java. It consists of a modification to a single `return` statement, as shown in the `diff` summarization below:

²<http://osdir.com/ml/java.cayenne.devel/2006-10/msg00044.html>

Name	Revisions	Domain	kLOC	Devs
FreeCol	2000–2200	Game	91	33
jFreeChart	1700–1900	Data Presenting	305	3
iText	3800–4000	PDF utility	200	14
Phex	3500–3700	File Sharing	177	13
JabRef	2500–2700	Bibliography	107	28
total	1000		880	91

Table 5.1: The set of benchmark programs used in this study. The “Revisions” column lists the span of commits to the program analyzed for the study. The “Devs” column counts the number of distinct developers (by user ID) that checked in modifications during the lifetime of the project.

```

1 19c19,22
2 <     else return "";
3 ---
4 >     else return pageParts[0];
5 >     //else return "";

```

Although the source code `diff` shows exactly what changed textually, it does not reveal the impact of the change. For example, without understanding more about the variable `pageParts`, the new behavior remains unclear. While `diff` can be instructed to provide additional adjacent lines of context, or can be viewed through a specialized tool to increase available context, such blanket approaches can become increasingly verbose without providing relevant information — as in this example, where `pageParts` is defined six lines above the change, out of range of a standard three-line context `diff`. Similarly, the `diff` does not explain the conditions under which the changed statement will be executed. By contrast, DELTADOC explains what the program does differently and when:

```

1 When calling LastPage format(String s)
2   If s is null
3     return ""
4   If s is not null and
5     s.split("[-]").length != 2
6     return s.split("[-]")[0]
7     Instead of return ""

```

Because it links behavioral changes to the conditions under which they occur, this message yields additional clues beyond the `diff` output. We claim that documentation of this form can provide significant insight into the true impact of a change without including verbose and extraneous information. That is, it can serve to document WHAT a change does. We formalize this intuition by studying the current state of log messages and proposing an algorithm to generate such documentation.

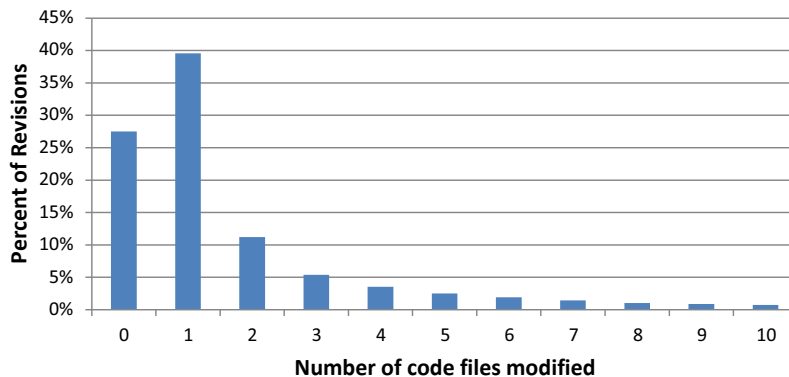


Figure 5.1: Percent of changes modifying a given number of source code files. This graph reports modifications to code files only: file additions or deletions, as well as modifications to non-code files, are excluded. By this metric, the majority of revisions (84%) involve changes to three or fewer source code files.

5.3 Empirical Study of Log Messages

In this section, we analyze the reality of commit log documentation for five open source projects. The benchmarks used are shown in Table 5.1; they cover a variety of application domains. From each project, we selected a range of two-hundred revisions taking place after the earliest, most chaotic stages of development. The projects averaged 18 developers. While even single-author projects benefit from clear documentation, the case for such documentation is particularly compelling when it is a form of communication between individuals.

We first note that log messages are nearly universal. Across the 1000 revisions studied, 99.1% had non-empty log messages. Since such human-written documentation occurs so pervasively, we infer that it is a desired and important part of such software development. However, humans desire up-to-date, accurate documentation [45], and the varying quality of extant log messages (see Section 5.5.2) suggests that many commits could profit from additional explanation [18].

The average size of non-empty human-written log messages is 1.1 lines. The average size of the textual `diff` showing the code change associated with a commit was 37.8 lines. Humans typically produce concise explanations or single-line references to bug database numbers. The longest human-written log-message was 7 lines. We hypothesize that messages of the same order of magnitude (i.e., 1–10 lines) would be in line with human standards, while longer explanations (e.g., 37-line `diffs`) are insufficiently concise. We use lines because they are a natural metric for size, but these findings also apply to raw character count.

We next address the scope of commits themselves. In Figure 5.1 we count the number of already-existing source code files that are modified in a given change. Note that this number may be zero if the change introduces new files, deletes old files, or changes non-code files (e.g., text or XML files, etc.) which our

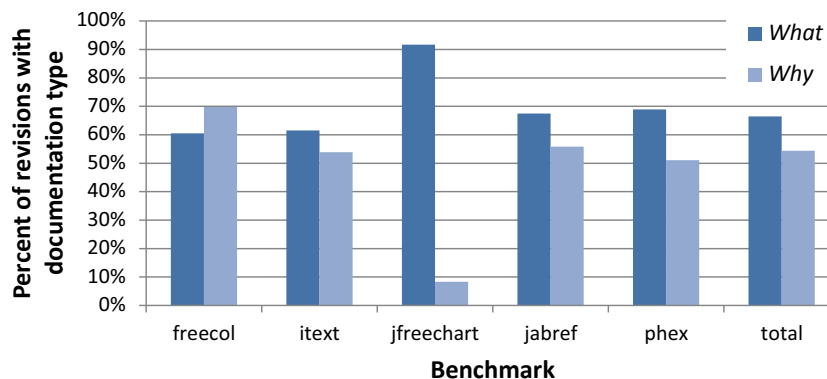


Figure 5.2: Percent of changes with WHAT and WHY documentation. In this study 375 indicative documented changes (about 75 from each benchmark; see Section 5.5.2 for methodology) were manually annotated as WHAT, WHY or both. WHAT documentation is 12% more common on average.

algorithm does not document. We note that the majority of changes edit one to three files, suggesting that a lightweight analysis that focuses on a small number of files at a time would be broadly applicable. These findings are similar to those of Purushothaman and Perry [155].

Finally, we characterize how often human-written log messages contain WHAT and WHY information. For this portion of the study, we selected a subset of 75 indicative log messages from each benchmark (see Section 5.5.2 for details) and manually annotated them as containing WHAT information, WHY information, or both. For this annotation, WHAT information expresses one or more specific relations between program objects (e.g., `x is null`). In Section 5.5.2 we describe the annotation process in detail. WHY information was defined to be all other text. In Figure 5.2 we see that humans include both WHAT and WHY information with a slight preference for WHAT. The FREECOL project, an interactive game, is an outlier: its messages often reference game play. We conclude that a significant amount of developer effort is exercised in composing WHAT information.

Given this understanding of the human-written log messages, the next section describes our approach for producing moderately-sized and human-readable WHAT documentation for arbitrary code changes.

5.4 Algorithm Description

We propose an algorithm called DELTADOC that takes as input two versions of a program and outputs a human-readable textual description of the effect of the change between them on the runtime behavior of the program (i.e., the output is WHAT documentation). Our algorithm produces documentation describing changes to methods. It does not document methods that are created from nothing or entirely deleted. Instead,

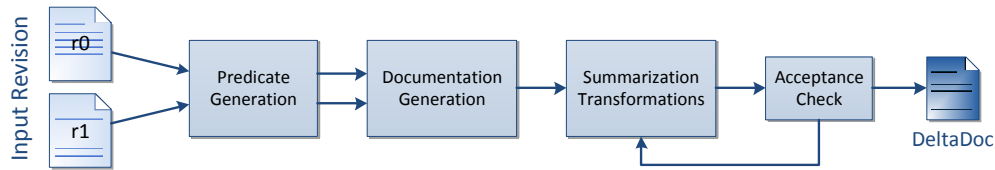


Figure 5.3: High-level view of the architecture of DELTADOC. Path predicates are generated for each statement. Inserted statements, deleted statements, and statements with changed predicates are documented. The resulting documentation is subjected to a series of lossy summarization transformations until it is deemed acceptable (i.e., sufficiently concise).

any such methods are simply listed in a separate step. Our algorithm is intraprocedural. If a change involves multiple methods or files we document changes to each method separately.

DELTADOC follows a pipeline architecture shown in Figure 5.3. In the first phase, as detailed in Section 5.4.1, we use symbolic execution to obtain path predicates for each statement in both versions of the code. In phase two, we identify and document statements which have been added, removed, or have a changed predicate (Section 5.4.2). Third, we iteratively apply lossy summarization transformations (Section 5.4.3) until the documentation is sufficiently concise. Finally, we group and print the statements in a structured, readable form.

5.4.1 Obtaining Path Predicates

Our first step is to obtain intraprocedural *path predicates*, formulae that describe conditions under which a path can be taken or a statement executed [96, 148, 149, 150]. To process a method, we first enumerate its loop-free control flow paths. We obtain loop-free paths by adding a statement to a path at most once: in effect, we consider each loop to either be taken once, or not at all. This decision can occasionally result in imprecise or incorrect documentation, however we show in Section 5.5.2 that this occurs infrequently in practice.

Each control flow path is symbolically executed (as in Section 4.5). We track conditional statements (e.g., `if` and `while`) and evaluate their guarding predicates using the current symbolic values for variables. When a statement has two successors, we duplicate the dataflow information and explore both: our analysis is thus path-sensitive and potentially exponential. We collect the resulting predicates; in conjunction, they form the path predicate for a given statement in the method. Some statements, such as the assignments to local variables, are heuristically less relevant to explaining WHAT a change does than others, such as function calls or return statements. Our enumeration method is parametric with respect to a *Relevant* predicate that flags such statements. In our prototype implementation, invocation, assignment, `throw`, and `return` statements are deemed relevant. We symbolically execute all *program expressions* contained in these statements and

store the result — later documentation steps will use the symbolic (i.e., partially-evaluated) value instead of the raw textual value to describe a statement (i.e., E in Figure 5.4). An off-the-shelf path predicate analysis and symbolic execution analysis could also be used with relevancy filtering as a post-processing step; we combine the three for efficiency.

5.4.2 Generating Documentation

Input: Method $P_{old} : \text{statements} \rightarrow \text{path predicates}$.

Input: Method $P_{new} : \text{statements} \rightarrow \text{path predicates}$.

Input: Mapping $E : \text{statements} \rightarrow \text{symbolic statements}$.

Output: emitted structured documentation

Global: set $MustDoc$ of statements = \emptyset

```

1: let  $Inserted = \text{Domain}(P_{new}) \setminus \text{Domain}(P_{old})$ 
2: let  $Deleted = \text{Domain}(P_{old}) \setminus \text{Domain}(P_{new})$ 
3: let  $Changed = \emptyset$ 
4: for all statements  $s \in \text{Domain}(P_{new}) \cap \text{Domain}(P_{old})$  do
5:   if  $P_{new}(s) \neq P_{old}(s)$  then
6:      $Changed \leftarrow Changed \cup \{s\}$ 
7:   end if
8: end for
9:  $MustDoc \leftarrow Inserted \cup Deleted \cup Changed$ 
10: let  $Predicates = \emptyset$  (a multi-set)
11: for all statements  $s \in MustDoc$  do
12:   let  $C_1 \wedge C_2 \cdots \wedge C_n = P_{new}(s) \wedge P_{old}(s)$ 
13:    $Predicates \leftarrow Predicates \cup \{C_1\} \cup \cdots \cup \{C_n\}$ 
14: end for
15:  $Predicates \leftarrow Predicates$  sorted by frequency
16: call HierarchicalDoc ( $\emptyset, , P_{new}, P_{old}, Predicates, E$ )

```

helper fun HierarchicalDoc($p, P_{new}, P_{old}, Predicates, E$)

```

17: for all statements  $s \in MustDoc$  with  $P_{new}(s) = p$ , sorted do
18:   output “Do Describe( $E(s)$ )”
19:    $MustDoc \leftarrow MustDoc \setminus \{s\}$ 
20: end for
21: for all statements  $s \in MustDoc$  with  $P_{old}(s) = p$ , sorted do
22:   output “Instead of Describe( $E(s)$ )”
23:    $MustDoc \leftarrow MustDoc \setminus \{s\}$ 
24: end for
25: for all predicates  $p' \in Predicates$ , in order do
26:   if  $MustDoc \neq \emptyset$  then
27:     output “If Describe( $p$ )” and tab right
28:     call HierarchicalDoc( $p \wedge p', P_{new}, P_{old}, Predicates$  with all occurrences of  $p'$  removed,  $E$ )
29:     output tab left
30:   end if
31: end for

```

Figure 5.4: High-level pseudo-code for Documentation Generation.

In this phase of the DELTADOC algorithm, two sets of statement \rightarrow predicate mappings, obtained from path predicate generation and representing the code before and after a revision, are used to make documentation of the form **If X, Do Y Instead of Z**.

Figure 5.4 shows the high-level pseudo-code for this phase. The algorithm matches statements with the same bytecode instruction and symbolic operands, enumerating all statements that have a changed predicate or that are only present in one version (lines 1–9). In our prototype implementation for Java programs, statements are considered to be unchanged if they execute the same byte code instruction under the same conditions.

Statements are then grouped by predicate (lines 10–14) and frequently-occurring predicates are handled first (line 15). Documentation is generated in a hierarchical manner. All of the statements guarded by a given predicate are sorted by line number (lines 17 and 21). Statements that are exactly guarded by the current predicate are documented via **Do** or **Instead of** phrases (lines 18 and 22) and removed from consideration. If multiple statements are guarded by the current predicate, they are printed in the same order they appear in the original source. If undocumented statements remain, the next most common predicate is selected (line 25) and documented using a **If** phrase (line 27). The helper function is then called recursively to document statements that might now be perfectly covered by the addition of the new predicate. The process terminates when all statements that must be documented have been documented; the process handles all such statements because the incoming set of interesting predicates is taken to be the union of all predicates guarding all such statements.

The statements and predicates themselves are rendered via a *Describe* function that pretty-prints the source language with minor natural language replacements (e.g., “is not” for `!=`). This algorithm produces complete documentation for all of its input (i.e., all heuristically *Relevant* statements see Section 5.4.3). For large changes, however, the documentation may be insufficiently concise; the next phase of the algorithm summarizes the result.

Finally, our algorithm also prints a list of fields or methods that have been added or removed. This mimics certain styles of human-written documentation, as in the following example from JFREECHART revision 1156:

```
(tickLength): New field,
(HighLowRenderer): Initialise tickLength,
(getTickLength): New method
```

5.4.3 Summarization Transformations

Summarization is a key component of our approach. Without explicit steps to reduce the size of the raw documentation created in Section 5.4.2, the output is likely too long and too confusing to be useful. We base

this judgment on the observation that human-written log messages are on the order of 1–10 lines long (see Section 5.3). Without summarization, the raw output is, on average, twice as long as `diff`. In Chapter 2, we found that the length of a snippet of code and the number of identifiers it contained were the two most relevant factors — and both were negatively correlated with readability. To mitigate the danger of unreadable documentation, we introduce a set of summarization transformations.

Our transformations are synergistic and can be sequentially applied, much like standard dataflow optimizations in a compiler. Just as copy propagation creates opportunities for dead code elimination when optimizing basic blocks, removing extraneous statements creates opportunities for combining predicates when optimizing structured documentation. Unlike standard compiler optimizations, however, not all of our transformations are semantics-preserving. Instead, many are *lossy*, sacrificing information content to save space. As the number of facts to document increases, so to does the number of transformations applied.

In the rest of this subsection, we detail these transformations. Where possible, we relate the transformations below to standard compiler optimizations that are similar in spirit; because our transformations are lossy and apply to documentation, this mapping is not precise.

Finally, note that our algorithm runs on control flow graphs, and is suitable for use with common imperative languages. Because our prototype implementation is for Java, we describe some aspects of the algorithm (particularly readability transformations in Section 5.4.3) in the context of Java. However, all of these transformations have natural analogs in most other languages.

Statement Filters. We document only a set of *Relevant* statements. Method invocations, which potentially represent many statements, are retained. We also retain assignments to fields, since they often capture the impact of code on visible program state. Finally, we document `return` and `throw` statements, which capture the normal and exceptional function behavior of the code. Notably, we avoid documenting assignments to local variables; our use of symbolic execution when documenting statements (as used in lines 18 and 22 of Figure 5.4) and predicates (line 27 of Figure 5.4) means that many references to local variables will be replaced by references to their values. We also take advantage of a standard Java textual idiom for accessor (i.e., “getter”) methods: we do not document calls to methods of the form `get[Fieldname]()`, since they are typically equivalent to field reads.

Single Predicate transformations. Our algorithm produces hierarchical, structured documentation, where changes are guarded by predicates that describe when they occur at run-time. A group of associated statements may include redundant or extraneous terms. We thus consider all of the subexpressions in the statement group and:

1. Drop method calls already appearing in the predicate.

2. Drop method calls already appearing in a **return**, **throw**, or assignment statement.
3. Drop conditions that do not have at least one operand in documented statements.

Transformation 1 and 2 can be viewed as a lossy variant of common subexpression elimination: it is typically more important to know that a particular method was called than how many times it was called. Transformation 3 is based on the observation that predicates can become large and confusing when they contain many conditions that are not directly related to the statements they guard. We thus remove conditions that do not have operands or subexpressions that also occur in the statements they guard. For example:

```
If s != null and x is true,
    b is true and c is true, return s  →    If s != null, return s
```

because the guards discussing `a`, `b`, and `c` are unrelated to the action `return s`. However, this transformation is only applied if there are at least three guard conditions and at least one would remain; these heuristics work in practice to prevent too much context from being lost. Thus, we retain `If s is null, throw new Exception()` even though it does not share subexpressions with its statement.

Whole Change transformations reduce redundancy across the documentation of multiple changes to a single method. They are conceptually akin to global common subexpression elimination or term rewriting. After generating the documentation for all changes to a method, we:

1. Inline **Instead of** conditions.
2. Inline **Instead of** predicates.
3. Add hierarchical structure to avoid duplication.

Transformation 1 converts an expression of the form:

```
If P, Do X Instead of If P, Do Y  →    If P, Do X Instead of Y
```

Similarly, transformation 2 converts:

```
If P, Do X Instead of if Q, Do X  →    Do X If P, Instead of If Q
```

The third transformation reduces predicate redundancy by nesting specific guards “underneath” general guards. Given two adjacent pieces of documentation, we merge them into a single piece of hierarchically-structured documentation by recursively finding and elevate the largest intersection of guard clauses they share. This is conceptually similar to conditional hoisting optimizations. For example, we convert:

```
If P and Q, Do X
If P and Q and R, Do Y  →    If P and Q,
                             Do X
                             If R,
                             Do Y
```

Simplification is a lossy transformation that saves space by eliding the targets and arguments of method calls. For example, `obj.method(arg)` may be transformed to `obj.method()`, `method(arg)`, or simply `method()`.

In addition, documented predicates that are still greater than 100 characters are not displayed. This heuristic limit could ideally be replaced by a custom viewing environment where users could expand or collapse predicates as desired.

High-level summarization is applied only in the case of large changes, where the documentation is still too long to be considered acceptable (e.g., more than 10 lines). For sweeping changes to code, we observe that high-level trends are more important than statement-level details. We thus remove predicates entirely, replacing them with documentation listing the invocations, assignments, return values and throw values that have been added, removed or changed. We have observed that this information often is sufficient to convey what components of the system were affected by the change when it would be impractical to describe the change precisely.

This transformation is similar to high-level documentation summarization from the domain of natural language processing, in that it attempts to capture the gist of a change rather than its details.

Readability enhancements are always applied. We employ a small number of recursive pattern-matching translations to phrase common Java idioms more succinctly in natural language. For example:

- `true` is removed
- `x != 0` becomes “`x` is not null”
- `x instanceof T` becomes “`x` is a `T`”
- `x.hasNext()` becomes “`x` is nonempty”
- `x.iterator().next()` becomes “`x`->{some element}”

Similar transformations have proved effective in Chapter 4 to generate readable documentation for exceptions.

5.5 Evaluation

Information and conciseness are important aspects of documentation [45]. A critical tradeoff exists between the information conveyed and the size of the description itself. In the extreme, printing the entire artifact to be documented is perfectly precise, but not concise at all. Conversely, empty documentation has the virtue of being concise, but conveys no information.

Our technique is designed to balance information and conciseness by imposing more summarization steps as the amount of information to document grows. In this evaluation, we aim to establish a tradeoff similar to that of real human developers.

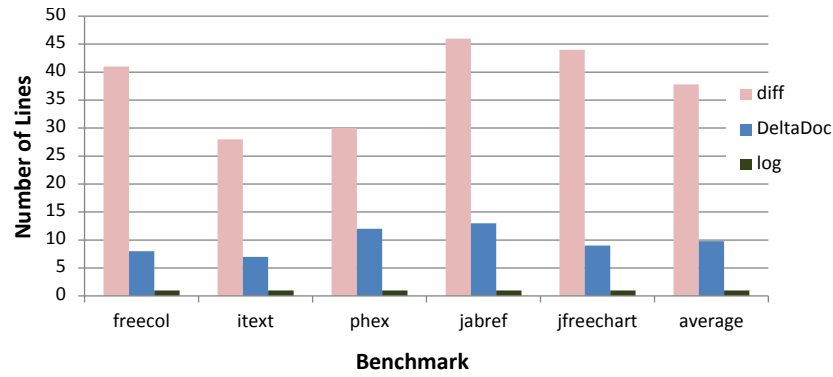


Figure 5.5: Size comparison between `diff`, DELTADOC, and human generated documentation over all 1000 revisions from Table 5.1.

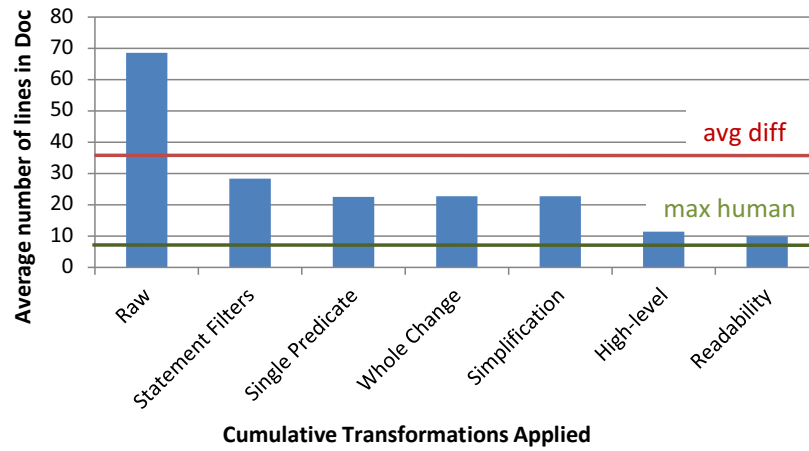


Figure 5.6: The cumulative effect of applying our set of transformations on the size of the documentation for all 1000 revisions from Table 5.1. We include the average size of `diff` output for comparison.

We have implemented the algorithm described in Section 5.4 in a prototype tool for documenting changes to Java programs. In this section, we compare DELTADOC documentation to human-written documentation and `diff`-generated documentation in terms of size (Section 5.5.1) and quality (Section 5.5.2). DELTADOC takes about 1 second on average to document a change, and 3 seconds at most in our experiments. Documenting an entire repository of 10,000 revisions takes about 3 hours.

5.5.1 Size Evaluation

An average size comparison between baseline `diff` output, DELTADOC, and target human documentation is presented in Figure 5.5. On average, DELTADOC is approximately nine lines longer than the single-line documentation that humans typically create, though human-written documentation may be up to seven lines long. Notably, our documentation is about 28 lines shorter than `diff` on average.

Figure 5.6 indicates how this level of conciseness is dependent upon the applications of the transformations described in Section 5.4.3. Our raw, unsummarized documentation is 68 lines long on average and approximately twice the size of `diff` output. Filtering statements reduces the size by 58%. Single predicate transformations restructure this documentation, removing significant redundancy and reducing the total size by another 21%. Whole change transformations are primarily for readability, and actually increase documentation size by about 1%. Simplifications have no effect on line count, but they do reduce the total number of characters displayed by 67%. High-level summarization, which is only applied to about 18% of changes, is effective at reducing the documentation size by another 49% overall. That is, high-level summarization is rarely applied, but when used it reduces overly-large documentation to a more manageable size. Finally, readability enhancements reduce the final line count by 13% (primarily by removing frequently-occurring `If true ... predicates`), however, they affect the total number of characters by less than 1%.

5.5.2 Content Evaluation

When comparing the quality of documentation for program changes, both the revisions selected for comparison and the criteria for quality are important considerations. We address each in turn before presenting our results.

Selection

Not all revisions to a source code repository can form the basis for a suitable comparison. For example, revisions to non-code files or with no human-written documentation do not admit an interesting comparison. We restricted attention to revisions that contained a non-empty log message with WHAT information, modified at least one and no more than three Java files, and modified non-test files (i.e., we ignore changes to unit test packages).

Revisions that touch one, two, or three Java files account for about 66–84% of all the changes that touch at least one file in our benchmarks (see Figure 5.1). We manually inspected revisions and log messages from each benchmark in Table 5.1 until we had obtained 50 conforming changes from each, or 250 total; this required the inspection of 375 revisions.

Documentation Comparison Rubric

We employed a three-step rubric for documentation comparison designed to be objective, repeatable, fine grained, and consistent with the goal of precisely capturing program behavior and structure. The artifacts to be compared are first cast into a formal language which expresses only specific information concerning

program behavior and structure (steps 1 and 2). This transformation admits direct comparison of the information content of the artifacts (step 3).

Step 1: Each program object mentioned in each documentation instance is identified. For DELTADOC, this step is automatic. We inspect each log message for nouns that refer to a specific program *variable*, *method*, or *class*. This criterion admits only *implementation* details, but not *specification*, *requirement*, or other details. The associated source code is used to verify the presence of each object. Objects not named in the source code (e.g., “The User”) are not considered program objects, and are not enumerated.

Step 2: A set of *true* relations among the objects identified in step 1 are extracted; this serves to encode each documentation into a formal language. Again, this process is automatic for DELTADOC. We inspect the human-written documentation to identify relations between the objects enumerated in step 1. Relations where one or both operands are implicit and unnamed are included. The full set of relations we consider are enumerated in Table 5.2, making this process essentially lock-step. Relations may be composed of constant expressions (e.g., 2), objects (e.g., X), mathematical expressions (e.g., $X + 2$), or nested relations. The associated source is used to check the validity of each encoded relation. If manual inspection reveals a relation to be false, it is removed. When in doubt, human-written messages were assumed to be true.

Step 3: For each relation in either the human documentation (H) or DELTADOC (D), we check to see if the relationship is also fully or partially present in the other documentation. For each pair of relations H and D , we determine if H *implies* D written $H \Rightarrow D$, or if $D \Rightarrow H$. A relation *implies* another relation if the operator is the same and each operand corresponds to either the same object, an object that is implicit and unnamed, or a more general description of the object (see below for an example of an unnamed object). If the relations are identical, then $H \Leftrightarrow D$. Again, the associated source code to check the validity of each relation. As with step 2, if the relation is found to be inconsistent with the program source, then it is removed from further consideration.

While steps 1 and 2 are lockstep, step 3 is not because it contains an instance of the *Noun phrase co-reference resolution problem*; it requires the annotator judge if two non-identical names refer to the same object (e.g., `getGold()` and `gold` both refer to the same program object). Automated approaches to this problem have been proposed (e.g., [156, 157]), however we use human annotators who have been shown to be accurate at this task [158].

This process is designed to distill from the message precisely the information we wish to model: the impact of the change on program behavior. Notably, we leave behind other (potentially valuable) information that is beyond the scope of our work (e.g., the impact of the change on performance). Furthermore, this process allows us to precisely quantify the difference in information content. This technique is similar to the NIST ROUGE metric [159] for evaluating machine-generated document summaries against human-written ones.

Type	Arity	Name			
logical	unary	is <i>true</i> is <i>false</i>			
	binary	or and equal to not equal to less-than greater-than less-than or equal to greater-than or equal to			
		programmatic	is empty call / invoke return throw		
			binary	assign to element of inserted in implies instance of	
			edit	unary	added changed removed

Table 5.2: Table of relational operators used for documentation comparison. *Logical* operators describe runtime conditions, *programmatic* operators express language concepts, and *edit* operators describe changes to code structure.

Score Metric

After applying the rubric, we quantify how well DELTADOC captures the information in the log message with a metric we will refer to as the *score*. To compute the score we inspect each implication in the human-established mapping. For each $X \Rightarrow Y$, we add two points to the documentation containing X and one point to the documentation containing Y . This rewards more precise documentation (e.g., “updated version” vs. “changed version to 1.2.1”). For each $X \Leftrightarrow Y$ we add two points to each. For all remaining unmapped relations in the log message we add two points to the total for the log message; we do not for DELTADOC, instead conservatively assuming any additional DELTADOC relations are spurious. The score metric is computed as the fraction of points assigned to the algorithm documentation:

$$score = \frac{points\ for\ DeltaDoc}{points\ for\ DeltaDoc + points\ for\ log\ message}$$

The score metric ranges from 0 to 1, where 0.5 indicates the information content is the same. A score above 0.5 indicates that the DELTADOC contains more information than the log message (i.e., there is at least

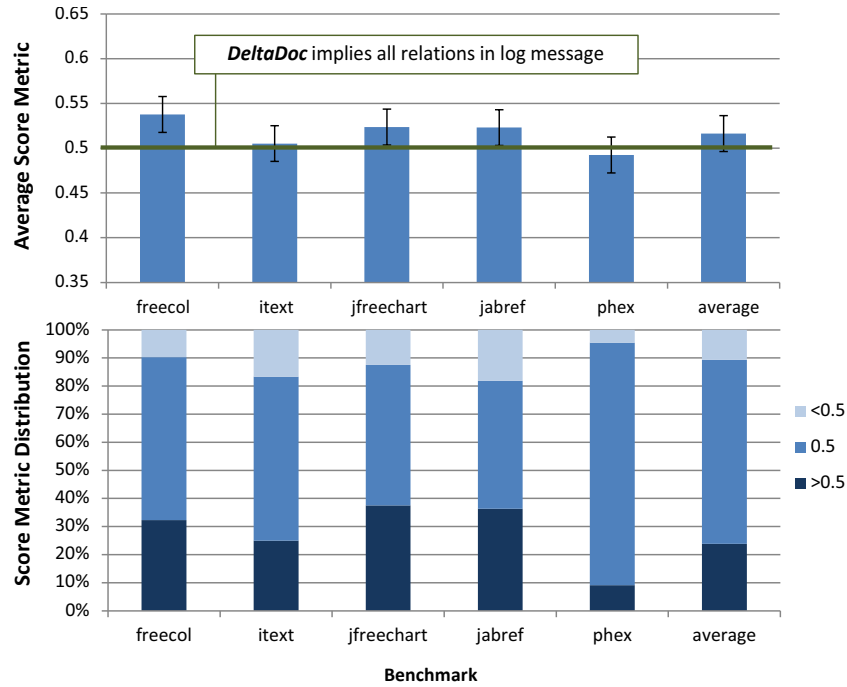


Figure 5.7: The distribution of *scores* for 250 changes in 880kLOC that were already human-documented with WHAT information. A score of 0.5 or greater indicates that the DELTADOC is a suitable replacement for the log message. Error bars represent the maximum level of disagreement from human annotators.

one relation in the DELTADOC not implied by the log message). The maximum score of 1.0 is possible only if the log message does not contain any correct information, but the DELTADOC does. From an information standpoint, we conjecture that a score ≥ 0.5 implies that the DELTADOC is a suitable replacement for human log message: it contains at least the same amount of WHAT information. Figure 5.7 shows the average score we assigned to the documentation pairs from each benchmark.

After following the rubric, we asked 16 students from a graduate programming languages seminar to check our work by reviewing a random sample consisting of 32 documentation pairs. For each change, the annotators were shown the existing log message, the output of our algorithm, and the output of standard `diff`. The annotators were asked to inspect the information mapping we established and provide their own if they detected any errors (see Figure A.2). We then computed the score according to each annotator and compared the results to our original scores. Over all the annotators, we found the maximum score deviation to be 2.1% and the average deviation to be 0.48%. The pairwise Pearson interrater agreement is 0.94 [82].

The results of our study are presented in Figure 5.7. We compared the results on 250 revisions, 50 from each benchmark. Error bars indicate the maximum deviation observed from our human annotators and represent a confidence bound on our data. The average score for each benchmark except PHEX is greater than 0.5. PHEX, however, shows the greatest number of changes with a score of at least 0.5. This is because PHEX

contains a number of large changes that were not effectively documented by DELTADOC according to our metric. These changes received a 0.0 and significantly reduced the average score. In general, our prototype DELTADOC compares most favorably for small to medium sized changes because we impose tight space limitations. In practice, use of interactive visualization similar to hypertext allowing parts of DELTADOC to be expanded or collapsed as needed would likely make such limitations unnecessary.

In 65.5% of cases we calculated the score to be 0.5; DELTADOC contained the same information as the log message. For example, revision 3837 of ITEXT contained the log message “no need to call clear()”. The DELTADOC contains the same relation resulting in a score of 0.5:

```
1 When calling PdfContentByte reset()
2   No longer If stateList.isEmpty(),
3     call stateList.clear()
```

In 23.8% of cases our documentation was found to contain more information (score > 0.5). In revision 2054 of FREECOL, the log message states “Commented unused constant.” The DELTADOC names the constant (score=0.67):

```
1 removed field: EuropePanel : int TITLE_FONT_SIZE
```

In the remaining 10.7% of cases, the DELTADOC contained less information. Typically, this arises because of unresolved aliases or imprecision in either predicate generation or summarization. For example, the log message from revision 3111 of JABREF states “Fixed bug: content selector for ‘editor’ field uses ‘,’ instead of ‘and’ as delimiter.” In this case the DELTADOC did not clearly contain a relation indicating ‘,’ was removed while ‘ and ’ was inserted:

```
1 When calling EntryEditor getExtra
2   When ed.getFieldName().equals("editor")
3     call contentSelectors.add(FieldContentSelector)
```

These findings support our hypothesis that the majority of WHAT information in human-written log messages (about 89%) could be replaced with machine generated documentation without diminishing the amount of WHAT information conveyed — reducing the overall effort required, and leaving humans with additional opportunities to focus on crafting WHY documentation.

5.5.3 Qualitative Analysis

In addition to validation of our information mapping, for each documentation and log message pair, we asked our annotators which they felt “was more useful in describing the change.” In 63% of cases the annotators either had no preference (17%) or preferred the DELTADOC (46%).

We also asked a few free-form qualitative questions regarding our algorithm output. Many participants noted that the strength of the algorithm output is that it is “more specific” when compared to human-written log messages, but simultaneously “not as low level with no context” as compared to `diff` output. Fifteen of the sixteen participants agreed that the algorithm would provide a useful supplement to log messages. Many went further, noting that it is “very useful,” “highly useful,” “would be a great supplement,” “definitely a useful supplement,” and “can help make the logic clear.” In comparison to human log messages the algorithm output was “often easier to understand,” “more accurate ... easy to read,” and “provides more information.” The general consensus was that “having both is ideal,” suggesting that DELTADOC can serve as a useful supplement to human documentation when both are available.

5.5.4 Threats To Validity

Although our results suggest that our tool can efficiently produce documentation for commit messages that is as good as, or better than, human-written documentation, in terms of WHAT information, they may not generalize.

First, the benchmarks we selected may not be indicative. We attempted to address this threat by selecting benchmarks from a wide variety of application domains (e.g., networking, games, business logic, etc.). Second, the revisions we selected for manual comparison may not be indicative. We attempted to mitigate this threat by selecting contiguous blocks of revisions from the middle of the projects’ lifetimes, thus avoiding non-standard practices from the early phases of development. Bird *et al.* note that studies that attempt to tie developer check-ins to bug fixes via defect databases are subject to bias [160]; we mitigate this threat by inspecting all revisions, not just those tied to defect reports.

Second, our manual annotation of documentation quality may not be indicative. We attempted to mitigate this threat by clearly separating WHAT and WHY information in the evaluation and using an almost-mechanical list of criteria. With the subjective and context-dependent WHY information removed, documentation can be evaluated solely on the fact-based WHAT components. Furthermore, we asked 16 annotators to review a random sample of documentation pairs in order to quantify the uncertainty in our quality evaluation and mitigate the potential for bias: the error bars in Figure 5.7 indicate the maximum human disagreement.

5.6 Related Work

Source code differencing is an active area of research. The goal of differencing is to precisely calculate which lines of code have changed between two versions of a program. Differencing tools are widely used in software

engineering tasks including impact analysis, regression testing, and version control. Recently, Apiwattanapong *et al.* [161] presented a differencing tool that employs some semantic knowledge to capture object-oriented “correspondences.” Our work is related to differencing, in that we also enumerate code changes. However, we recognize that the output of `diff` tools is often too verbose and difficult to interpret for use as documentation, since they necessarily list all changes. Our work is different because (1) our goal is to describe the impact of a change rather than the change itself (2) we focus on summarization instead of completeness and (3) our output is intended to be used as human-readable documentation and not as input to another analysis or tool.

Kim and Notkin described a tool called `LSDiff` which discovers and documents highlevel structure in code changes (e.g., refactorings) [162]. `LSDiff` has the potential to complement `DELTA`DOC which instead focuses on precise summarization of low-level differences.

Hoffman *et al.* [163] presented another approach to determining the difference between program versions. Like ours, their analysis focus on program paths. However, their tool is dynamic, and its primary application is regression testing and not documentation.

Automatic natural language document summarization was attempted as early as 1958 [164]. Varadarajan *et al.* [165] note that the majority of systems participating in the past *Document Understanding Conference* and the *Text Summarization Challenge* are extraction based. Extraction based systems extract parts of original documents to be used as summaries; they do not exploit the document structure. However, some analyses as early as Mathis *et al.* [166] in 1973 make significant use of structural information. Our analysis makes heavy use of the structure of our input since programming languages are much more structured than natural ones.

XML document summarization [167] and HTML summarization (for search engine preprocessing or web mining [168, 169, 170]) produce structured output. However, the intent in these cases is not to produce human-readable text, but to increase the speed of queries to the summarized data.

Finally, semantic differencing (e.g., [171, 172]) uses semantic cues in order to characterize the difference between two versions of a document. This characterization is often a binary judgment (e.g., “Is this change important?”), and is suitable for use in impact analysis and regression testing, but not for documentation.

5.7 Conclusion

Version control repositories are used pervasively in software engineering. Log messages are a key component of software maintenance, and can help developers to validate changes, triage and locate defects, and understand modifications. However, this documentation can be burdensome to create and can be incomplete or inaccurate, while undocumented source code `diffs` are typically lengthy and difficult to understand.

We propose DELTADOC, an algorithm for synthesizing succinct, human-readable documentation for arbitrary program differences. Our technique is based on a combination of symbolic execution and a novel approach to code summarization. The technique describes what a code change does; it does not provide the context to explain why a code change was made. Our documentation describes the effect of a change on the runtime behavior of a program, including the conditions under which program behavior changes and what the new behavior is.

We evaluated our technique against 250 human-written log messages from five popular open source projects. We proposed a metric that quantifies the information relationship between DELTADOC and human-written log messages. From an information standpoint, we found that DELTADOC is suitable for replacing about 89% of log messages. A human study with sixteen annotators was used to validate our approach. This suggests that the “what was changed” portion of log message documentation can be produced or supplemented automatically, reducing human effort and freeing developers to concentrate on documenting “why it was changed.”

Chapter 6

Synthesizing API Usage Examples

“In learning the sciences, examples are of more use than precepts.”

– *Isaac Newton*

6.1 Introduction

WE have observed that an automated tool can be useful for synthesizing documentation that describes program behavior. In this chapter we take automated documentation a step further by generating new code suitable for instruction.

Much of modern development focuses on interfacing between powerful existing frameworks and libraries. In reports by and studies of developers, **Application Program Interface (API)** use examples have been found to be a key learning resource [173, 49, 50, 51, 52, 53]. That is, documenting how to *use* an API is preferable to simply documenting the function of each of its components. One study found that the greatest obstacle to learning an API in practice is “insufficient or inadequate examples” [54]. In this chapter, we present an algorithm that automatically generates API usage examples. Given a data-type and software corpus (i.e., a library of programs that make use of the data-type), we extract abstract use-models for the data-type and render them in a form suitable for use by humans as documentation.

The state of the art in automated support for usage examples is known as *code search*. Typically, the problem is phrased as one of ranking concrete code snippets on criteria such as “representativeness” and “conciseness.” In 2009, Zhong *et al.* described a technique called MAPO for mining and recommending example code snippets [59]. More recently, Kim *et al.* presented a tool called EXOADOCS which also finds and ranks code examples for the purpose of supplementing JAVADOC embedded examples [58]. Such examples

can be useful, but they are very different from human-written examples. Mined examples often contain extraneous statements, even when slicing is employed. In addition, they often lack the context required to explicate the material they present. In general, mined examples are long, complex, and difficult to understand and use. Good human-written examples, on the other hand, often present only the information needed to understand the API and are free of superfluous context. Human written documentation has two important disadvantages, however: it requires a significant human effort to create, and is thus often not created; and it may not be representative of, or up-to-date with, actual use.

In this chapter we present a technique for automatically synthesizing human-readable API usage examples which are well-typed and representative. We adapt techniques from specification mining [55] to model API uses as finite state machines describing method call sequences, annotated with control flow information. We use data-flow analysis to extract important details about each use beyond the sequence of method calls, such as how the type was initialized and how return values are used. We then abstract concrete uses into high-level examples. Because a single data-type may have multiple common use scenarios, we use clustering to discover and coalesce related usage patterns before expressing them as documentation.

Our generated examples display a number of important advantages over both state-of-the-art code search and human written examples, both of which we compare to in a human study. Unlike mined examples, our generated examples contain only the program statements needed to demonstrate the target behavior. Where concrete examples can be needlessly specific, our examples adopt the most common types and names for identifiers. Unlike human-written examples, our examples are, by construction, well-formed syntactically and well-typed. Where previous approaches to code ranking adopted simple heuristics based on length and a simple use count (e.g., [174, 175]), our abstract examples are structured and generated with a robust and well-defined notion of representativeness. Because our approach is fully automatic, the examples are also cheap to construct and can be always up-to-date. Additionally, their well-formedness properties make them ideal for code completion and other automated tasks [176].

The main contributions of this chapter are:

- A study of API usage examples. We characterize gold-standard human-written examples from the Java SDK. Furthermore, we present results from a large survey on example quality and utility.
- An algorithm for the automatic construction of example documentation. The algorithm takes as input a software corpus and a target data-type to be documented. It produces a ranked list of well-typed, human-readable code snippets which exemplify typical use of that data-type in the provided corpus.
- A prototype implementation of the algorithm, and a comparison of its output to 35 instances of human-written example documentations. Our experiments, backed by a human study with over 150 participants, suggest our tool could replace as many as 82% of human-generated examples.

6.2 Motivating Example

In modern software development, API documentation tools such as JAVADOC have become increasingly prevalent, and variants exist for most languages (e.g., PYTHONDOC, OCAMLDON, etc.). One of the principles of JAVADOC is “including examples for developers.” [177] Not all examples are created equal, however. Features such as conciseness, representativeness, well-chosen variable names, correct control flow, and abstraction all relate to documentation quality.

Consider Java’s `BufferedReader` class, which provides a buffering wrapper around a lower-level, non-buffered stream. The human-written usage example included in the official Java Development Kit, version 6 [178] is:

```
1  BufferedReader in = new BufferedReader(new FileReader("foo.in"));
```

While this example has the merit of being concise, it shows only how to create a `BufferedReader`, not how to use one. By contrast, our algorithm produces:

```
1  FileReader f; //initialized previously
2  BufferedReader br = new BufferedReader(f);
3  while(br.ready()) {
4      String line = br.readLine();
5      //do something with line
6  }
7  br.close();
```

This exemplifies one common usage pattern for a `BufferedReader`: repeatedly calling its `readLine` method while it remains `ready`. The variable names `br` and `f` were selected from among the most common human choices for `BufferedReader` and `FileReader`, and were synthesized together here: no single usage example need exist that uses both of those names in tandem. In addition, the example also demonstrates the importance of control flow: `readLine` is called repeatedly, but only after checking `ready`. Finally, the `//initialized previously` and `//do something with line` comments indicate points where different human developers would write different code and highlight the most direct places for a developer to adapt this code example into an existing setting.

In practice, there is more than one way to use a `BufferedReader`. Our algorithm can produce a ranked list of examples based on clusters of representative human usages. The second example we produce is:

```
1  InputStreamReader i; //initialized previously
2  BufferedReader reader = new BufferedReader(i);
3  String s;
4  while ((s = reader.readLine()) != null)
5      //do something with s
6  }
7  reader.close();
```

This second example shows that other concrete argument types can be used to create `BufferedReader`s (e.g., `InputStreamReaders` can be used as well as `FileReaders`). In addition, it shows that there is a different usage pattern that involves always calling `readLine` but then checking the return value against `null` (rather than calling `ready`). Both of the examples produced by our algorithm are well-formed and introduce commonly-named, well-typed temporaries for function arguments and return values.

It is also possible to use code search and slicing techniques to produce API examples. The tool of Kim *et al.* [58] produces the following output on the same `BufferedReader` query:

```

1 public static void main( String args[] ) throws Throwable{
2     BufferedReader br = new BufferedReader(
3         new InputStreamReader( System.in ));
4     String acl_in = br.readLine();
5
6     System.err.print( "Parsing input..." );
7     ACL parsed = parse( acl_in.getBytes() );
8
9     System.err.println( "OK\n\n" );
10    System.err.println( parsed );
11
12    System.err.print( "\nDeparsing, reparsing and deparsing..." );
13    String out = new String(deparsing(parse(deparsing(parsed))));
14
15    System.err.println( "OK\n\n" );
16
17    System.out.println( out );
18 }
```

Despite some use of slicing, the example is large and includes some irrelevant details (e.g., all but the first six lines). It does not typecheck without additional information (e.g., the `deparsing` and `parse` methods are specific to the example code from which the example extracted). Insufficient abstraction is employed: to a new user, it may be difficult to tell if the `System.in` argument to `InputStreamReader` is necessary in this case or coincidental (cf. the `//initialized previously` or `"foo.in"` of previous examples). Even the variable names are too specific to the example: `String acl_in = br.readLine()` is less descriptive than `String line = br.readLine()` for the general case.

We seek to produce this sort of usage example documentation automatically. To do so we must first understand how humans write usage examples and what they look for in usage documentation.

6.3 Human-Written Usage Examples

In this section we present a study of human-written API examples. The purpose of this study is to establish key guidelines for automatically creating examples that share the best properties of human-written ones. We seek to determine the desired size and readability of examples, and the importance of generality and

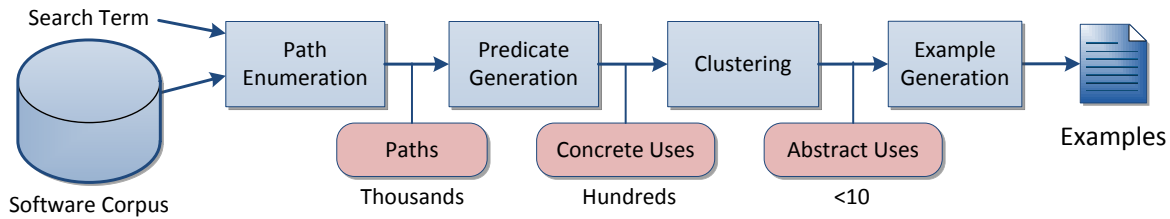


Figure 6.1: High-level view of the architecture of our algorithm.

correctness. We answer such questions by analyzing examples found in the standard Java API docs and through the results of a brief survey on the importance of various characteristics of examples.

6.3.1 Properties of Human-Written Examples

We explore the Java Software Development Kit (SDK) documentation because it is authoritative, generally considered to be of high quality, and is written for a general audience of Java developers. Furthermore, each example is tied to a specific class (and not, for example, to a coding task or other request).

We identified examples in the Java SDK by searching for pre-formatted text (e.g., contained in `<pre>` or `<code>` tags) and found 234 instances. We then manually inspected each one, selecting only those consisting of Java statements (e.g., ignoring lists of methods and other text). In all, we found 47 classes (3% of the total) which contain a usage example.

Length. We first consider the sizes of examples measured in lines of code. The full distribution of example sizes is presented in Figure 6.2. On average, human-written examples were 11 lines long, but the median size was 5 lines. While a significant number of examples are quite long, we note that human-written examples are typically very concise. We hypothesize that automated examples should be of a similar length to agree with human preferences and make a suitable supplement for existing examples.

Abstract initialization. We also note that many human-written examples use special markers, such as ellipses, to indicate that an input variable should be assigned a context-specific value. For example, the `glyphIndex` variable in this documentation for `java.awt.font.GlyphMetrics`:

```

1  int glyphIndex = ...;
2  GlyphMetrics metrics =
3      GlyphVector.getGlyphMetrics(glyphIndex);
4  int isStandard = metrics.isStandard();
5  float glyphAdvance = metrics.getAdvance();

```

Similarly, this example from `java.text.MessageFormat` uses variables initialized with 7 and "a disturbance in the Force", which are clearly chosen arbitrarily.

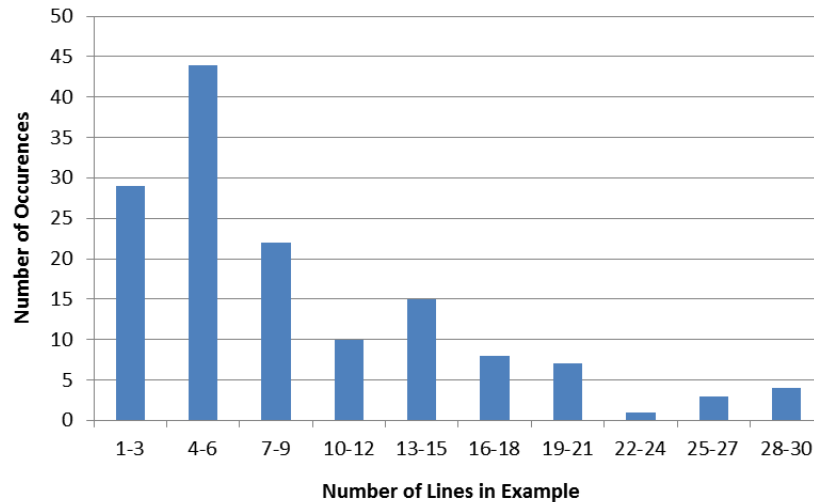


Figure 6.2: Histogram of the sizes (in lines) of usage examples embedded in the standard Java API documents.

```

1  int planet = 7;
2  String event = "a disturbance in the Force";
3  String result = MessageFormat.format(
4      "At {1,time} on {1,date}, there was {2}" +
5      "on planet {0,number,integer}.",
6      planet, new Date(), event);

```

We hypothesize that automatically-constructed documentation should also use such abstract initialization to highlight “inputs” to the examples.

We hypothesize that generated documentation should employ common concrete type and variable names where applicable, but only when they represent truly frequent usage patterns. The algorithm presented in this chapter annotates all variables which should be initialized with additional context with an `//initialized previously` comment.

Abstract use. Many examples contained an abstract placeholder indicating where the user should insert context-specific usage code. For example, the documentation for `java.text.CharacterIterator` makes it clear that the user should “do something with” or otherwise process the character `c` inside the loop:

```

1  for(char c = iter.first(); c != CharacterIterator.DONE; c = iter.next())
2  {
3      processChar(c);
4  }

```

We hypothesize that synthesized examples must similarly and concisely indicate where context-specific user code should be placed and what variables it should manipulate. The algorithm presented in this chapter employs `//do something with X` annotations in such cases.

Exception Handling. 16 of the 47 examples contained some exception handling. Reasoning about programs that use exceptions is difficult for humans and also for automatic tools and analyses (e.g., [134, 135, 138]). Often exceptions are caught trivially (i.e., no action is taken to resolve the underlying error [131]) or the mechanism is purposely circumvented [132, 133]. This example for `java.nio. file.FileVisitor` is indicative:

```
1  try {
2      file.delete();
3  } catch (IOException exc) {
4      // failed to delete, do error handling here
5  }
6  return FileVisitResult.CONTINUE;
```

We hypothesize that tool-generated documentation should mention exception handling, but only when it is common or necessary for correctness. The algorithm presented in this chapter learns common exception handling patterns in the corpus and distills examples representative of exception handling practice.

6.3.2 Survey Results

As part of the evaluation of the tool presented in this chapter, we conducted a human study involving over 150 participants, primarily undergraduates at *The University of Virginia* enrolled in a class entitled *Software Development Methods* which places a heavy focus on learning the Java language and its standard set of APIs. Details about the study can be found in Section 6.7.1. As part of the study, humans were asked “What factors are important in a good example?” Some common themes emerged:

Multiple uses. Users wanted examples of different ways to use the class: “It shouldn’t model something extremely specific, it should include something that the class will be most commonly used for.” Documentation “must be able to show multiple uses. If you just show one example of a possible use, it probably won’t be the one that interests me.” We should “show examples of different ways a class can be used.” The best documentation shows “all the different ways to use something, so it’s helpful in all cases.” Our algorithm uses clustering to capture multiple uses.

Readability. Users wanted examples that were easy to read and understand: “a good example is easy to understand and read.” Many were explicit: “readable and understandable are the most important aspects.” Slicing away unrelated code was critical: “less irrelevant, unrelated stuff in the example is better.” A key component of readability is conciseness: the example should use “as little code as possible” and show “the most basic version of the problem. You can have additional more complicated problems if necessary.” In Section 6.6.2 we empirically demonstrate that our algorithm produces readable examples.

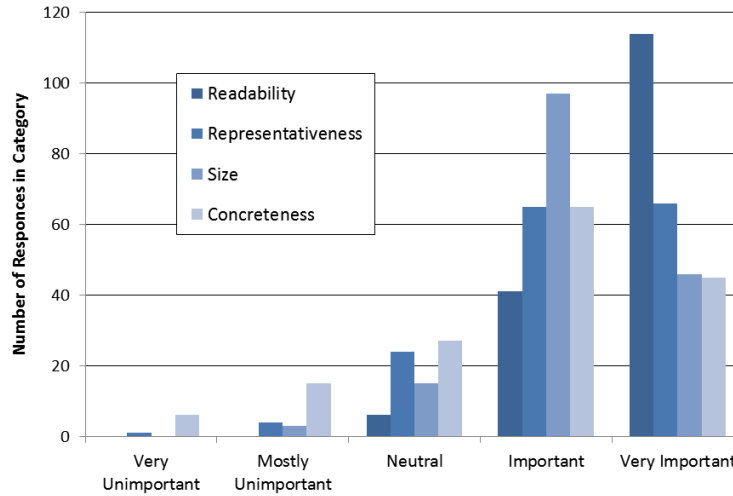


Figure 6.3: Human survey responses about the importance of various features of example documentation. Readability was the most important feature.

Naming. One key aspect of readability was the choice of identifier names: “they should be simple and understandable.” Users preferred “logical variable names”, “declaring variable names to represent what they do/are”, “clear naming of variables”, and variables demonstrating “standard naming”. The importance of identifier names has been previously studied (e.g., [30]). Our algorithm tracks common variable naming information from concrete source code to produce understandable variable names.

Variables. Users also prefer documentation that includes intermediate variables and temporaries: “showing declarations” and “the use of many temporary variables helps” to make good documentation. The human-written documentation in Section 6.2 elides temporary variables; by contrast our algorithm produces well-typed, clearly-named temporaries.

The study also included a set of Likert-scale questions concerning the importance of several aspect of usage examples. The properties were *size* (“How concise is the example?”), *readability* (“How easy is it to understand?”), *representativeness* (“Will it be useful for what I want to do?”), and *concreteness* (“Can I compile and use it?”).

Figure 6.3 shows the results. Notably, readability was the most important of the four features, with almost every participant ranking it as “very important” or “important.” Note that readability is judged even more important than representativeness, which as been traditionally considered most important (e.g., [58]) — Representativeness and size were the next most important, and concreteness was least important among the four. However, all four features were at least “important” on average, and must thus inform the design of our automatic example documentation algorithm.

6.4 Algorithm Description

This section describes our documentation-generation algorithm. In general, the design decisions in the algorithm were made to produce documentation with the qualities found in human-written examples (Section 6.3.1) and the qualities praised by humans in our survey (Section 6.3.2). We first summarize the overall algorithm, and then describe each major step in detail. We take as input a target class to document and a corpus of client code. Figure 6.1 shows the flow of steps in our algorithm, which we also describe below:

1. Our first step is *path enumeration* (Section 6.4.1), in which we statically enumerate intra-procedural traces in corpus which invoke a method on the target class.
2. The resulting acyclic program paths are subjected to a data-flow analysis and path *predicate generation* (Section 6.4.2). This step merges program paths into a smaller number of *concrete uses*. Conceptually, each concrete use corresponds a single (static) instantiation of the target type. These uses are represented as annotated finite state machines.
3. The third step applies a *clustering* algorithm to identify groups of concrete uses that are similar (Section 6.4.3). These clusters are then merged into a small set of *abstract uses* which intuitively correspond to the ways the class is used in the corpus. The abstract uses are also represented as finite state machines but with edges weighted by concrete use counts.
4. Finally, we sort the abstract uses by their representativeness, flatten the state machines into a “best method ordering” and distill them into human-readable documentation (Section 6.4.4), using naming information from Step 2 and introducing well-typed temporaries.

6.4.1 Path Enumeration

For efficiency, we first filter the entire corpus, selecting only files that include a reference to the target class. We then process each method in each of the remaining classes of the corpus in turn. Note, however, that example generation does not require complete coverage of a corpus. Once a sufficiently large number of examples are found such that the model becomes stable (i.e., additional examples have no appreciable effect), the process can be allowed to terminate.

To process a method, we first enumerate its loop-free control flow paths: our analysis is thus path-sensitive and potentially exponential. We obtain loop-free paths by adding a statement to a path at most once: in effect, we consider each loop to either be taken once, or not at all. This decision can occasionally result in imprecise or incorrect documentation, however we see in Section 6.7 that this occurs infrequently in practice: the ultimate goal is to generate documentation that will be consumed by humans.

6.4.2 Predicate Generation

The path traces are sliced [149] to contain only events related to the class to be documented. That is, while many programs may use `sockets`, we wish to abstract away program-specific details (e.g., the data sent over the `socket`) while retaining API-relevant details (e.g., `open` before `read` or `write` before `close`). Our algorithm retains and considers statements containing any of the following elements:

1. Any method invocation or new object allocation which returns an instance of the target class or one of its subclasses.
2. Any static invocation on the target class.
3. Any method invocation on an object from (1).
4. Any statement using (e.g., as a function argument or subexpression) a return value from (1), (2), or (3).

We then compute intraprocedural *path predicates*, logical formulae that describe conditions under which each statement can be reached [96, 148, 149, 150]. To obtain these formulae, each control flow path is symbolically executed (e.g., [17]). We track conditional statements (e.g., `if` and `while`) and evaluate their guarding predicates using the current symbolic values for variables. We collect the resulting predicates; in conjunction, they form the path predicate for a given statement in the method.

From this information we form *concrete uses* which capture method ordering in a finite state automaton where edges express the *happens before* relationship (cf. many specification mining techniques [55, 179, 56, 180] where the edges represent method calls and an accepting sequence corresponds to a valid ordering). Concrete uses represent possible transitions between the statements linked to a single static allocation site (i.e., from (1) above) or to a specific class field. Merging on class fields is of particular importance for Java, where method invocations on the same dynamically-allocated object often occur in multiple procedures. For example, a `HashTable` may be allocated in a constructor, populated in a second method and queried in a third. Although no *happens before* relation can be established between statements in separate methods in a purely intra-procedural analysis (except that constructors must always be called first), this permits the ordering between invocations located in the same method to be preserved. This is critical to retaining as much ordering information as possible and eventually emitting the most representative documentation.

Nodes retain dataflow information, predicates, and identifier names. For example, a statement from a use of `BufferedReader` (e.g., as in Section 6.2) could be represented as:

$$\left\{ \begin{array}{l} stmt : \text{println}([result \text{ of } readLine()]) \\ pred : [result \text{ of } readLine()] == \text{null} \\ ident : [result \text{ of } readLine()] = \text{String line} \end{array} \right\}$$

6.4.3 Clustering and Abstraction

Even for a single target class, there are often many representative usage patterns to be found within a software corpus. We desire a ranked list such patterns. We thus cluster the concrete uses from the previous step into abstract uses: sequences of statements, including information about path predicates, types, and names. Abstract use categories can then be counted and compared for representativeness.

Intuitively, two concrete uses are similar and should be viewed as examples of the same abstract use pattern if they perform the same operations on the same data-types in the same order. We thus propose a formal *distance metric* between concrete uses which captures both statement ordering from the underlying state machines and type information. Our distance metric is similar to Kendall's τ [82], and captures both the number of sorting operations and number of type substitutions needed to transform one concrete use into another (see below). Our distance function is parametrized by α and β which express the relative importance of method ordering and type information. Our experience generating documentation for popular Java classes suggests that output quality is not highly sensitive to choice of α and β . However, in languages other than Java, particularly those with either weak or dynamic type systems, selection of these parameters could be important. For all experiments presented in this chapter, we set $\alpha = \beta = 1$, giving types and ordering equal weight.

In the following explanation, C_i is a concrete use: a set of statements admitting a *happens before* relation (see Section 6.4.2). The selector function K determines if two concrete instances include two statements in the same order. The selector function T determines if two concrete instances use the same types in all relevant subexpressions of two statements.

$$\begin{aligned}
 \text{Dist}(C_1, C_2) &= \sum_{\{i,j\} \in (C_1 \cup C_2)} \alpha K_{i,j}(C_1, C_2) + \beta T_{i,j}(C_1, C_2) \\
 K_{i,j}(C_1, C_2) &= \begin{cases} 1 & \text{if } i \text{ and } j \text{ are } \textit{not} \text{ in the} \\ & \text{same order in } C_1 \text{ and } C_2 \\ 0 & \text{otherwise} \end{cases} \\
 T_{i,j}(C_1, C_2) &= \begin{cases} 1 & \text{if } i = j \text{ and } i \text{ and } j \text{ use} \\ & \text{different types in } C_1 \text{ and } C_2 \\ 0 & \text{otherwise} \end{cases}
 \end{aligned}$$

Given this distance metric, we use the k-medoids algorithm [181] to cluster concrete uses. Because our distance metric must necessarily be computed between concrete objects, traditional k-means cannot be used: the k-means algorithm requires that distance can be computed between objects and also between arbitrary points in the metric space.

K-medoids takes a parameter: k , the number of clusters to find. k represents the inherent trade-off between precision and generality that exists with any documentation system. In our setting, a large value of

k results in the algorithm returning multiple examples of the same basic usage pattern (e.g., the two uses of `BufferedReader` in Section 6.2). If k is too small, however, it is possible that multiple usage patterns could be conflated. An overly large k can result in less representative examples, especially in terms of identifier names. All of the experiments in this chapter use $k = 5$, since our initial study (see Section 6.3.1) found no classes with more than four common patterns.

While previous approaches have applied clustering to code search (e.g., [58, 59]), to the best of our knowledge our approach is the first to leverage type information and statement ordering. We believe that these properties are essential to documenting many usage patterns (e.g., the best choice of data structure often depends on the usage pattern).

After clusters are discovered, the associated concrete uses are merged into *abstract uses* which are also represented as finite state machines. To merge a set of concrete uses, we first union all of the states (i.e., nodes) contained in each concrete use. The merging step is conceptually akin to the least-upper-bound lattice computation in constant-folding dataflow analysis: $5 \sqcup 5 = 5$, but $x \sqcup y = \text{SOMETHING}$. For example:

$$A = \left\{ \begin{array}{l} \text{stmt} : \text{print}(\mathbf{x}) \\ \text{pred} : \mathbf{x}.\text{isTrue}() \end{array} \right\}$$

$$B = \left\{ \begin{array}{l} \text{stmt} : \text{return } \mathbf{x} \\ \text{pred} : \mathbf{x}.\text{isTrue}() \end{array} \right\}$$

$$A \sqcup B = \left\{ \begin{array}{l} \text{stmt} : \text{Do something with } \mathbf{x} \\ \text{pred} : \mathbf{x}.\text{isTrue}() \end{array} \right\}$$

Transitions (i.e., edges) between statements in the abstract use are then constructed and weighted according to the number of concrete statements which include the transition. Information about the types and names of parameters and return values is retained.

6.4.4 Emitting Documentation

The final stage of our algorithm transforms each abstract use into a representative, well-formed, and well-typed Java code fragment. Recall that abstract uses are finite state machines with edge weights that correspond to usage counts in the corpus (e.g., an edge with weight 15 connecting S and T indicates that there were 15 concrete uses which observed the ordering S happens before T).

An abstract use can contain both branches and cycles. Usage documentation, however, must be presented as linear text. Our next step is thus to find a representative ordering of the statements (i.e., a partial

serialization) in each abstract use. We choose to approach this problem using a weighted topological sort. The goal is to find an ordering of state machine nodes (from start node to end node) which maximizes the sum of the weight of all outgoing edges minus the sum of the weight of all incoming edges. Such an ordering is optimal in the sense that any other ordering will be less representative of the concrete orderings in the corpus (i.e., more total statements must be swapped to align all of the concrete orderings with this this ordering).

More formally, we take as input set of statements S and function W , representing edge weights, which maps all pairs of statements in S to a value greater than or equal to 0. The task is to find a total ordering O , defined here as a function mapping a pair of unique statements (s_i, s_j) to 1 if and only if s_i comes before s_j in O and to -1 otherwise. The notion of representativeness can then be formalized by the *Flow* objective function below.

Input: Statement set $S : \{s_1, s_2, \dots s_n\}$

Input: Edge weight mapping $W : (s_i, s_j) \rightarrow \{0 \dots \infty\}$

Output: Ordering $O : (s_i, s_j) \rightarrow \{-1, 1\}$ maximizing:

$$Flow = \sum_{\{s_i, s_j\} \in S} O(s_i, s_j) \cdot W(s_i, s_j)$$

Our algorithm finds such an ordering by enumerating all possible orderings and computing *Flow*, returning the ordering for which *Flow* is largest. Note that the time complexity of this algorithm is factorial in the number of statements to be documented. However, this remains tractable for $n < 12$ and we have not encountered instances with $n > 8$. We implemented an unoptimized version of the algorithm and found that it was not a performance bottleneck in practice (see Section 6.6).

Once a statement ordering has been determined, the next step is to generate code (i.e., usage example documentation) from those ordered sets of statements. We desire code that is representative of the underlying concrete statements which were clustered into this abstract use. We first generate a basic block for each statement, guarded by its most common predicate (taken over the concrete uses forming this abstract use cluster). Statements are printed using standard Java syntax; names and types are assigned to all return values and parameters as needed. Adjacent blocks with the same predicate are merged (e.g., “if (p) X; if (p) Y;” becomes “if (p) {X; Y;}”).

When choosing a name, we first check if any one name is used $X\%$ more often than all other names by human-written code in the corpus. In our experiments we set X to 100%, thus selecting a name if it occurs twice as often as any other name. If no such popular name exists, we use the declared formal parameter name for actual arguments and the first letter of the type name for other variables. For statements that are not invocations of the target class, we print “//do something with Y” where Y is the appropriate

identifier. Types are chosen in the same manner, making use of declared return types for function return values. Declarations are inserted and marked with “`//initialized previously`” for all otherwise-undefined (“temporary”) variables.

For each statement, we also count how often it was found inside a `try`, `catch`, or `finally` clause, and the type of the associated exception. If a statement is part of exception handling more often than not in the corpus, we then impose similar error handling in the generated example. In Java, well-formed exception handling involves a `try` block followed immediately by zero or more `catch` clauses and possibly a `finally` clause, with at least one `catch` or `finally` clause. All exception handling generated by our algorithm is of this form; we will not to generate a `catch` clause if there is no preceding `try`.

By constructing examples in this way, we ensure that no Java syntax rules are violated. Furthermore, we guarantee that all assignments are made to an identifier of the proper type and that all methods are invoked with the right number and types of arguments. It is important to note, however, that we cannot be sure that generated examples are free of other types of errors (e.g., run-time errors, unchecked exceptions, etc.).

The representativeness of our examples is well defined: if the order of any two statements were inverted (or if some statement were added or removed), then the resulting example would correspond to fewer real-word examples from the corpus. We are not aware of any previous approach to documenting APIs which offers such guarantees. Other correctness properties are, to some degree, orthogonal. For example, we could compile our examples and run them to check for run-time exceptions, or perhaps use model checking to verify temporal safety properties. While the topics are related, we are not proposing a static specification mining technique (where correctness is paramount); rather, we are proposing a common-usage documentation synthesis technique which could leverage work in specification mining. We view common behavior in the corpus as correct by definition for this task.

6.4.5 Implementation Details

In creating a prototype implementation of the algorithm described in this section we perform source-code and byte-code analysis side-by-side. We use the SOOT [128] toolkit to analyze byte-code and perform path extraction and symbolic execution. A source code analysis is necessary to record certain identifier names; it also simplifies the process of recording exception handling structures that are more difficult to recover from byte-code.

Name	Version	Domain	kLOC
FindBugs	1.2.1	Code Analysis	154
FreeCol	0.7.2	Game	91
hsqldb	1.8.0	Database	128
iText	2.0.8	PDF utility	145
jEdit	4.2	Word Processing	123
jFreeChart	1.0.6	Data Presenting	170
tvBrowser	2.5.3	TV Guide	138
ca Weka	3.5.6	Machine Learning	402
XMLUnit	1.1	Unit Testing	10
total			1361

Table 6.1: Corpus for example generation in this study.

6.5 Indicative Examples

In this section we present a number of examples indicative of the strengths and limits of our approach. We constructed these examples using the program corpus enumerated in Table 6.1, which includes nine popular open-source Java programs, as input to our prototype example generation tool. We issued example queries for each of the 47 SDK classes for which human-written example documentation was available (see Section 6.3.1 for details). Thirty-five of the classes were used at least once in our corpus and thus produced example documentation. After examining these examples in detail we perform to a formal empirical evaluation and human study in Section 6.6 and Section 6.7.

6.5.1 Naming

The names of types and variables are critical to producing readable, representative examples. The output of our tool can be sensitive to the types and variables used in the available corpus. Consider the following example for `java.util.Iterator`, which demonstrates how an `Iterator` can be used to enumerate arbitrary objects.

```

1 Iterator iter = SOMETHING.iterator();
2 while(iter.hasNext()) {
3     Object o = iter.next();
4     //do something with o
5 }
```

The same query, conducted using only the FREECOL benchmark, results in a very different example. Because use of the `EventIterator` class, which is a subclass of `java.util.Iterator`, is common in FREECOL, it is chosen as the most representative example.

```

1 EventIterator eventIterator =
2     SOMETHING.eventIterator();
```

```

3  if(eventIterator.hasNext()) {
4      Event e = eventIterator.nextEvent();
5      //do something with e
6  }

```

We hypothesize that both behaviors are useful: the former for constructing generic library API documentation, and the later for crafting company-specific documentation for internal APIs.

6.5.2 Patterns

Our algorithm finds and preserves common usage patterns: frequently occurring sequences of statements. However, there are many classes, such as the generic `Throwable`, for which there is no truly common pattern. The example our algorithms generates for `Throwable` shows that an instance might be queried for its *Class*, *Cause*, or *StackTrace*.

```

1  PrintWriter p; //initialized previously
2  Throwable e = SOMETHING.getThrown();
3  e.getClass();
4  e.getCause();
5  e.printStackTrace(p);

```

For classes such as this, more traditional API documentation which lists all the methods of a class and their functionality is clearly important. Furthermore, there are some patterns that cannot be captured by our algorithm. Interactions such as message passing patterns between multiple threads or client-server systems could not be discovered without additional analyses.

6.5.3 Exceptions

Ideally, examples should contain complete and correct exception handling. However, our tool learns actual exception handling from real code, which may not be fully correct. The example below of `java.io.ObjectInputStream` is one of a small number of classes for which our top-ranked example uses exception handling.

```

1  BufferedInputStream b; //initialized previously
2  ObjectInputStream stream =
3      new ObjectInputStream(b);
4  try {
5      Object o = stream.readObject();
6      //Do something with o
7  } catch(IOException e) {
8  } finally {
9      stream.close();
10 }

```

We view the correct use of exceptions and the correct handling of resources in the presence of exceptions as an orthogonal problem [132, 133, 131].

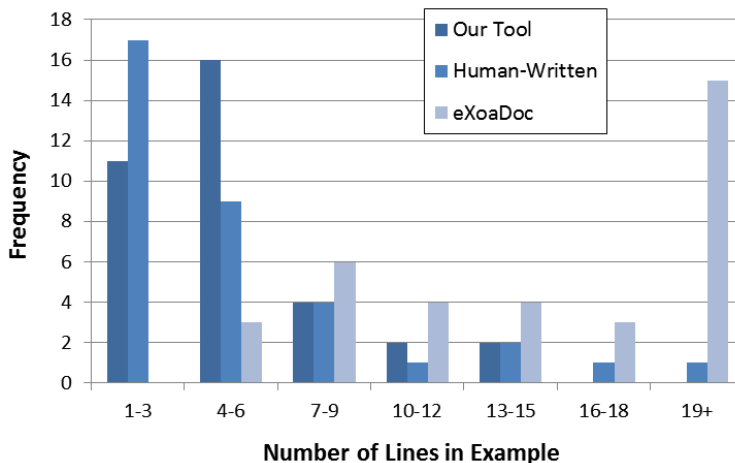


Figure 6.4: Size comparison (in lines of code) of examples generated by our tool using the corpus from Table 6.1, written by humans, and mined by EXOADOC. The dataset is described in Section 6.6.

6.6 Empirical Evaluation

In this section we begin the evaluation of our proposed algorithm and prototype implementation with two quantitative metrics: size and readability. Both of these features are considered very important by users of documentation (see Section 6.3.2). Running our prototype tool on 1,361k lines of code to produce example documentation for 35 classes took 73 minutes (about 2 minutes per class). About 95% of this time is spent filtering the corpus and running path enumeration; because much of this work could be precomputed and/or parallelized, we conjecture that it is possible to generate examples in matter of seconds given a fixed corpus.

Throughout our evaluation we compare the output of our tool to both human-written examples from the Java SDK and also the EXOADOC tool of Kim *et al.* [58]. EXOADOC works by leveraging an existing code search engine to find examples of a class. Kim *et al.* then employ slicing to extract “semantically relevant” lines. These examples are then clustered and ranked based on *Representativeness*, *Conciseness* and *Correctness* properties. EXOADOC has been shown to increase productivity by as much as 67% in a small study.

Our dataset consists of examples for 35 classes: those classes from the standard Java APIs for which we have one example of each of the three example types (see Section 6.5). Because EXOADOCs are associated with methods rather than classes, we chose the top example for the most popular method (by static count of concrete uses in our benchmark set). For our tool, we chose the top (i.e., most representative) example for each class.

6.6.1 Size Evaluation

We compared the size (in lines of code) of each documentation type. The results are presented in Figure 6.4. Examples produced by our tool are slightly longer on average as compared to human written examples, but they follow approximately the same long-tail distribution. Much of the size difference can be attributed to our use of separate lines for “previously initialized” variables, which are often declared in-line by humans (see example in Section 6.2).

Nonetheless, both human-written examples and our generated examples are shorter than EXODOC examples, which are often more than 19 lines long. This difference is largely due to the number of less-relevant lines present in such mined examples.

6.6.2 Readability Evaluation

Because readability was considered the most important characteristic of examples, we chose to evaluate the readability of the three documentation types directly. For this purpose we used the readability metric described in Chapter 2. The metric reports the readability of a Java code snippet on a scale of 0 (least readable) to 1 (most).

Figure 6.5 presents the results of that analysis. Overall, examples generated by our tool are about 25% more readable than human-written examples and over 50% more readable than examples mined with EXODOC (t-test significance level < 0.002 in both cases). This difference is consistent across the major Java package areas to which the documented classes belong. Additionally, generated documentation is more consistently readable: the standard deviation in the readability score of our tool output is about 40% lower than that of the other two documentation types.

6.7 Human Study

In this section we present a human study of API example quality. The goal of this study is to quantify the desirability of the output of our tool in comparison to both human-written examples (from the Java SDK) and the state-of-the-art tool EXODOC of Kim *et al.* [58]. The study involved 154 participants evaluating 35 pairs of API examples.

6.7.1 Experimental Setup

For each of the 35 classes from the dataset described in Section 6.6, the participant is shown the name of the target class and is randomly shown two of the three documentation types (i.e., ours, EXODOC, and

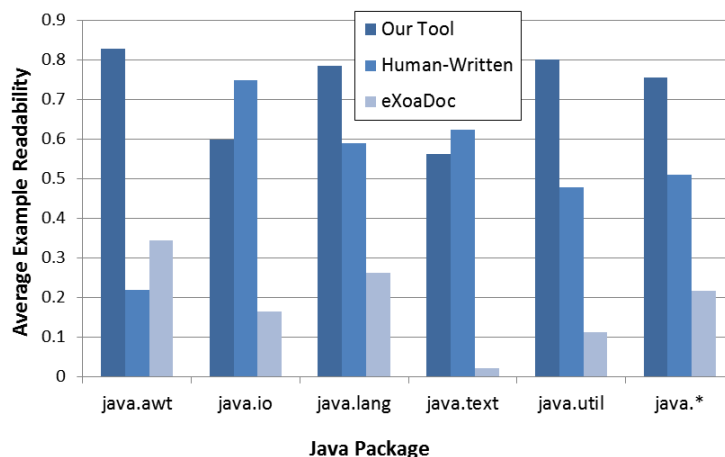


Figure 6.5: Readability comparison of examples generated by our tool, written by humans, and mined by EXOADOC. The scale ranges from 1.0 (most readable) to 0.0 (least). The dataset, which we breakdown into 5 Java packages, is described in Section 6.6.

human-written). The participant is then required to make a preference evaluation by selecting one option from a five-element Likert scale: “Strong Preference” for A, “Some Preference” for A, “Neutral”, “Some Preference” for B, “Strong Preference” for B. If desired, the participant may also choose to “Skip” the pair. An annotated interface screenshot is presented in Figure A.4.

The study was advertised to students at *The University of Virginia* enrolled in a class entitled *Software Development Methods*, which places a heavy focus on learning the Java language and its standard set of APIs. For comparison, 16 computer science graduate students also participated. 179 students participated in total, however, to help preserve data integrity, we removed from consideration the results from 25 undergraduate students who completed the study in less than five minutes. The average time to complete the study for the remaining 154 participants was 13 minutes.

Participation was voluntary and anonymous. The participants were instructed to “Pretend that [they] are a programmer or developer who needs to use, or understand how to use, the class.” No additional guidance was given; participants formed their own opinions about each example.

6.7.2 Quantitative Results

In total, 154 participants compared 35 example pairs each, producing 5,390 distinct judgments. The aggregate results are presented in Figure 6.6. Overall, the output of our tool was judged at least as good as human written examples over 60% of the time and strictly better than EXOADOC in about 75% of cases. For 82% of examples, on average either humans preferred our generated documentation to human-written examples or had no preference. For 94% of examples, the output of our tool was preferred to EXOADOCs.

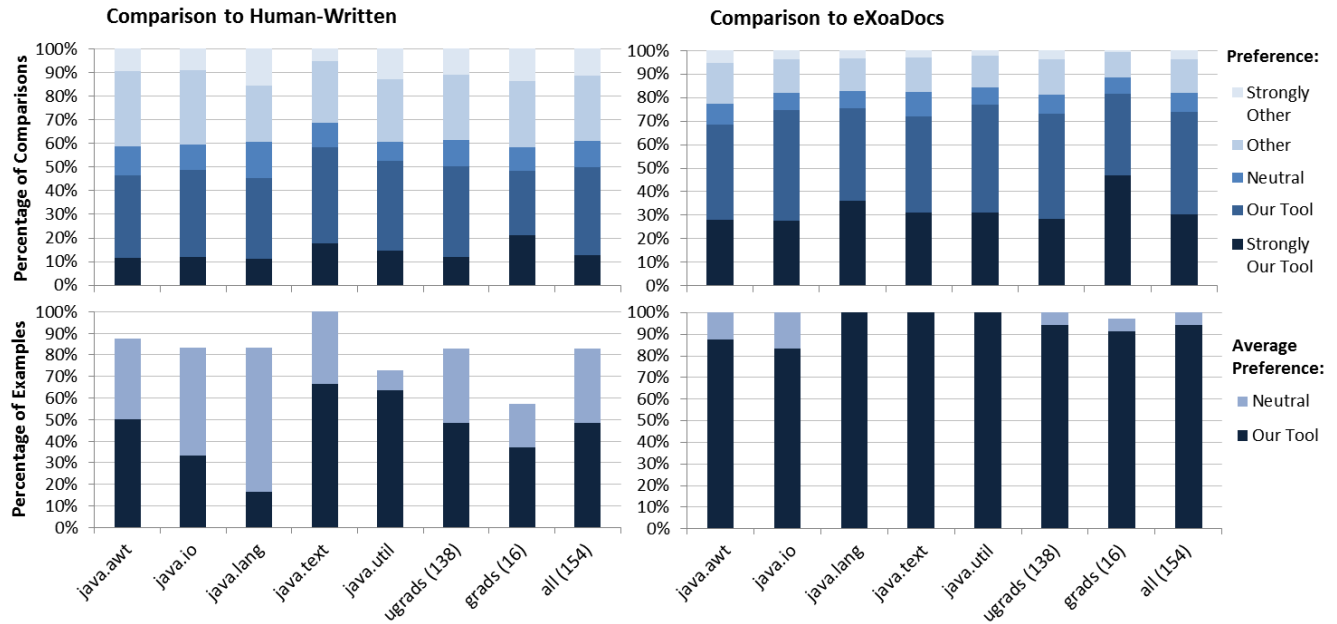


Figure 6.6: Aggregated results from our human study. The charts on the left shows responses when our tool was compared to human-written examples and the charts on the right compares our tool to the EXOADOc tool of Kim *et al.* [58]. The charts on the top categorize all comparisons (e.g., in 60% of comparisons, our tool output was judged at least as good as human-written examples). The charts on the bottom reflect the consensus opinion for each example (e.g., 82% of examples produced by our tool were judged to be at least as good as human-written ones on average). Examples are broken down by package (e.g., `java.X`) and by category of participants (138 undergrads, 16 grads).

Comparing graduate students to undergrads, we found that their raw score distributions were very similar. However, taken example by example, grad students had a 20% reduced preference for tool-generated examples when compared to human-written examples. Nonetheless, both grad students and undergrads judged our generated documentation to be at least as good as gold-standard human-written for over half of the examples. Compared to EXOADOc, our tool was preferred in almost all cases by both groups.

Finally, we asked participants whether “a program that automatically generates examples like these would be useful?”, and 81% agreed that such a tool would be either “Useful” or “Very Useful.”

6.8 Threats To Validity

Although our experiments suggest that our algorithm produces examples that are preferred to state-of-the-art code search techniques and as good as human-written documentation over 80% of the time, our results may not generalize.

First, the benchmarks we selected may not be indicative. We mitigated this threat by performing our human evaluation on documentation for standard library classes — classes used in almost every program and

by almost every programmer. It is possible, however, that results for these classes may not generalize to all APIs. However, the documented classes did feature a wide range of usage patterns. In addition, the output of our tool was shown to be consistently good over all major packages in the Java SDK.

Our use of students for evaluation may not generalize across all populations and to professional developers in particular. This threat is somewhat mitigated by the observation that our graduate student population agreed quite closely with our undergraduate one despite the fact that the graduates had significantly varying backgrounds and typically had more years of programming experience.

6.9 Related Work

The most closely related work is that of Kim *et al.* [58] and Zhong *et al.* [59]. We compare to Kim *et al.* directly and have discussed Zhong *et al.* previously. We now briefly describe other works related to mining API usage patterns and their documentation.

Our technique is inspired in part by the work of Whaley *et al.* who, in 2002, used multiple finite state machine submodels to model the interface of a class. They used dynamic instrumentation techniques to extract such models from execution runs. Whaley *et al.* note that these models can potentially serve as documentation, however they do not evaluate the efficacy of such an approach.

Wasykowski *et al.* [182] adapted static specification mining techniques to learn common, but not required, method-call sequences. Like our technique, they use finite state machines to represent common usage patterns for an object. Nguyen *et al.* [183] employ a similar approach to mine usage patterns, but across multiple objects. Both approaches focus on defect and anomaly detection. By contrast, our work focuses on generating and evaluating human-readable documentation, assuming that average behavior in the input corpus is correct (also a standard assumption for specification mining) and validating the final result with a human study.

Dekel *et al.* [184] describe a technique for decorating method invocations with rules or caveats of which client authors must be aware. The technique is designed to increase awareness of important documentation written by humans. Holmes *et al.* [176] use structural context to recommend source code examples. Their approach finds relevant code in an example repository that is based on heuristically matching the structure of the code under development. We conjecture that both of these tools could make use of our machine-generated examples.

6.10 Conclusion

API examples are known to be a key learning resource, however they are expensive to create and are thus often unavailable when needed or out-of-date. Current tools for mining such examples tend to produce output that is complex and difficult to understand, compromising usefulness. These observations led us to propose an algorithm for *generating* API usage examples. Our technique is efficient and fully automated; to the best of our knowledge, it is the first approach that synthesizes *human-readable* documentation for APIs. Our output is correct-by-construction in several important respects (e.g., syntactically valid and type-safe) and adheres to a well-defined notion of representativeness: if the order of any two statement in one of our examples were inverted, the result would correspond to fewer real-world uses. In a human study involving over 150 participants, we have shown it to produce output judged at least as good as gold-standard human-written documentation 82% of the time and strictly better than a state-of-the-art code search tool 94% of the time.

Chapter 7

Conclusion

“There’s no right, there’s no wrong, there’s only popular opinion.”

– *Jeffrey Goines, Twelve Monkeys*

In this dissertation we explored whether program structure and behavior can be accurately modeled with semantically shallow features and documented automatically. Our stated goal was to produce a set of models and algorithms suitable for helping developers better understand programs. We began by exploring models for key aspects of program understanding: readability and runtime behavior — both based on surface-level features. Each model was shown to have high predictive power in multiple respects. We then described techniques for improving comprehension. In particular, we presented algorithms for documenting exceptions, program changes, and APIs; each was shown to produce human-competitive output.

In this concluding chapter we tabulate the main findings presented in this document (Section 7.1). In Section 7.2 we discuss broader impact goals for the presented work. Final remarks are presented in Section 7.3.

7.1 Summary of Results

Our five thrusts gave rise to twelve central research questions originally described in Chapter 1. Table 7.1 enumerates each question, the section of this document in which the question is addressed, and the key experiential result obtained.

7.2 Research Impact

We envision a series of models and tools that can be used to address many challenges associated with program understanding.

#	Research Question	Section	Key Result
1	To what extent do humans agree on code readability?	2.4.2	Spearman's $\rho = 0.55$ ($p < 0.01$).
2	Is the proposed readability model accurate?	2.4.2	Spearman's $\rho = 0.71$ ($p < 0.0001$).
3	Is the proposed readability model correlated with external notions of software quality?	2.5	Moderate f-measure correlation: FindBugs = 0.62, version changes = 0.63, churn = 0.65.
4	What static code features are predictive of path execution frequency?	3.8.1	see Figure 3.8.
5	Is the proposed path frequency model accurate?	3.6	5% hottest paths by our metric have 50% temporal coverage.
6	Is the proposed algorithm for exception documentation accurate?	4.6.1	In 85% of cases is at least as accurate as human documentation.
7	Does the proposed algorithm for exception documentation scale to large programs?	4.6	Algorithm terminates in 1,210 seconds on largest benchmark.
8	What qualities are necessary and sufficient for an effective commit message?	5.3	see Figure 5.2.
9	Is the output of the proposed algorithm for change documentation accurate?	5.5.2	By score metric DELTADOC could replace over 89% of WHAT documentation.
10	Is the output of the proposed algorithm for change documentation readable?	5.5.3	In 63% of cases the annotators had no preference or preferred DELTADOC.
11	What qualities are necessary and sufficient for an effective usage example?	6.3.2	see Figure 6.3.
12	Is the output of the proposed algorithm for usage examples correct and useful?	6.7	82% of examples were judged at least as good as human-written.

Table 7.1: Summary of Research Questions.

Evolution and Composition. Many software problems arise from misunderstandings rather than classic coding errors [185]. Vulnerabilities may be introduced when code is integrated with third-party libraries or “injected” as a program is changed or maintained. The Spring Framework, a popular J2EE web application framework with over five million downloads [186], was discovered in September 2008 to have two such security vulnerabilities allowing attackers to access arbitrary files on the affected webserver [187]. Dinis Cruz, who helped identify the vulnerabilities, notes that they are “not security flaws within the Framework, but are design issues that if not implemented properly expose business critical applications to malicious attacks” [187]. Specifically, the vulnerability occurs when developers mistakenly assume that certain types of input fields are properly validated within the framework when they are not. Cruz notes that “developers don’t fully understand what’s happening. They don’t see the side effects of what they’re doing. In a way the framework almost pushes you to implement it in an insecure way” [187]. Malicious adversaries might also insert trojans or backdoors into open-source software, and outsourced or offshored components may not meet local standards.

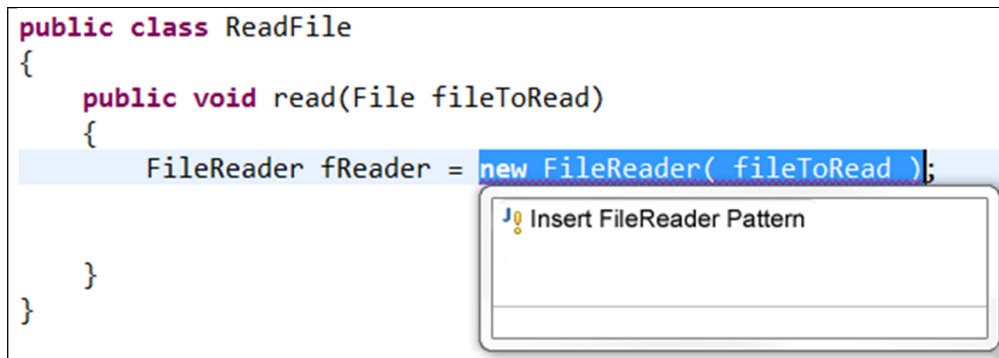


Figure 7.1: Mock-up of an API-usage auto-completion tool. In this case the user has declared and initialized a `FileReader` object. The tool responds by suggesting to complete the full pattern associated with this API.

Automatically generated, up-to-date documentation would make it easier for integrators to avoid such problems by understanding the code [188]. Formal readability models and static path frequency predictions could focus attention on harder-to-understand code and corner cases — areas more likely to contain defects [131].

Increasing Productivity. Many of the tools we presented hold out the promise of not just improving software quality, but also developer productivity. Our algorithms for documenting exceptions and program changes can save on time developers spend writing documentation. Our algorithm for API documentation can be employed to help developers quickly find examples. Furthermore, IDE integration may make in-line development assistance possible. We envision a tool that can automatically suggest usage patterns. The user could then choose to insert the code for a pattern with a single mouse click (e.g., see Figure 7.1).

Identifying Expectation Mismatch. While our metrics are designed to predict human judgments, in practice they will not always match expectations. Cases where the mismatch is the greatest may reveal code that should be scrutinized. A mismatch occurs when, for example, dynamic profiling reveals a path that was labeled uncommon by our path frequency model but proves to be very common in practice. Such a situation may indicate a performance bug, where usage deviates from developer expectations. Similarly, when usage of an API strays from common usage an error maybe be indicated. We conjecture that a IDE plugin or similar tool could help developers avoid mistakes when using APIs by displaying a warning when an API is used in an unusual way (for an example see Figure 7.2). Additionally, our readability metric can help developers identify code that may not be as readable to others as they thought.

Improving Existing Analyses. Recent work has demonstrated that our path frequency metric is useful for reducing false positives in the domain of specification mining [57], where it was three times as significant as other commonly-considered features (e.g., code churn, path density, presence of duplicate code) and helped to reduce false positives by an order of magnitude.

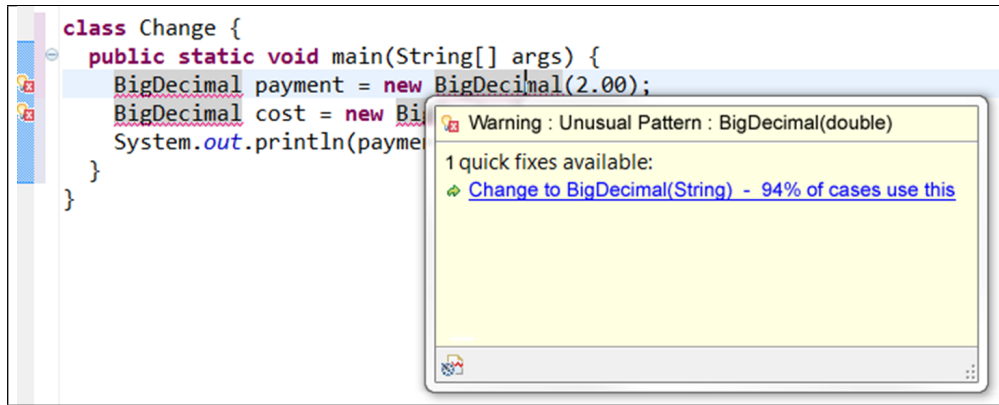


Figure 7.2: Mockup of API-usage uncommon pattern detection tool. In this case the user has employed the `BigDecimal(double)` constructor which is a common error. The tool suggests the `String` constructor which should be used in most cases.

Profile-guided optimization refers to the practice of optimizing a compiled binary subsequent to observing its runtime behavior on some workload (e.g., [118, 100, 99]). Our path frequency model has the potential to make such classes of optimization more accessible; first, by eliminating the need for workloads, and second by removing the time required to run them and record profile information. A static model of relative path frequency could help make profile-guided compiler optimizations more mainstream.

Finite maintenance resources inhibit developers from inspecting and potentially addressing every warning issued by existing software analyses [125] or flagged in bug tracking systems [189, 190]. An analysis that produces a bug report that is never fixed is not as useful as it could be. Existing analysis reports, such as backtraces, are not a panacea: “there is significant room for improving users’ experiences ... an error trace can be very lengthy and only indicates the symptom ... users may have to spend considerable time inspecting an error trace to understand the cause.” [191] Our metrics for readability and path frequency can help developers prioritize effort to bugs that are most likely to be encountered in practice. Furthermore, there has been a recent effort to create self-certifying alerts [192] and automatic patches [193, 194, 195, 196] to fix the flaws detected by static analyses. In mission-critical settings, such patches must still be manually inspected. Our documentation tools could make it easier to read and understand machine-generated patches, increasing the trust in them and reducing the time and effort required to verify and apply them.

Metrics. There are several popular readability metrics targeted at natural language [65, 66, 68, 67]. These metrics, while far from perfect, have become ubiquitous because they can very cheaply help organizations gain confidence that their documents meet goals for readability. Our software readability metric, backed with empirical evidence for effectiveness, can serve an analogous purpose in the software domain. For example, when constructing software from off-the-shelf components or subcontractors, the subcomponents might be

Chapter	Venue	Publication
2	ISSTA '08	<i>A Metric for Software Readability</i> [197]
2	TSE '10	<i>Learning a Metric for Code Readability</i> [198]
3	ICSE '09	<i>The Road Not Taken: Estimating Path Execution Frequency Statically</i> [199]
4	ISSTA '08	<i>Automatic Documentation Inference for Exceptions</i> [200]
5	ASE '10	<i>Automatically Documenting Program Changes</i> [201]
6	ICSE '12	<i>Synthesizing API Usage Examples</i> [202]

Table 7.2: Publications supporting this dissertation.

required to meet a overall code readability standards to help ensure proper integration and safe modification and evolution in the future, just as they are required to meet English readability standards for documentation.

7.3 Final Remarks

We believe this work was a successful step toward making software more understandable. Given a program we can automatically characterize its readability: as a whole, in parts, and over time. We can also infer a great deal about the runtime behavior of the program. Furthermore, we can automatically add documentation. For each method we can precisely document the conditions under which it may throw an exception. For each change to a code base we can precisely explain the effect of the change on the program's functional behavior. Finally, we can provide developers with reliable examples for using APIs. The primary findings of this dissertation have been published at major software engineering research venues (see Table 7.2). We hope that many of these techniques will be adopted in the future and that ultimately they will contribute toward a better understanding of program comprehension, more understandable programs, more reliable systems, and more productive developers.

Appendix A

Study Instruments

This appendix presents instruments used in the three human studies described in this dissertation: a study of code readability, a study evaluating our algorithm for documenting code changes, and a study which evaluated our algorithm for API example documentation.

For our study of code readability (see Section 2.2) we asked a group of students at *The University of Virginia* to rate the readability of a set of short snippets of code. To conduct the study we chose to use a light-weight web-based tool shown in Figure A.1. The tool presents each of snippet in sequence. For each snippet the study participant is asked to rate perceived readability on a likert-style 1-5 scale. The ratings are then recorded for later analysis.¹

We employed a second human study to evaluate our algorithm for documentation of code changes. In this study we asked participants to compare the information content of two candidate log messages; one human written and the other generated by our DELTADOC tool. To perform the study we elected to use a paper-based form. Figure A.2 and Figure A.3 include the full set of instructions presented to our participants. To summarize, participants were asked to verify the set of relations described by each message and also indicate their preference. See Section 5.5.2 for details.

Our third and final study was designed to evaluate our algorithm for API example documentation (see Section 6.7). We conducted the study using the web-based tool shown in Figure A.4. Each study participant was presented with a sequence of pairs of examples. Each pair was randomly drawn from the set consisting of output from our tool, out-put from a previous tool, and human-written. Participants were asked to express a preference between each pair of examples {strong preference, some preference, neutral}. Ratings were recorded for later analysis.

¹The dataset and our tool are available at <http://www.cs.virginia.edu/~weimer/readability>.

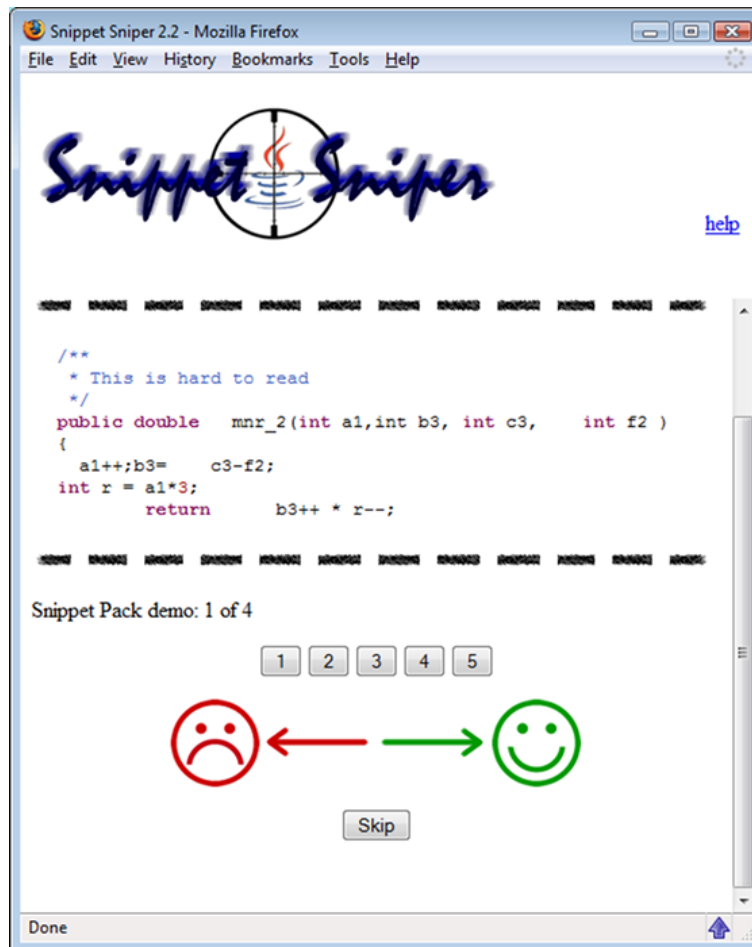


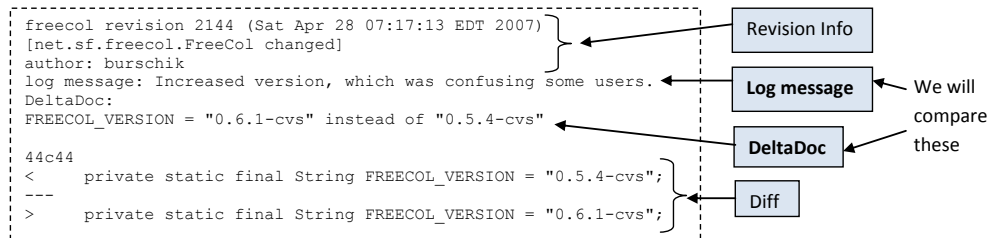
Figure A.1: Web-based tool for annotating the readability of code snippets.

DeltaDoc Annotation Instructions - v1.2

Introduction

DeltaDoc is an *algorithm* that documents changes to code. The output of the algorithm is comparable to *log messages* developers sometimes include when they commit a code change to a version control repository like CVS or Subversion. The goal of this study is to help us understand the similarities and differences between output of the DeltaDoc algorithm and log messages.

For this study we selected a set of 32 changes to 4 real programs. For each change we recorded a *change record* which includes the log message, the output of the DeltaDoc algorithm, and the Diff (textual difference). For example:



You have been given forms that record specific information about each change. Each form enumerates a set of *relations* between *program elements* as expressed in the DeltaDoc and the log message. We consider **only relations from the attached table** which also gives their abbreviations (in the example form below Δ means “changed” and + and – mean “added” and “removed”). Program elements can be *classes*, *methods*, and *variables*. These rules mean that we only compare specific facts about how the change impacts the structure or runtime behavior of the code and not facts about WHY a change was made (e.g., “Fixed bug #123” is ignored). We use a ? to stand for program elements that are implicit and unnamed. For example, from “has an insufficient amount of gold” we can extract: gold < ?.

Benchmark: freecol

Revision number:

Log Message Relations		DeltaDoc Relations	
1	Δ version	A	+ FREECOL_VERSION = "0.6.1-cvs"
2		B	- FREECOL_VERSION = "0.5.4-cvs"

Notes:

Suggested Mapping										Your mapping (if different)										Preferred?									
Log	1	2	3	4	5	6	7	8	9	L	1	2	3	4	5	6	7	8	9	L									
DeltaDoc	A	B	C	D	E	F	G	H	I	D	A	B	C	D	E	F	G	H	I	D									

On each form we have drawn arrows under *Suggested Mapping* to indicate when one relation *implies* another relation. In the above example we inferred that *version* and *FREECOL_VERSION* refer to the same program variable. Moreover, if A or B are true, then 1 must be true. Thus, we have drawn arrows from A to 1 and B to 1. A double-headed arrow may be used if two relations imply each other (e.g., if they contain the same information).

Figure A.2: Instructions for paper-based study for comparing change documentation (page 1/2).

Page 2/2

Your Task

For each change:

- Inspect the set of relations** and decide if they accurately capture all *true* relations.
 To help you do this, we provide all the *change records* in a text file that can be found here:
[arrestedcomputing.com/annotate/\[benchmark name\].txt](http://arrestedcomputing.com/annotate/[benchmark name].txt)
 → If you agree with the set of relations, do nothing. If not, note on the form which relations you would change.
- Inspect the *Suggested Mapping*** and decide if it accurately captures all *implies* relations.
 → If you agree with the suggested mapping, do nothing. If not, record *Your Mapping* in the space provided.
- Check the box** corresponding to the documentation you prefer. Which documentation (DeltaDoc or log message) do you think is more useful for describing the impact of the change? If you can't decide, or think they are about equal, check **BOTH** boxes.

We again suggest you consider the *change record* in making this judgment.

Look this up at:
arrestedcomputing.com/annotate/freecol.txt

Benchmark: freecol

Revision number:

Log Message Relations

1	Δ version
2	

DeltaDoc Relations

A	+ FREECOL_VERSION = "0.6.1-cvs"
B	- FREECOL_VERSION = "0.5.4-cvs"

Notes:

Suggested Mapping

Log	1	2	3	4	5	6	7	8	9	L	1	2	3	4	5	6	7	8	9	L	<input type="checkbox"/>
DeltaDoc	A	B	C	D	E	F	G	H	I	D	A	B	C	D	E	F	G	H	I	D	<input type="checkbox"/>

Your mapping (if different)

Preferred?

Do you see any errors?

If yes, record your own mapping

Which do you think is more useful for describing the change?

Finally, please respond to a few questions in the attached Summary Questions from.

We expect the entire task will take about 30min, to complete. You can send any questions to deltadoc@arrestedcomputing.com Thank you for participating!

Figure A.3: Instructions for paper-based study for comparing change documentation (page 2/2).

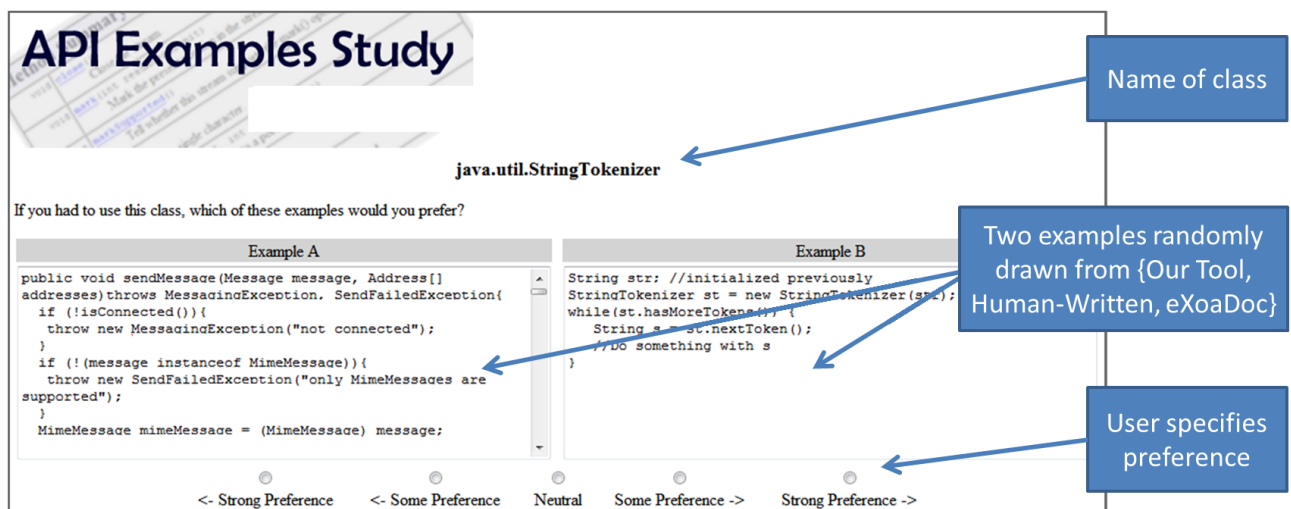


Figure A.4: An annotated screen shot of the web-based study we conducted at *The University of Virginia* (see Section 6.7). Over 150 students indicated their preference on a 5-point Likert-style scale between examples generated by our tool, examples written by humans, and examples mined by EXOADC.

Bibliography

- [1] Peter Hallam. What do programmers really do anyway? In *Microsoft Developer Network (MSDN) — C# Compiler*, Jan 2006.
- [2] Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall, NJ, USA, 2001.
- [3] David Parnas. Software aging. In *Software Fundamentals*. Addison-Wesley, 2001.
- [4] David Zokaities. Writing understandable code. In *Software Development*, pages 48–49, jan 2002.
- [5] Lionel E. Deimel Jr. The uses of program reading. *SIGCSE Bull.*, 17(2):5–14, 1985.
- [6] Robert Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley, 2003.
- [7] Darrell R. Raymond. Reading source code. In *Conference of the Centre for Advanced Studies on Collaborative Research*, pages 3–16. IBM Press, 1991.
- [8] Spencer Rugaber. The use of domain knowledge in program understanding. *Ann. Softw. Eng.*, 9(1-4):143–192, 2000.
- [9] Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. John Wiley & Sons, Inc., 1996.
- [10] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman, MA, USA, 2003.
- [11] NASA Software Reuse Working Group. Software reuse survey. In http://www.esdswg.com/softwarereuse/Resources/library/working_group_documents/survey2005, 2005.
- [12] Jr. Frederick P. Brooks. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.
- [13] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Programming Language Design and Implementation*, June 9–11 2003.
- [14] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Technical Report 02-3, Research Triangle Institute, May 2002.
- [15] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122, May 2001.
- [16] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2004.
- [17] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: path-sensitive program verification in polynomial time. *SIGPLAN Notices*, 37(5):57–68, 2002.

- [18] Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.
- [19] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.
- [20] David Hovemeyer and William Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, New York, NY, USA, 2004. ACM Press.
- [21] David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. *SIGSOFT Softw. Eng. Notes*, 31(1):13–19, 2006.
- [22] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Programming language design and implementation*, pages 141–154, 2003.
- [23] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.
- [24] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
- [25] Frederick P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.
- [26] R.W. Selby. Enabling reuse-based software development of large-scale systems. *Software Engineering, IEEE Transactions on*, 31(6):495–510, June 2005.
- [27] W B Frakes and B A Nejme. Software reuse through information retrieval. *SIGIR Forum*, 21(1-2):30–36, 1987.
- [28] Krishan Aggarwal, Yogesh Singh, and Jitender Kumar Chhabra. An integrated measure of software maintainability. *Reliability and Maintainability Symposium*, pages 235–241, September 2002.
- [29] Aaron Watters, Guido van Rossum, and James C. Ahlstrom. *Internet Programming with Python*. MIS Press/Henry Holt publishers, New York, 1996.
- [30] Phillip A. Relf. Tool assisted identifier naming for improved software readability: an empirical study. *Empirical Software Engineering*, November 2005.
- [31] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, 1976.
- [32] George C. Necula and Peter Lee. The design and implementation of a certifying compiler (with retrospective). In *Best of Programming Languages Design and Implementation*, pages 612–625, 1998.
- [33] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, California, 1995.
- [34] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Programming Language Design and Implementation*, pages 35–46, 2000.
- [35] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21:848–894, July 1999.
- [36] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Programming Language Design and Implementation*, pages 290–299, 2007.

- [37] John B. Goodenough. Exception handling: issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, 1975.
- [38] Robert Miller and Anand Tripathi. Issues with exception handling in object-oriented systems. In *European Conference on Object-Oriented Programming*, pages 85–103, 1997.
- [39] Martin P. Robillard and Gail C. Murphy. Regaining control of exception handling. Technical Report TR-99-14, Dept. of Computer Science, University of British Columbia, 1, 1999.
- [40] G. Alonso, C. Hagen, D. Agrawal, A. El Abbadi, and C. Mohan. Enhancing the fault tolerance of workflow management systems. *IEEE Concurrency*, 8(3):74–81, July 2000.
- [41] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. Discovering faults in idiom-based exception handling. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 242–251, 2006.
- [42] Tom Cargill. Exception handling: a false sense of security. *C++ Report*, 6(9), 1994.
- [43] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, 1996.
- [44] Donna Malayeri and Jonathan Aldrich. Practical exception specifications. In *Advanced Topics in Exception Handling Techniques*, pages 200–220, 2006.
- [45] David G. Novick and Karen Ward. What users say they want in documentation. In *Conference on Design of Communication*, pages 84–91, 2006.
- [46] John Anvik, Lyndon Hiew, and Gail C. Murphy. Coping with an open bug repository. In *Eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 35–39, New York, NY, USA, 2005. ACM Press.
- [47] A. Mockus and L.G. Votta. Identifying reasons for software changes using historic databases. In *International Conference on Software Maintenance*, pages 120–130, 2000.
- [48] Giuliano Antoniol, Massimiliano Di Penta, Harald Gall, and Martin Pinzger. Towards the integration of versioning systems, bug reports and source code meta-models. *Electr. Notes Theor. Comput. Sci.*, 127(3):87–99, 2005.
- [49] Rajeev Jain. API-writing and API-documentation. In <http://api-writing.blogspot.com/>, April 2008.
- [50] Samuel G. McLellan, Alvin W. Roesler, Joseph T. Tempest, and Clay I. Spinuzzi. Building more usable apis. *IEEE Softw.*, 15(3):78–86, 1998.
- [51] Janet Nykaza, Rhonda Messinger, Fran Boehme, Cherie L. Norman, Matthew Mace, and Manuel Gordon. What programmers really want: results of a needs assessment for sdk documentation. In *International Conference on Computer documentation*, pages 133–141, 2002.
- [52] Forrest Shull, Filippo Lanubile, and Victor R. Basili. Investigating reading techniques for object-oriented framework learning. *IEEE Trans. Softw. Eng.*, 26(11):1101–1118, 2000.
- [53] Jeffrey Stylos, Brad A. Myers, and Zizhuang Yang. Jadeite: improving api documentation using usage information. In *extended abstracts on Human factors in computing systems*, pages 4429–4434, 2009.
- [54] Martin P. Robillard. What makes apis hard to learn? answers from developers. *IEEE Softw.*, 26(6):27–34, 2009.
- [55] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Principles of programming languages*, pages 4–16, 2002.

- [56] Westley Weimer and George C. Nacula. Mining temporal specifications for error detection. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 461–476, 2005.
- [57] Claire Le Goues and Westley Weimer. Specification mining with few false positives. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2009.
- [58] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. Towards an intelligent code search engine. In *AAAI Conference on Artificial Intelligence*, 2010.
- [59] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and recommending API usage patterns. In *European Conference on Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 318–343. Springer Berlin / Heidelberg, 2009.
- [60] Donald Knuth. Literate programming (1984). In *CSLI*, page 99, 1992.
- [61] James L. Elshoff and Michael Marcotty. Improving computer program readability to aid modification. *Commun. ACM*, 25(8):512–521, 1982.
- [62] John C. Knight and E. Ann Myers. Phased inspections and their implementation. *SIGSOFT Softw. Eng. Notes*, 16(3):29–35, 1991.
- [63] Forrest Shull, Ioana Rus, and Victor Basili. Improving software inspections by using reading techniques. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 726–727, Washington, DC, USA, 2001. IEEE Computer Society.
- [64] Nuzhat J. Haneef. Software documentation and readability: a proposed process improvement. *SIGSOFT Softw. Eng. Notes*, 23(3):75–77, 1998.
- [65] R. F. Flesch. A new readability yardstick. *Journal of Applied Psychology*, 32:221–233, 1948.
- [66] R. Gunning. *The Technique of Clear Writing*. McGraw-Hill International Book Co, New York, 1952.
- [67] G. Harry McLaughlin. Smog grading – a new readability. *Journal of Reading*, May 1969.
- [68] J. P. Kinciad and E. A. Smith. Derivation and validation of the automated readability index for use with technical materials. *Human Factors*, 12:457–464, 1970.
- [69] Scott MacHaffie, Robin McLeod, Bill Roberts, Philip Todd, and Leigh Anderson. A readability metric for computer-generated mathematics. Technical report, Saltire Software, <http://www.saltire.com/equation.html>, retrieved 2007.
- [70] Benjamin B. Bederson, Ben Shneiderman, and Martin Wattenberg. Ordered and quantum treemaps: Making effective use of 2d space to display hierarchies. *ACM Trans. Graph.*, 21(4):833–854, 2002.
- [71] A. E. Hatzimanikatis, C. T. Tsalidis, and D. Christodoulakis. Measuring the readability and maintainability of hyperdocuments. *Journal of Software Maintenance*, 7(2):77–90, 1995.
- [72] E. J. Weyuker. Evaluating software complexity measures. *IEEE Trans. Softw. Eng.*, 14(9):1357–1365, 1988.
- [73] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 284–292, 2005.
- [74] Scott Ambler. Java coding standards. *Softw. Dev.*, 5(8):67–71, 1997.
- [75] L. W. Cannon, R. A. Elliott, L. W. Kirchhoff, J. H. Miller, J. M. Milner, R. W. Mitze, E. P. Schan, N. O. Whittington, Henry Spencer, David Keppel, , and Mark Brader. *Recommended C Style and Coding Standards: Revision 6.0*. Specialized Systems Consultants, Inc., Seattle, Washington, June 1990.
- [76] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley Professional, 2004.

- [77] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001.
- [78] C. V. Ramamoorthy and Wei-Tek Tsai. Advances in software engineering. *Computer*, 29(10):47–58, 1996.
- [79] T. Tenny. Program readability: Procedures versus comments. *IEEE Trans. Softw. Eng.*, 14(9):1271–1279, 1988.
- [80] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.
- [81] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 140:44–53, 1932.
- [82] Steven E. Stemler. A comparison of consensus, consistency, and measurement approaches to estimating interrater reliability. *Practical Assessment, Research and Evaluation*, 9(4), 2004.
- [83] Jody Kashden and Michael D. Franzen. An interrater reliability study of the luria-nebraska neuropsychological battery form-ii quantitative scoring system. *Archives of Clinical Neuropsychology*, 11:155–163, 1996.
- [84] Yasmine S. Wasfi, Cecile S. Rose, James R. Murphy, Lori J. Silveira, Jan C. Grutters, Yoshikazu Inoue, Marc A. Judson, and Lisa A. Maier. A new tool to assess sarcoidosis severity. *CHEST*, 129(5):1234–1245, 2006.
- [85] Lawrence L. Giventer. *Statistical Analysis in Public Administration*. Jones and Bartlett Publishers, 2007.
- [86] Tom Mitchell. *Machine Learning*. McGraw Hill, 1997.
- [87] G. Holmes, A. Donkin, and I.H. Witten. Weka: A machine learning workbench. *Australia and New Zealand Conference on Intelligent Information Systems*, 1994.
- [88] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *International Joint Conference on Artificial Intelligence*, 14(2):1137–1145, 1995.
- [89] Tsong Yueh Chen, Fei-Ching Kuo, and Robert Merkel. On the statistical properties of the f-measure. In *International Conference on Quality Software*, pages 146–153, 2004.
- [90] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.
- [91] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.
- [92] jUnit.org. jUnit 4.0 now available. In http://sourceforge.net/forum/forum.php?forum_id=541181, February 2006.
- [93] Charles Simonyi. Hungarian notation. *MSDN Library*, November 1999.
- [94] Richard J. Miara, Joyce A. Musselman, Juan A. Navarro, and Ben Shneiderman. Program indentation and comprehensibility. *Commun. ACM*, 26(11):861–867, 1983.
- [95] Takanobu Baba, Tomohisa Masuho, Takashi Yokota, and Kanemitsu Ootsu. Design of a two-level hot path detector for path-based loop optimizations. In *Advances in Computer Science and Technology*, pages 23–28, 2007.
- [96] Thomas Ball and James R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.
- [97] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. *SIGPLAN Not.*, 35(11):202–211, 2000.

- [98] Matthew C. Merten, Andrew R. Trick, Christopher N. George, John C. Gyllenhaal, and W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *International Symposium on Computer architecture*, pages 136–147, 1999.
- [99] Rajiv Gupta, Eduard Mehofer, and Youtao Zhang. Profile-guided compiler optimizations. In *The Compiler Design Handbook*, pages 143–174. ACM, 2002.
- [100] P. Berube and J.N. Amaral. Aestimo: a feedback-directed optimization evaluation tool. In *Performance Analysis of Systems and Software*, pages 251–260, 2006.
- [101] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *SIGSOFT Softw. Eng. Notes*, 22(6):432–449, 1997.
- [102] Glenn Ammons and James R. Larus. Improving data-flow analysis with path profiles. *SIGPLAN Not.*, 33(5):72–84, 1998.
- [103] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. *SIGPLAN Not. (best of PLDI '82)*, 39(4):49–57, 2004.
- [104] Amitabh Srivastava and Alan Eustace. Atom: a system for building customized program analysis tools. In *Programming language design and implementation*, pages 196–205, 1994.
- [105] Westley Weimer and George C. Necula. Exceptional situations and program reliability. *ACM Trans. Program. Lang. Syst.*, 30(2):1–51, 2008.
- [106] James R. Larus. Whole program paths. In *Programming language design and implementation*, pages 259–269, 1999.
- [107] James F. Bowring, James M. Rehg, and Mary Jean Harrold. Active learning for automatic classification of software behavior. *SIGSOFT Softw. Eng. Notes*, 29(4):195–205, 2004.
- [108] David W. Wall. Predicting program behavior using real or estimated profiles. *SIGPLAN Not.*, 26(6):59–70, 1991.
- [109] Antoine Monsifrot, François Bodin, and Rene Quiniou. A machine learning approach to automatic production of compiler heuristics. In *AIMSA '02: Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pages 41–50, London, UK, 2002. Springer-Verlag.
- [110] Thomas Ball and James R. Larus. Branch prediction for free. In *Programming language design and implementation*, pages 300–313, 1993.
- [111] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Evidence-based static branch prediction using machine learning. *ACM Trans. Program. Lang. Syst.*, 19(1):188–222, 1997.
- [112] Thomas Ball, Peter Mataga, and Mooly Sagiv. Edge profiling versus path profiling: the showdown. In *Principles of programming languages*, pages 134–148, 1998.
- [113] Diwakar Krishnamurthy, Jerome A. Rolia, and Shikharesh Majumdar. A synthetic workload generation technique for stress testing session-based systems. *IEEE Trans. Softw. Eng.*, 32(11):868–882, 2006.
- [114] Pankaj Mehra and Benjamin Wah. Synthetic workload generation for load-balancing experiments. *IEEE Parallel Distrib. Technol.*, 3(3):4–19, 1995.
- [115] A. Nanda and L. M. Ni. Benchmark workload generation and performance characterization of multiprocessors. In *Conference on Supercomputing*, pages 20–29. IEEE Computer Society Press, 1992.
- [116] Koushik Sen. Concolic testing. In *Automated Software Engineering*, pages 571–572, 2007.

- [117] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*, pages 419–423, 2006.
- [118] Matthew Arnold, Stephen J. Fink, Vivek Sarkar, and Peter F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 52–64, 2000.
- [119] David W. Wall. Global register allocation at link time. In *Fast Printed Circuit Board Routing. WRL Research Report 86/3*, pages 264–275, 1986.
- [120] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation, PLDI '90*, pages 16–27, New York, NY, USA, 1990. ACM.
- [121] Norbert Esser, Renga Sundararajan, and Joachim Trescher. Improving trimedia cache performance by profile guided code reordering. In *Proceedings of the 7th international conference on Embedded computer systems: architectures, modeling, and simulation, SAMOS'07*, pages 96–106, Berlin, Heidelberg, 2007. Springer-Verlag.
- [122] S. McFarling. Program optimization for instruction caches. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems, ASPLOS-III*, pages 183–191, New York, NY, USA, 1989. ACM.
- [123] Simon F. Goldsmith, Alex S. Aiken, and Daniel S. Wilkerson. Measuring empirical computational complexity. In *Foundations of software engineering*, pages 395–404, 2007.
- [124] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Principles of Programming Languages*, pages 4–16, 2002.
- [125] Ted Kremenek and Dawson R. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Static Analysis Symposium*, pages 295–315, 2003.
- [126] Ulrik Pagh Schultz, Julia L. Lawall, and Charles Consel. Automatic program specialization for java. *ACM Trans. Program. Lang. Syst.*, 25(4):452–499, 2003.
- [127] Michael Boyer, Kevin Skadron, and Westley Weimer. Automated dynamic analysis of CUDA programs. In *Workshop on Software Tools for MultiCore Systems*, 2008.
- [128] Raja Vallée-Rai et. al. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.
- [129] Edward J. Emond and David W. Mason. A new rank correlation coefficient with application to the consensus ranking problem. In *Journal of Multi-Criteria Decision Analysis*, pages 17–28, Department of National Defence, Operational Research Division, Ottawa, Ontario, Canada, 2002. John Wiley Sons, Ltd.
- [130] Shaula Yemini and Daniel Berry. A modular verifiable exception handling mechanism. *ACM Transactions on Programming Languages and Systems*, 7(2), April 1985.
- [131] Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In *Conference on Object-oriented programming, systems, languages, and applications*, pages 419–431, 2004.
- [132] Martin P. Robillard and Gail C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol.*, 12(2):191–221, 2003.
- [133] Barbara G. Ryder, Donald Smith, Ulrich Kremer, Michael Gordon, and Nirav Shah. A static study of Java exceptions using JESP. In *International Conference on Compiler Construction*, pages 67–81, 2000.
- [134] Ramkrishna Chatterjee, Barbara G. Ryder, and William Landi. Complexity of points-to analysis of java in the presence of exceptions. *IEEE Trans. Software Eng.*, 27(6):481–512, 2001.

- [135] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of java programs. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 21–31, 1999.
- [136] Manish Gupta, Jong-Deok Choi, and Michael Hind. Optimizing Java programs in the presence of exceptions. In *European Conference on Object-Oriented Programming*, pages 422–446, 2000.
- [137] Saurabh Sinha and Mary Jean Harrold. Criteria for testing exception-handling constructs in java programs. In *ICSM*, pages 265–, 1999.
- [138] Saurabh Sinha, Alessandro Orso, and Mary Jean Harrold. Automated support for development, maintenance, and testing in the presence of implicit control flow. *icse*, 0:336–345, 2004.
- [139] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. A study of the documentation essential to software maintenance. In *International Conference on Design of Communication*, pages 68–75, 2005.
- [140] Douglas Kramer. Api documentation from source code comments: a case study of javadoc. In *International Conference on Computer Documentation*, pages 147–153, 1999.
- [141] Andrew Forward and Timothy C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *DocEng '02: Proceedings of the 2002 ACM symposium on Document engineering*, pages 26–33, 2002.
- [142] Shihong Huang and Scott Tilley. Towards a documentation maturity model. In *International Conference on Documentation*, pages 93–99, 2003.
- [143] Bill Thomas and Scott Tilley. Documentation for software engineers: what is needed to aid system understanding? In *International Conference on Computer Documentation*, pages 235–236, 2001.
- [144] Byeong-Mo Chang, Jang-Wu Jo, Kwangkeun Yi, and Kwang-Moo Choe. Interprocedural exception analysis for java. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 620–625, 2001.
- [145] Michael Hind. Pointer analysis: haven’t we solved this problem yet? In *Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.
- [146] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.
- [147] Scott Tilley and Hausi Müller. Info: a simple document annotation facility. In *International Conference on Systems Documentation*, pages 30–36, 1991.
- [148] Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Jeanne Ferrante. Path analysis and renaming for predicated instruction scheduling. *International Journal of Parallel Programming*, 28(6):563–588, 2000.
- [149] Ranjit Jhala and Rupak Majumdar. Path slicing. In *Programming Language Design and Implementation*, pages 38–47, 2005.
- [150] Torsten Robschink and Gregor Snelting. Efficient path conditions in dependence graphs. In *International Conference on Software Engineering*, pages 478–488, 2002.
- [151] Kenneth L. McMillan. Applications of craig interpolants in model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–12, 2005.
- [152] Amanda McPherson Greg Kroah-Hartman, Jonathan Corbet. Linux kernel development. *The Linux Foundation*, 2009.
- [153] Marc J. Rochkind. The source code control system. *IEEE Trans. Software Eng.*, 1(4):364–370, 1975.

- [154] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–367, 2009.
- [155] Ranjith Purushothaman and Dewayne E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Trans. Softw. Eng.*, 31(6):511–526, 2005.
- [156] Joseph F McCarthy and Wendy G Lehnert. Using decision trees for coreference resolution. In *Joint Conference on Artificial Intelligence*, pages 1050–1055, 1995.
- [157] Wee Meng Soon, Hwee Tou Ng, and Daniel Chung Yong Lim. A machine learning approach to coreference resolution of noun phrases. *Comput. Linguist.*, 27(4):521–544, 2001.
- [158] Claire Cardie and Kiri Wagstaff. Noun phrase coreference as clustering. In *Joint Conference on Empirical Methods in NLP and Very Large Corpora*, pages 82–89, 1999.
- [159] C.-Y. Lin and F. J. Och. Looking for a few good metrics: Rouge and its evaluation. In *NTCIR Workshop*, 2004.
- [160] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar T. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Foundations of Software Engineering*, pages 121–130, 2009.
- [161] Taweessup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Jdiff: A differencing technique and tool for object-oriented programs. *Automated Software Engg.*, 14(1):3–36, 2007.
- [162] Alex Loh and Miryung Kim. Lsdiff: a program differencing tool to identify systematic structural differences. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE '10*, pages 263–266, New York, NY, USA, 2010. ACM.
- [163] Kevin J. Hoffman, Patrick Eugster, and Suresh Jagannathan. Semantics-aware trace analysis. *SIGPLAN Not.*, 44(6):453–464, 2009.
- [164] H. P. Luhn. The automatic creation of literature abstracts. *IBM Journal of Research and Development*, 2(2):159–165, 1958.
- [165] Ramakrishna Varadarajan and Vagelis Hristidis. A system for query-specific document summarization. In *Information and knowledge management*, pages 622–631, 2006.
- [166] B. A. Mathis, J. E. Rush, and C. E. Young. Improvement of automatic abstracts by the use of structural analysis. *Journal of the American Society for Information Science*, 24(2):101–109, 1973.
- [167] Sara Comai, Stefania Marrara, and Letizia Tanca. XML document summarization: Using XQuery for synopsis creation. In *Database and Expert Systems Applications*, pages 928–932, 2004.
- [168] D. Cai, X. He, J. Wen, and W. Ma. Block-level link analysis. *SIGIR Research and development in information retrieval*, pages 440–447, 2004.
- [169] C.H. Lee, M.Y. Kan, and S. Lai. Stylistic and lexical cotraining for web block classification. In *Workshop on Web information and data management*, pages 136–143, 2004.
- [170] R. Song, H. Liu, J. Wen, and W. Ma. Learning block importance models for web pages. In *International World Wide Web Conference*, pages 203–211, 2004.
- [171] D. Binkley, R. Capellini, R. Raszewski, and C. Smith. An implementation of and experiment with semantic differencing. In *International Conference on Software Maintenance*, page 82, 2001.
- [172] Elizabeth Soechting, Kinga Dobolyi, and Westley Weimer. Syntactic regression testing for tree-structured output. *International Symposium on Web Systems Evolution*, September 2009.

- [173] R. Holmes, R. Cottrell, R.J. Walker, and J. Denzinger. The end-to-end use of source code examples: An exploratory study. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 555–558, sept 2009.
- [174] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *Programming Languages Design and Implementation*, pages 48–61, 2005.
- [175] Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM.
- [176] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 117–125, New York, NY, USA, 2005. ACM.
- [177] javadoctool@sun.com. How to write doc comments for the javadoc tool. In <http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>, 2010.
- [178] Oracle. Java SE 6 documentation. In <http://download.oracle.com/javase/6/docs/>, 2010.
- [179] Rajeev Alur, Pavol Cerny, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for Java classes. In *Principles of Programming Languages*, 2005.
- [180] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *International Symposium of Software Testing and Analysis*, 2002.
- [181] L. Kaufman and P.J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley Interscience, New York, 1990.
- [182] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 35–44, New York, NY, USA, 2007. ACM.
- [183] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 383–392, New York, NY, USA, 2009. ACM.
- [184] Uri Dekel and James D. Herbsleb. Improving api documentation usability with knowledge pushing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 320–330, Washington, DC, USA, 2009. IEEE Computer Society.
- [185] Robert Dunn. *Software Defect Removal*. McGraw-Hill, 1984.
- [186] SpringSource. Spring framework, December 2008. <http://www.springframework.org/>.
- [187] Ryan Berg and Dinis Cruz. Security vulnerabilities in the spring framework model view controller. White paper, Ounce Labs, September 2008. Available online (21 pages).
- [188] Mira Kajko-Mattsson. The state of documentation practice within corrective maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance, ICSM*, pages 354–363, 2001.
- [189] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *Automated software engineering*, pages 34–43, 2007.
- [190] Nicholas Jalbert and Westley Weimer. Automated duplicate detection for bug tracking systems. In *Dependable Systems and Networks*, pages 52–61, 2008.

- [191] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *Lecture Notes in Computer Science*, volume 2648, pages 121–135, January 2003.
- [192] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worm epidemics. *ACM Trans. Comput. Syst.*, 26(4):1–68, 2008.
- [193] Westley Weimer. Patches as better bug reports. In *Generative Programming and Component Engineering*, pages 181–190, 2006.
- [194] Westley Weimer, ThanVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–374, 2009.
- [195] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102, New York, NY, USA, 2009. ACM.
- [196] Stelios Sidiroglou and Angelos D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6):41–49, 2005.
- [197] Raymond P. L. Buse and Westley R. Weimer. A metric for software readability. In *International Symposium on Software Testing and Analysis*, pages 121–130, 2008.
- [198] Raymond P.L. Buse and Westley R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36:546–558, 2010.
- [199] Raymond P. L. Buse and Westley Weimer. The road not taken: Estimating path execution frequency statically. In *International Conference on Software Engineering*, 2009.
- [200] Raymond P. L. Buse and Westley Weimer. Automatic documentation inference for exceptions. In *International Symposium on Software Testing and Analysis*, pages 273–282, 2008.
- [201] Raymond P.L. Buse and Westley R. Weimer. Automatically documenting program changes. In *Automated software engineering*, ASE '10, pages 33–42, 2010.
- [202] Raymond P. L. Buse and Westley Weimer. Synthesizing api usage examples. In *International Conference on Software Engineering [To Appear]*, 2012.