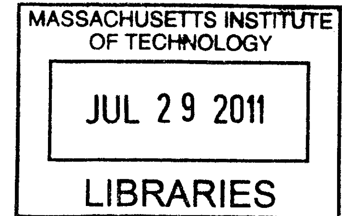
Improved Source Code Editing for Effective Ad-hoc Code Reuse

by

Sangmok Han

S.M. Mechanical Engineering
Massachusetts Institute of Technology, 2006

B.S. Mechanical Engineering
Seoul National University, 2000


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 29 2011
LIBRARIES
ARCHIVES

Submitted to the Department of Mechanical Engineering
in Partial Fulfillment of The Requirements for the Degree of
Doctoral of Philosophy in Mechanical Engineering
at the
Massachusetts Institute of Technology

February 2011

© 2011 Massachusetts Institute of Technology. All rights reserved.

The author hereby grants to MIT permission to reproduce
and to distribute publicly paper and electronic
copies of this thesis document in whole or in part
in any medium now known or hereafter created.

Signature of Author . . . .
Department of Mechanical Engineering
January 15, 2011

Certified by. . . . .
David Wallace
Professor of Mechanical Engineering
Thesis Supervisor

Accepted by. . . . .
David E. Hardt
Graduate Officer, Department of Mechanical Engineering

Improved Source Code Editing for Effective Ad-hoc Code Reuse

by

Sangmok Han

Submitted to the Department of Mechanical Engineering
on January 15, 2011 in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Mechanical Engineering

Abstract

Code reuse is essential for productivity and software quality. Code reuse based on abstraction mechanisms in programming languages is a standard approach, but programmers also reuse code by taking an ad-hoc approach, in which text of code is reused without abstraction. This thesis focuses on improving two common ad-hoc code reuse approaches—*code template reuse* and *code phrase reuse*—because they are not only frequent, but also, more importantly, they pose a risk to quality and productivity in software development, the original aims of code reuse.

The first ad-hoc code reuse approach, code template reuse refers to programmers reusing an existing code fragment as a structural template for similar code fragments. Programmers use the code reuse approach because using abstraction mechanisms requires extra code and preplanning. When similar code fragments, which are only different by several code tokens, are reused just a couple of times, it makes sense to reuse text of one of the code fragments as a template for others. Unfortunately, code template reuse poses a risk to software quality because it requires repetitive and tedious editing steps. Should a programmer forget to perform any of the editing steps, he may introduce program bugs, which are difficult to detect by visual inspection, code compilers, or other existing bug detection methods.

The second ad-hoc code reuse approach, code phrase reuse refers to programmers reusing common code phrases by retyping them, often regularly, using code completion. Programmers use the code reuse approach because no abstraction mechanism is available for reusing short yet common code phrases. Unfortunately, code phrase reuse poses a limitation on productivity because retyping the same code phrases is time-consuming even when a code completion system is used. Existing code completion systems completes only one word at a time. As a result, programmers have to repeatedly invoke code completion, review code completion candidates, and select a correct candidate as many times as the number of words in a code phrase.

This thesis presents new models, algorithms, and user interfaces for effective ad-hoc code reuse. First, to address the risk posed by code template reuse, it develops a method for detecting program bugs in similar code fragments by analyzing sequential patterns of code tokens. To proactively reduce program bugs introduced during code template reuse, this thesis proposes an error-preventive code editing method that reduces the number of code editing steps based on cell-based text editing. Second, to address the productivity limitation posed by code phrase reuse, this thesis develops an efficient code phrase completion method. The code phrase completion accelerates reuse of common code phrases by taking non-predefined abbreviated input and expanding it into a full code phrase. The code phrase completion method utilizes a statistical model called Hidden Markov model trained on a corpus of code and abbreviation examples.

Finally, the new methods for bug detection and code phrase completion are evaluated through corpus and user studies. In 7 well-maintained open source projects, the bug detection method found 87 previously unknown program bugs. The ratio of actual bugs to bug warnings (precision) was 47% on average, eight times higher than previous similar methods. The code phrase completion method is evaluated on the basis of accuracy and time savings. It achieved 99.3% accuracy in a corpus study and achieved 30.4% time savings and 40.8% keystroke savings in a user study when compared to a conventional code completion method. At a higher level, this work demonstrates the power of a simple sequence-based model of source code. Analyzing vertical sequences of code tokens across similar code fragments is found useful for accurate bug detection; learning to infer horizontal sequences of code tokens is found useful for efficient code completion. Ultimately, this work may aid the development of other sequence-based models of source code, as well as different analysis and inference techniques, which can solve previously difficult software engineering problems.

Thesis Supervisor: David Wallace
Professor of Mechanical Engineering

사랑하는 부모님께

To my parents

Contents

Chapter 1 Introduction **11**
1.1 Problems of Ad-hoc Code Reuse 12
1.2 Solution Approaches 13
1.2.1 Effective Code Template Reuse 14
1.2.2 Effective Code Phrase Reuse 14
1.3 Key Contributions 15
1.4 Thesis Outline 16
Chapter 2 Related Work **17**
2.1 Overview 17
2.2 Copy-paste-related Bug Detection 18
2.3 Simultaneous Text Editing 20
2.4 Code Completion 21
2.4.1 Conventional, Single-keyword Code Completion 21
2.4.2 Multiple-keyword Code Completion 22
2.4.3 Text Entry Systems for Natural Language Input 23
Chapter 3 Effective Code Template Reuse **24**
3.1 Overview 24
3.1.1 Automated Bug Detection 24
3.1.2 Error-preventive Code Editing 26
3.2 Hypothesis 27
3.2.1 New Hypothesis on the Source of Program Bugs 27
3.2.2 Common Editing Errors during Code Template Reuse 27
3.2.3 Characterization of Program Bugs Caused by Common Editing Errors 30
3.3 Model and Algorithm 32
3.3.1 Sequence-based Source Code Model 32
3.3.2 Target Code Regions for the Search of Sequence-driven Code 33
3.3.3 Sequence-driven Code Search Algorithm 35
3.3.4 Bug Detection Algorithm Based on Code Token Sequence Analysis 38
3.4 Implementation of the Bug Detection Method 44
3.4.1 Implementation 44
3.4.2 Iterative Development Process 45
3.5 Evaluation of the Bug Detection Method 46
3.5.1 Selection of Open Source Projects 46

3.5.2 Bug Finding Experiment Procedure....46
3.6 Results and Discussion of the Bug Detection Method....47
3.6.1 Empirical Support for the Hypothesis....47
3.6.2 Evaluation of Accuracy....48
3.6.3 Comparison with a Previous Method for Copy-paste-related Bug Detection....51
3.7 User Interface for Error-preventive Code Editing....52
3.7.1 Efficient Multiple-Selection User Interface....53
3.7.2 Efficient Code Generation User Interface for Code Template Reuse....58
3.7.3 Predictive Evaluation....60
3.8 Discussion of Error-preventive Code Editing....63
3.8.1 Estimated Time Savings....63
3.8.2 Comparison with Existing Code Generation Methods....64
3.8.3 Multiple-Selection Copy-Paste Support for Handling Many New Code Tokens....65
3.8.4 Generation of Sequence-driven Code with More than One Code Token Sequences....66
Chapter 4 Effective Code Phrase Reuse....69
4.1 Overview....69
4.2 Groundwork: Value and Feasibility Estimation....72
4.2.1 Reuse Potential of Code Phrases....72
4.2.2 N-Gram Entropy Comparison: Java vs. English....74
4.3 Model and Algorithm....74
4.3.1 Code Completion of Multiple Keywords as a Decoding Problem of an HMM....75
4.3.2 An Extended Hidden Markov Model for Code Completion by Disabbreviation....76
4.3.3 Modified Viterbi Algorithm....78
4.3.4 An *n*-Gram Model for Code Completion by Extrapolation....79
4.4 Parameter Estimation....82
4.4.1 Estimation of Start Probabilities and Transition Probabilities....82
4.4.2 Estimation of Match Probabilities....84
4.4.3 Estimation of Conditional Probability Distribution of the *n*-Gram Model....88
4.5 User Interface and Implementation....89
4.5.1 User Interface....89
4.5.2 Incremental Feedback of Code Completions....91
4.5.3 Incremental Indexing of Source Code....91
4.6 Artificial Corpus Study....92
4.6.1 Study Setup....92
4.6.2 Results and Discussion....96
4.7 User Study....101

4.7.1 Participants........102
4.7.2 Usage Scenario and Assumptions........102
4.7.3 Study Setup........102
4.7.4 Tasks........103
4.7.5 Results........104
4.7.6 Discussion........106
Chapter 5 Conclusions........108
5.1 Summary........108
5.2 Future Work........109
References........111
Acknowledgment........114

List of Figures

Figure 1.1: An example of code template reuse. 12

Figure 1.2: A program bug found in the source code of Eclipse 13

Figure 1.3: In conventional code completion, code completion of code phrases might require many extra keystrokes for driving code completion dialogs. 13

Figure 2.1: An example of program bugs in short similar code fragments with less than 30 code tokens. 19

Figure 2.2: An example of program bugs that cannot be detected by previous methods 20

Figure 2.3: An inference-based approach to multiple-selection specification. 21

Figure 3.1: A usage scenario of the new bug detection method. 25

Figure 3.2: A usage scenario of the new code editing method for reducing code editing steps for code template reuse.. 26

Figure 3.3: The first type of common text editing error: forget-to-replace 28

Figure 3.4: The second type of common text editing error: replace-with-wrong. 30

Figure 3.5: An example of sequence-driven code consisting of three structurally equivalent, adjacent code fragments 32

Figure 3.6: An example of sequence-driven code found in a *switch-case* statement. 33

Figure 3.7: An example of sequence-driven code found in an *if-else-if* statement. 33

Figure 3.8: An example of sequence-driven code found in consecutive if statements. 34

Figure 3.9: An example of sequence-driven code found in consecutive method declarations. 34

Figure 3.10: An example of sequence-driven code found in consecutive statement expressions 35

Figure 3.11: The sequence-driven code search algorithm. 36

Figure 3.12: Supporting functions used by the sequence-driven code search algorithm. 38

Figure 3.13: The bug detection algorithm can be viewed as looking for a certain suspicious code token pattern. 39

Figure 3.14: Bug detection algorithm based on code token sequence analysis. 44

Figure 3.15: An example of bug warnings reported in a spreadsheet file 47

Figure 3.16: Sequence-driven code with a program bug found in NetBeans. 49

Figure 3.17: Screenshots of TexelEdit, an experimental text editor supporting cell-based text editing 54

Figure 3.18: Mouse-click selection in cell-based text editing 55

Figure 3.19: Mouse-double-click selection in cell-based text editing. 56

Figure 3.20: Mouse-drag selection in cell-based text editing 57

Figure 3.21: Keyboard-based selection in cell-based text editing. 58

Figure 3.22: A user interface for collecting required information for automatically generating sequence-

driven code. 59
Figure 3.23: Sequence-driven code generated from user input. 60
Figure 3.24: A comparison of estimated time usage between normal and cell-based text editing. 63
Figure 3.25: An example of Code-Template necessary for code template reuse when using the markup-language-based approach. 64
Figure 3.26: An example of Code-Template necessary for code template reuse when using the markup-language-based approach. 65
Figure 3.27: New code token text can be copy-pasted from multiple selections in a comma separated format. 66
Figure 3.28: An example of sequence-driven code with two unique code token sequences. 67
Figure 3.29: Cell-based text editing supports a multiple-selection replace command, called type-to-replace, which can accelerate common text replacement tasks during code template reuse. 68
Figure 4.1: Abbreviation Completion can complete multiple keywords from abbreviated input in a single code completion dialog. 69
Figure 4.2:A user can extrapolate a code completion candidate by pressing the Tab key. 70
Figure 4.3:Resolution of multiple keywords is solved as a decoding problem of Hidden Markov Model. 76
Figure 4.4:A modified HMM with match indicator nodes. 77
Figure 4.5: A user interface for multiple-keyword code completion 90
Figure 4.6: The system gives consistent top-N accuracy across the open source projects in code completion by disabbreviation. 97
Figure 4.7: Top-*N* accuracy of code completion by extrapolation in predicting (a) the next one keyword-connector pair and (b) the next two keyword-connector pairs. 99
Figure 4.8:The histogram of the number of keywords expanded from abbreviated input per code completion. 100
Figure 4.9: Code lines used in the user study to measure time usage and keystrokes needed for code-writing using two code completion systems. 103
Figure 4.10:Time usage average of all code lines for each subject. 105
Figure 4.11: Time usage average of all subjects for each code line. 105
Figure 4.12:Keystrokes average of all code lines for each subject. The baseline keystrokes, also an average of all code lines, are shown as a dotted line. 106
Figure 4.13:Keystrokes average of all subjects for each code line. The baseline keystrokes are shown as dotted lines. 106

List of Tables

Table 3.1: The number of program bugs found in the 7 open source projects. 47
Table 3.2: Precision and recall accuracy of the bug detection method.. 50
Table 3.3: Program bugs found by two different bug detection methods in the source code of Eclipse project. 52
Table 4.1: Number of unique code phrases and the number of code phrase occurrences. 73
Table 4.2: Reuse potential averaged on six open source projects. 73
Table 4.3: n-gram entropy of two English novels. 74
Table 4.4: n-gram entropy of six Java open source projects. 74
Table 4.5: The number of all possible unique KC, KCKC, and KCKCKC sequences in source code of six open source projects. K and C denote a keyword and a connector, respectively. 81
Table 4.6: An example of lexical analyzer output. 83
Table 4.7: Similarity features for estimating match probabilities. 85
Table 4.8: Estimated parameters of the logistic regression model of the match probability. 86
Table 4.9: Examples of abbreviated code lines generated using the abbreviation generator. 94
Table 4.10: Top-N accuracy of code completion by disabbreviation. 96
Table 4.11: Top-N accuracy of code completion by extrapolation in predicting next one keyword-connector pair. 98
Table 4.12: Top-N accuracy of code completion by extrapolation in predicting next two keyword-connector pairs 98
Table 4.13: Statistics of HMMs for code completion by disabbreviation. 101

Chapter 1

Introduction

Code reuse is essential for quality and productivity in software development [1]. Code reuse based on abstraction mechanisms in programming languages is a standard approach, but programmers also reuse code by taking an ad-hoc approach, in which text of code is reused without abstraction. This thesis focuses on improving two common ad-hoc code reuse approaches—*code template reuse* and *code phrase reuse*—because they are not only frequent, but also, more importantly, they pose a risk to quality and productivity in software development, the original aims of code reuse.

The first ad-hoc code reuse approach, *code template reuse* refers to programmers reusing an existing code fragment as a structural template for similar code fragments using copy-paste. Figure 1.1 shows an example of creating three similar if-statement blocks using duplicates of the first if-statement block as structural templates for writing other two if-statement blocks. Given the overhead for abstraction mechanisms, which normally require pre-planning and writing additional methods or classes, the copy-paste approach makes particular sense when code for reuse is short and reused only a few times. Code template reuse is found frequent in an empirical study [2]. In the study of copy-paste usage by programmers, it was observed that the most common copy-paste intention is to use copied code as a structural template for another code fragment.

The second ad-hoc code reuse approach, *code phrase reuse* refers to programmers reusing common code phrases by retyping them, often regularly, using code completion. Programmers are expected to retype the same or similar code phrases for several reasons. Some of such common code phases are those which programmers reuse customarily as if people use idioms in a natural language; for instance, Java programmers may think of *StringBuffer sb = new StringBuffer()* as an idiom for creating a *StringBuffer* instance and usually type the code phrase in that particular way using the same variable name and constructor. Some others are those which programmers write to follow certain usage patterns of programming resources. For example, when Java programmers use *Pattern* class, they often type *Pattern p = Pattern.compile()* and *Matcher m = p.matcher()* to follow its usage pattern. Programmers are also

forced to retype some frequent code phrases, such as *public static final int*, because many programming languages, including Java and C#, do not support phrase-level abstraction (such as a macro preprocessor in C).

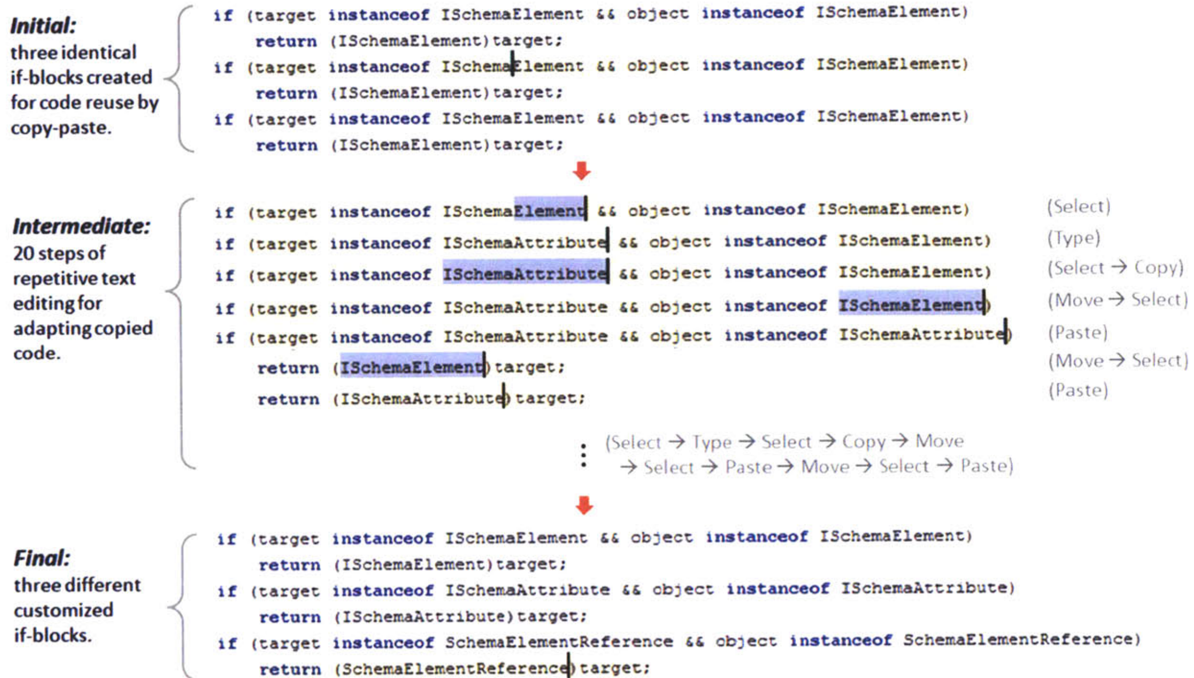


Figure 1.1: An example of code template reuse. First, an if-block is duplicated using copy-paste, resulting in three identical if-blocks. To adapt copied code blocks to serve their intended purposes, a series of editing operations are then performed. Note that the figure shows only the modified code line for each intermediate step. Finally, three differently customized if-blocks are created. The source code in the figure was adapted from the Eclipse project, an open-source Java development tool.

1.1 Problems of Ad-hoc Code Reuse

Code template reuse is frequent in programming, but its execution procedure poses a risk to software quality. Code template reuse often requires repetitive editing steps, as in the example shown in Figure 1.1, which must be performed exactly by programmers. Forgetting to perform any of the editing steps can introduce program bugs, such as the one shown in Figure 1.2, an actual program bug found in Eclipse, an open-source Java development tool[1]. Although such program bugs are introduced by trivial human errors, time spent on debugging them can be significant. An empirical study of code debugging tasks [3] found that programmers tended not to look into copied code for errors until they test many other false

[1] http://www.eclipse.org

hypotheses because they believed that their copied code was correct.

```
if (target instanceof ISchemaElement && object instanceof ISchemaElement)
    return (ISchemaElement)target;
if (target instanceof ISchemaAttribute && object instanceof ISchemaAttribute)
    return (ISchemaAttribute)target;                        ↗ SchemaElementReference
if (target instanceof SchemaElementReference && object instanceof ISchemaElement)
    return (SchemaElementReference)target;
```

Figure 1.2: A program bug found in the source code of Eclipse, which appears to have been introduced by forgetting to perform one of the repetitive text editing steps during code template reuse by copy-paste.

Code phrase reuse poses a threat to productivity. Requiring programmers to type the same or very similar code phrases over and over again certainly slow down software development. Programmers may use code completion to increase speed of typing [15]; however, conventional code completion systems are limited in that they complete only one keyword at a time. To code-complete a code phrase with multiple keywords, as shown in Figure 1.3, programmers have to repeatedly invoke code completion, review code completion candidates, and select a correct candidate as many times as the number of keywords. A system that can support efficient reuse of code phrases has much potential to improve productivity of software development. A corpus study of six Java open source projects, described in Section 3.1, found that 3 percent of the most frequent 3-code-token phrases account for 36 percent of total occurrences of 3-code-token phrases.

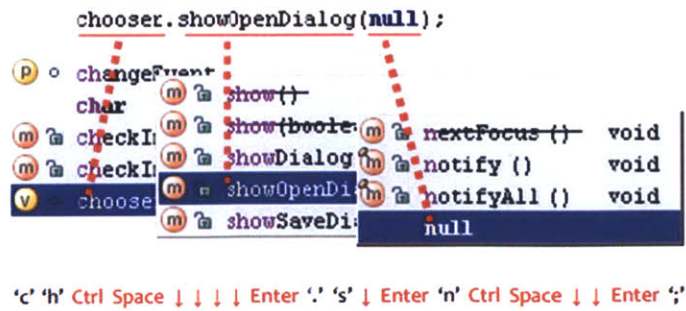


Figure 1.3: In conventional code completion, code completion of code phrases might require many extra keystrokes for driving code completion dialogs.

1.2 Solution Approaches

This thesis aims to improve effectiveness of ad-hoc code reuse approaches based on sequence-based code

token analysis, spreadsheet-inspired user interaction, and statistical models of frequent code token phrases.

1.2.1 Effective Code Template Reuse

To addresses the quality risks posed by code template reuse, a new bug detection method for automatically locating code editing errors in existing structurally equivalent code fragments is developed (Chapter 3). There are two key insights enabling the bug detection approach: firstly, differences of structurally equivalent code fragments can be compactly expressed using a set of code token sequences crossing the code fragments; and secondly, program bugs can be accurately located by suspiciously inconsistent patterns within a set of code token sequences. To proactively reduce program bugs introduced during code template reuse, this thesis also introduces an error-preventive code editing method that can reduce the number of code editing steps. The new code editing method takes a cell-based text editing approach to speed selection of editing targets and introduces a code generation user interface to further reduce the number of editing steps for code template reuse.

The effectiveness of the new bug detection method is evaluated on the basis of the number of detected program bugs and the ratio of actual bugs to bug warnings (precision). In 7 well-maintained open source projects, including Eclipse, NetBeans, and JRuby, a total of 87 program bugs were found; many of them were program bugs that could not be detected by previous methods for copy-paste-related bug detection [7,8,9]. Bug detection precision in Eclipse was 51%, measured as the ratio of the number of true bugs to the number of bug warnings, and is 9 times higher than a previous method [9]. The new code editing approach is compared with normal text editing on the basis of estimated editing time in two code template reuse scenarios randomly selected from open source projects. Large savings in estimated editing time were observed when using the cell-based text editing approach; this is promising because such savings can help programmers stay focused and make fewer errors during code template reuse.

1.2.2 Effective Code Phrase Reuse

To the productivity limitation posed by code phrase reuse, a new code completion method that can infer common code phrases from non-predefined abbreviated input using a statistical model learned from a corpus of code and abbreviation examples (Chapter 4). The code phrase completion method completes multiple code tokens at a time by taking non-predefined abbreviated input and expands it into a full code phrase. To accelerate code phrase completion even further, the code completion method is also extended

to support prediction of the next code tokens following a code completion candidate, a technique called code completion by extrapolation.

A key insight underlying this statistical-model-based approach to code completion is from our observation (Section 4.2.2) that source code in a programming language is statistically as predictable as, or even more predictable than, text in a human natural language; therefore a well-engineered statistical model of source code can infer idiomatic code phrases from a small but essential amount of ambiguous input—such as non-predefined abbreviation—just like statistical models of English words enable speech recognition from noisy voice signals and enables auto-complete and auto-correction for text entry on mobile devices.

The effectiveness of new code completion method is evaluated on the basis of accuracy and efficiency. Top-*N* accuracy, the rate of predicting a correct code completion as one of the *N* most likely code completion candidates, is measured in a corpus study. The code completion method achieved 99.3% of top-10 accuracy on average against 4919 code phrases sampled from six open source projects. Time and keystroke savings of the new code completion method are evaluated in a user study. Time savings and keystroke savings were measured by comparing time usage and the number of keystrokes of the new code completion method with those of a conventional code completion method. The system achieved average 30.4% savings in time and 40.8% savings in keystrokes in a user study with eight participants.

1.3 Key Contributions

This work develops new models, algorithms, and user interfaces for effective ad-hoc code reuse by addressing limitations in previous methods for copy-paste-related bug detection and code completion. The key contributions of this thesis are the following:

- Development of *Bugsy Finder* (Buggy-sequence Finder), a bug detection method based on a novel sequence-based model of similar code fragments. The bug detection method found many previously unknown program bugs in well-maintained open source projects and achieved a significant improvement in precision compared to previous methods for copy-paste-related bug detection.
- Development of *Texel Editing* (Text-cell Editing), a novel cell-based text editing method that incorporates a code generation user interface optimized for code template reuse. The code editing method aims to prevent repetitive text editing errors by reducing the number of editing steps

required for code template reuse.

- Development of *Abbreviation Completion*, a novel code phrase completion method for accelerating code phrase reuse by typing and code completion. The new method completes multiple code tokens at a time based on non-predefined input, utilizing frequent keyword patterns learned from a corpus of code and abbreviation examples. It achieves high accuracy as well as significant time and keystroke savings by addressing limitations in previous code completion methods.

1.4 Thesis Outline

The rest of this dissertation is organized according to the aforementioned key contributions. Chapter 2 discusses related work on ad-hoc code reuse, including copy-paste-related bug detection, simultaneous text editing, and multiple-keyword code completion, to discuss how this thesis work contributes to other related fields of research. Chapter 3 presents a new bug detection method. A sequence-based source code model, a bug detection algorithm, and evaluation of the bug detection method are described. Chapter 3 also introduces a new code editing method for reducing repetitive text editing. Chapter 4 presents a new code phrase completion method; algorithms for efficient code phrase completion and its evaluation based on a corpus study are described. Chapter 5 concludes and describes interesting future directions for improving ad-hoc code reuse.

Chapter 2
Related Work

This chapter surveys related work to ad-hoc code reuse. It first presents an overview of previous work in six related fields of research: some provide empirical findings that motivated this thesis work while some others present different approaches to similar problems this thesis aims to address. Then it discusses limitations of three categories of previous work with further details and examples as they are closely related to new methods developed in this thesis for effective ad-hoc code reuse.

2.1 Overview

The thesis work primarily builds on and contributes to the following six fields of research:

First, the work builds on empirical studies of copy-paste usage by programmers [2,3]. The empirical studies have shown that programmers perform copy-paste-based code reuse frequently with justifiable reasons. Kim et al. discovered through interviews with programmers, asking intention of copy-paste usages, that the most common copy-paste intention is to use copied code as a structural template for another code fragment [2]. Ko et al. [3] also found that program bugs in copy-pasted code are costly to fix although they are caused by a trivial text editing error. In a debugging task that had a program bug in copy-pasted code, programmers tended not to look into copied code until they test many other false hypotheses because they believed that their copied code was correct. These findings motivate the development of systems that can facilitate code template reuse, which is typically performed using copy-paste.

Second, the work builds on research work in the field of code clone detection [4,5,6], which has developed various methods for finding similar code fragments in a large code base. Our bug detection method builds on techniques used in token-based clone detection systems, but has been optimized for finding structurally equivalent, adjacent code fragments.

Third, several copy-paste-related bug detection methods [7,8,9] have been developed based on the idea that inconsistency within siblings of code clones indicates a program bug. The previous methods developed counter-based heuristics, such as comparing the number of unique identifiers in code lines or calculating the ratio between the numbers of changed and unchanged code tokens, to detect inconsistent modification of code clones. Our method for finding program bugs also detects inconsistency, but it uses a novel sequential-pattern-based approach and integrates a sophisticated false positive pruning mechanism. As a result, the new bug detection method detects more program bugs with higher precision than previous methods.

Fourth, the work is related to a field of research on improving code completion [16,17,18,19,20]. Our code phrase completion system, Abbreviation Completion, presents a new method for supporting multiple-keyword code completion from abbreviated input. When compared to other multiple-keyword code completion systems that outputs a multiple-line code snippet primarily based on type constraints, our system outputs a single-line code phrase based on statistical model of keyword transitions built from a corpus of code and achieves much higher accuracy, which is important for practical use.

Fifth, the algorithm for appending the most likely code tokens to a code phrase completion candidate builds on previous work on N-gram-based word prediction in natural languages [27]. We developed a new N-gram-based model of programming languages and a depth-limited beam search algorithm in order to apply the previous work to our problem domain of code phrase completion.

Sixth, the cell-based text editing method builds on existing work on simultaneous text editing [10, 11]. Simultaneous text editing, which is characterized by the use of multiple selections in text editing, can reduce repetitive text editing errors because users can apply a text editing command on multiple selections simultaneously. Because manually specifying multiple selections is time-consuming and error prone, it is important to support efficient multiple-selection for a simultaneous editing approach to be effective. Unfortunately, existing selection methods are not optimal for common selection tasks during code template reuse, as described in Section 2.3. The cell-based text editing aims to address the challenge (Section 3.7).

2.2 Copy-paste-related Bug Detection

Previous work on copy-paste-related bug detection focuses on addressing a particular type of human error in code template reuse that programmers may fail to apply consistent modification to all copy-pasted code

fragments because they may not know or recall where all the copy-pasted code fragments are located. Building on top of code clone detection tools that can locate similar code fragments, called code clones, which may be scattered in a code base [4,5,6], copy-paste-related bug detection tools aim to detect inconsistency within a set of code clones [7,8,9]. Previous methods for copy-paste-related bug detection could find some of the program bugs accidentally introduced in code template reuse by copy-paste, but they have two limitations that make the methods ignore many potential bugs and report many false bug warnings.

The first limitation is related to a configuration of code clone detection tools used with the bug detection methods. Based on a potentially questionable assumption that code fragments that are shorter than a certain length are not likely to have copy-paste-related errors [7,8,9], code clone detectors in the previous work imposed a certain minimum token count requirement when reporting similar code fragments, typically a minimum of 30 code tokens, and ignore smaller similar code fragments. As a result, the previous bug detection methods will not even try to detect program bugs in Figure 2.1, an actual program bug found in JBoss Seam, an open source web application framework, because the similar code fragments are only 8 code tokens in length.

```
                                                                                  getStartColumn
int useStartColumn = uiImage.getStartColumn() == null ? currentColumnIndex : uiImage.getStartRow();
int useStartRow = uiImage.getStartRow() == null ? currentRowIndex : uiImage.getStartRow();
```

Figure 2.1: An example of program bugs in short similar code fragments with less than 30 code tokens, which will be missed by bug detection methods in previous work.

Even if previous bug detection methods are configured to analyze smaller code clones, they may still fail to detect bugs or report too many false bugs because of the second limitation on metrics used for measuring inconsistency. Previous methods are based on count-based metrics, such as difference in the number of unique identifiers or the ratio of the number of occurrences that a given identifier is unchanged and the number of total occurrences of the identifier, called an unchanged ratio; when the counter-based metrics exceed certain thresholds, bug warnings are reported. Count-based metrics are inherently prone to false positive or false negative errors because they tend to aggressively summarize intended differentiations and unintended inconsistencies in similar code fragments. For instance, Figure 2.2 shows an example of a program bug that cannot be detected by previous methods. Difference in the number of unique identifiers is zero, which is interpreted as consistent modification in [9]; the unchanged ratio of *setDiffuse* is one, which is interpreted as consistent invariance in [7]. Due to the limitation of count-based metrics, precision of previous bug detection methods is low, ranging between 5% and 10%, indicating that

programmers are expected to inspect 10 to 20 program bug candidates to find an actual program bug.

```
Node diffuseNode = getChildNode(light, "colourDiffuse");
if (diffuseNode != null)
    l.setDiffuse(loadColor(diffuseNode));

Node specularNode = getChildNode(light, "colourSpecular");
if (specularNode != null)
    l.setDiffuse(loadColor(specularNode));
        setSpecular
```

Figure 2.2: An example of program bugs that cannot be detected by previous methods using count-based metrics.

2.3 Simultaneous Text Editing

Simultaneous text editing can help to reduce repetitive text editing errors by allowing users to apply a text editing command on multiple selections in a single editing step [10]. Given that the text editing approach makes sure that a consistent editing operation is applied to all selected text regions by design, an important challenge is to make sure that all necessary editing targets are selected before invoking an editing command.

Because manually specifying multiple selections can be time-consuming and error-prone, Miller and Myers developed a powerful inference algorithm for generalizing multiple selections from a small number of selection examples [10]. The inference-based approach is particularly effective when similar selections need to be made on many dozens or hundreds of structurally similar text regions, such as when all text regions for the year in bibliographic citations need to be selected.

Unfortunately, the inference-based selection approach is not directly applicable to specifying multiple selections for code template reuse. In code template reuse, multiple selections are normally made on words or sub-words—a sub-word refers to a word in a compound word, such as *max* in *maxWidth*—with the same text. Those selected identifiers or sub-words may not have inferable structural similarities. For example, in Figure 2.2, all five occurrences of sub-word *diffuse* in the upper code fragment need to be selected and replaced with *specular* to complete the lower code fragment. All occurrences of the sub-word have the same text, but no structural similarity can be found. Actually, selecting the five sub-word selections using LAPIS, an experimental text editor implementing the inference-based approach, requires an equivalent amount of keyboard or mouse operations as doing it manually, as can be seen in Figure 2.3.

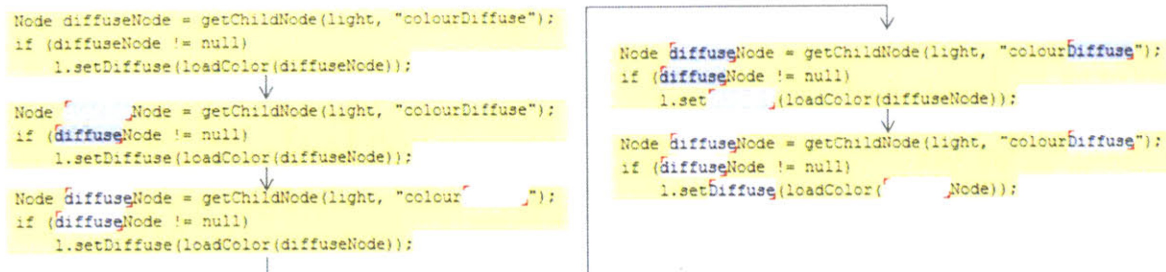


Figure 2.3: An inference-based approach to multiple-selection specification based on their structural similarity is not directly applicable to multiple selections for code template reuse. In many cases, there exists no structural similarity among identifiers or sub-words that need to be selected for code template reuse, except that they have the same text.

2.4 Code Completion

Code completion is used by many programmers [15] to accelerate code-writing by avoiding typing the whole character sequence of keywords. The next two subsections survey two different styles of code completion, one that completes one keyword at a time and the other that does multiple keywords at a time. Then it surveys text entry methods developed for accelerating natural language input to discuss how they are related to a code phrase completion method developed in this thesis.

2.4.1 Conventional, Single-keyword Code Completion

Conventional code completion has three notable limitations. First, code completion completes only one keyword at a time. Therefore, the number of extra keystrokes increases in proportion to the number of completed keywords. For example, a typical code completion interaction, like the one shown in Figure 1.3, might take 20 keystrokes to complete three keywords of 29 characters. Second, code completion finds a keyword based on an exact match of leading characters. Because the leading parts of keywords are often identical among candidates while the ending parts are more distinguishable, as in *showOpenDialog* and *showSaveDialog*, programmers often need to type potentially lengthy sequences of leading characters before they type distinctive characters close to the end. Third, code completion normally puts the default selection of code completion candidate on the first candidate in an alphabetically sorted candidate list. It leads to additional Up/Down Arrow keystrokes to adjust the selection to a correct candidate.

To address the third limitation of conventional code completion, researchers worked on improving the ordering of code completion candidates [19,20]. Effective prioritization of code completion candidates may not only reduce the number of keystrokes required for navigating to a correct code completion candidate, but also save time and effort spent on examining many code completion candidates. Prototype

systems in [19] and Mylyn [20] explore the utility of the change history and the task context, respectively, as additional sources of information for prioritizing code completion candidates. The multiple-keyword code completion method developed in this thesis explores the utility of keyword sequences extracted from a corpus of source code for prioritizing multiple-keyword candidates.

2.4.2 Multiple-keyword Code Completion

Saving keystrokes and time for code-writing is one of the major design objectives of source code editors. Generating multiple keywords from a short character sequence is one way of achieving the objective. There have been two major approaches for supporting multiple keyword generation.

The first approach is based on a code template, a predefined code fragment that can be inserted into the code editor. Each code template is given an alias, such as *sysout*, so that programmers can insert a code fragment using the alias as a reference. Many code editors, including Emacs [13] and Eclipse [14], implement this approach. The code template approach is effective at handling a handful of very frequently reused code phrases. However, the burden of memorizing aliases can put a limit on the number of code phrases a user can actually reuse. The time-consuming process of adding new code templates may also limit its usage.

The second approach is based on a type-constrained search that can construct programming expressions containing multiple keywords. Keyword Programming [16], XSnippet [17], and Prospector [18] implement this approach, but they have differences in the type of input queries and output expressions and the kind of heuristics for guiding type-constrained search. Prospector takes two Java types as an input and outputs code lines for converting one type to the other type using heuristics based on the graph path. XSnippet also takes two Java types as an input, but optionally it can take additional Java types to specify a context. XSnippet uses heuristics based on snippet lengths, frequencies, and context matches to generate multiple lines of code for instantiation. Keyword Programming takes a set of keywords as input and outputs a code line of Java expression using heuristics based on the keyword matches. Type-constrained search systems focus on serving a specific type of users who need help with choosing or using classes. They do not serve a different type of user who already has a good idea of what needs to be written and wants to write it more efficiently.

A code template system is a kind of multiple-keyword code completion system that allows programmers to quickly insert a blob of code by expanding a predefined short alias into a predefined longer code fragment, called a code template, corresponding to the that alias. For example, if *sysout* is a predefined

alias for code fragment *System.out.println()*, a programmer can insert the code fragment by typing *sysout* and invoking a code template system. Many code editors, including Emacs [13] and Eclipse [14], implement code template systems. Code template systems are effective at handling a handful of very frequently reused code phrases. However, the burden of memorizing aliases can put a limit on the number of code phrases a user can actually reuse. The time-consuming process of adding new code templates may also limit its usage.

2.4.3 Text Entry Systems for Natural Language Input

Text entry systems for natural language input also explored the use of abbreviations as a means to improve efficiency of text entry [23] [24]. A system proposed by Shieber and Nelken [23] is the closest to our own in that it incorporates a language model that aims to capture frequent word transition patterns. However, their system supports a particular, predefined type of abbreviations, which is constructed by applying strict word compression rules. Their user study found that users could unknowingly make small deviations from the strict rules, which resulted in disabbreviation failures. Our approach addresses such a problem by supporting non-predefined abbreviations and thus allowing users to abbreviate keywords any way they like. Actually, the Abbreviation Completion algorithm was applied to text entry for mobile devices and achieved considerable time and keystroke savings, 26% and 32 % [30]. The use of an *n*-gram language model or a suffix-tree-based language model to predict the next words was explored by Bickel et al. [25] and Nandi and Jagadish [26]. Our *n*-gram model for extrapolating a code completion candidate builds on the previous work and introduces programming-language-specific extensions to predict both keywords and non-alphanumeric characters.

Chapter 3

Effective Code Template Reuse: Automated Bug Detection and Error-preventive Code Editing

Chapter 1 has mentioned that code template reuse poses a risk to quality because of repetitive text editing errors during the code reuse. This chapter presents a bug detection method that can locate repetitive text editing errors and address the risk in existing code. To prevent further repetitive text editing errors in new and modified code, this chapter also presents a new code editing method that can reduce the number of editing steps required for code template reuse.

3.1 Overview

3.1.1 Automated Bug Detection

This thesis develops a new bug detection method, called *Bugsy Finder*, which can find program bugs introduced by code editing errors during code template reuse. It not only finds program bugs not found by prior methods, but it also achieves high precision, which promises the development of a practical tool for automated bug detection. Figure 3.1 shows a usage scenario of the current implementation of the bug detection method:

1. Specify the location of source code.

2. It takes about 31 min to analyze 2.2M lines of source code of Eclipse.

3. 49 bug warnings are reported. Manual verification finds 25 true bugs.

left sequence (normalized)	right token sequence (normalized)	occurrences	ratio	length	left line no.	right line no.	file
[repository decorator]	[repository-id repository-id]	2	2	1	2	65	65 C:\eclipse
[move copy]	[drop-move drop-move]	4	2	2	2	92	92 C:\eclipse
[move copy]	[move-button move-button]	4	2	2	2	92	92 C:\eclipse
[cannot-open-file interface-not-f	[cannot-open-file interface-not-founc	1	1	1	9	303	303 C:\eclipse
[container declaration]	[import-declaration import-declaratio	2	2	1	2	3153	3153 C:\eclipse
[type method field-initi	[illegible]	[illegible]	1	1	3	1144	328 C:\eclipse
[1-3 1-4 1-5 1-6]	[illegible]	2	[illegible]	[illegible]	[illegible]	[illegible]	4239 C:\eclipse
[ws arch nl]	[ws arch arch]	2	[illegible]	[illegible]	[illegible]	[illegible]	534 C:\eclipse
[tool project]	[illegible]	2	[illegible]	[illegible]	[illegible]	[illegible]	92 C:\eclipse
[push radio check]	[illegible]	3	3	1	3	124	124 C:\eclipse
[1-1 1-2 1-5 1-6 1-7]	[1-3 1-3 1-5 1-6 1-7]	1	1	1	5	3410	3410 C:\eclipse
[1-6 1-5 1-2 1-1]	[1-6 1-5 1-3 1-3]	1	1	1	4	784	784 C:\eclipse
[add remove]	[add add]	2	2	1	2	50	50 C:\eclipse
[add remove]	[add add]	2	2	1	2	165	165 C:\eclipse
[background foreground]	[background background]	6	2	3	2	227	227 C:\eclipse
[check full-selection]	[check check]	6	2	3	2	412	412 C:\eclipse

Figure 3.1: A usage scenario of the new bug detection method. First, a user specifies the location of source code for analysis. All source code files under the specified directory are analyzed. Second, the user starts the bug detection process. In case of analyzing source code of Eclipse, a Java Development Tool, which contains 2.2 million lines of code, it takes about 31 minutes to complete the bug detection process. Finally, bug warnings are reported to the user in a spreadsheet file. The user manually verifies whether each bug warning is a true program bug or a false warning. A total of 49 bug warnings are generated by running the bug detection method on the source code of Eclipse. Manual verification of bug warnings finds that 25 of them are true program bugs and 24 are false warnings, which indicates that precision of bug detection, the ratio of actual bugs to bug warnings, is 51% in this case.

Key contributions of the new bug detection method are:

- First, a simple yet novel hypothesis on the source of program bugs: repetitive text editing error during code template reuse may be a significant source of program bugs. This hypothesis guides the development of a bug detection method that can find different types of program bugs than previous methods [7,8,9].
- Second, an empirical support of the hypothesis: the new bug detection method found 87 previously unknown program bugs in 7 well-maintained open source projects. The large number of program bugs provides an empirical support for the substantial quality risk posed by code editing errors during code template reuse.
- Third, an accurate bug detection method for finding program bugs introduced during code template reuse: the new bug detection method locates suspicious code fragments by analyzing sequential patterns of code tokens, instead of counter-based heuristics of previous methods. A

new source code model, called sequence-driven code, is introduced to facilitate the sequence-based analysis of code tokens.

3.1.2 Error-preventive Code Editing

A bug detection method is only useful for locating program bugs already existing in code, but it does not help with preventing program bugs in new code. This thesis also introduces a new code editing method for proactively reducing program bugs during code template reuse. Figure 3.2 shows a usage scenario of the new code editing method:

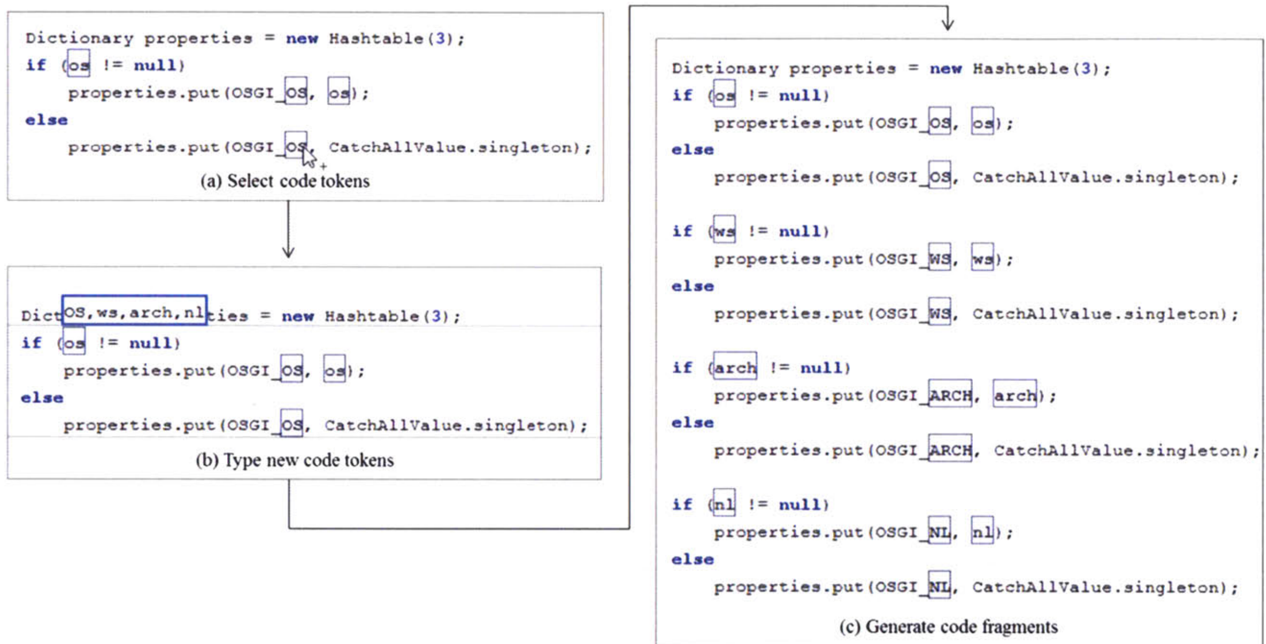


Figure 3.2: A usage scenario of the new code editing method for reducing code editing steps for code template reuse. (a) A user selects code tokens that need to be different in each similar code fragment; the user selects code tokens using Ctrl+Click in a similar way to selecting multiple items in a spreadsheet or a file browser. (b) Once selection is complete, the user types the new code tokens that will replace the selected code tokens in generated code fragments. A comma is used as a separator between new code tokens. (c) Finally, the user presses the Enter key to generate three additional code fragments based on the selections and new code tokens. In this example, the new code editing approach completes the task in 4 clicks and 13 keystrokes. Note that the same code editing task might take 124 keystrokes if done using normal text editing.

Two key contributions of the new code editing method are:

- First, an efficient multiple-selection user interface: the new user interface supports cell-based text editing to optimize keyboard and mouse interactions for common selection tasks during code template reuse.
- Second, an efficient code generation user interface for code template ruse: the new user interface

allows programmers to generate sequence-driven code by selecting code tokens that need to be different in each similar code fragment and typing a list of code tokens that will replace the selected code tokens in new similar code fragments.

The next section introduces a new hypothesis on the source of program bug as well as properties of potential program bugs. Then a sequence-based source code model and bug detection algorithms are described. The subsequent two sections present implementation and evaluation of the bug detection method. The next section introduces new user interfaces for efficient code template reuse, followed by a predictive evaluation and discussions.

3.2 Hypothesis

This section first introduces a new hypothesis on the source of program bugs: code editing errors during code template reuse may be a significant source of program bugs. Then it derives properties of program bugs introduced by common text editing errors during code template reuse.

3.2.1 New Hypothesis on the Source of Program Bugs

Prior work on locating code clones and detecting program bugs in code clones was developed based on a hypothesis that inconsistent evolution of similar code fragments causes program bugs in copy-pasted code [7,8,9]. This thesis investigates a different hypothesis related to copy-pasted code that:

Code editing errors during code template reuse may be a significant source of program bugs.

To provide an empirical support for the hypothesis, this thesis aims to locate a significant number of program bugs in well-maintained open source projects. The simple yet novel hypothesis is important because it guides the development of a new bug detection method that can find different kinds of program bugs than previous methods.

3.2.2 Common Editing Errors during Code Template Reuse

This section discusses two types of common editing errors during code template reuse, which will be used for deriving heuristics for automatic bug detection. The first type of editing error is to forget to replace some code tokens, which will be referred to as *forget-to-replace* error; the second is to replace a code token with a wrong one, referred to as *replace-with-wrong*.

Figure 3.3 shows an example of forget-to-replace error found in source code of JRuby 1.1.5, an open

source project implementing Ruby language in Java, in SkinnyMethodAdapter.java file. Although 6 methods are shown in the figure, there are a total of 53 methods, which are located just before and after the code shown, with the same token structure. The code essentially defines a mapping from methods of the SkinnyMethodAdapter class to corresponding virtual machine operation codes. Given that all methods have the same code token structure, but are only different by two code tokens, it is highly likely that a programmer wrote this code through code template reuse.

Unfortunately, it appears that the programmer forgot to replace a code token in one of the 53 methods. After duplicating the fourth method in the figure, *f2d()*, to use it as a template for subsequent similar methods, the programmer probably intended to replace two code tokens, *f2d* and *F2D*, in the first copy with *f2i* and *F2I*. However, the programmer actually replaced only one code token and introduced a program bug as a result. No compilation error is raised because both constants, *F2D* and *F2I*, are valid in the program context. This program bug is particularly difficult to detect through test cases because its buggy behavior is almost invisible, only affecting memory usage and speed. The program bug causes the float-to-double conversion (*F2D*) to be performed when float-to-integer (*F2I*) conversion is requested. It may not cause any loss of numerical precision, but may have a negative impact on memory usage and speed.

```
public void fdiv() {
    getMethodVisitor().visitInsn(FDIV);
}

public void frem() {
    getMethodVisitor().visitInsn(FREM);
}

public void fneg() {
    getMethodVisitor().visitInsn(FNEG);
}

public void f2d() {
    getMethodVisitor().visitInsn(F2D);
}
                                         F2I
public void f2i() {
    getMethodVisitor().visitInsn(F2D);
}

public void f2l() {
    getMethodVisitor().visitInsn(F2L);
}
```

Figure 3.3: The first type of common code editing error: forget-to-replace. In this example, it appears that a programmer forgot to replace one of the two code tokens in the fifth method. The program bug was found in SkinnyMethodAdapter.java file of JRuby1.1.5.

The second type of error, replace-with-wrong, is expected to occur with the use of code completion. Many programmers use code completion to save keystrokes, as well as to reduce potential typing errors [15]. Programmers are expected to use code completion during code template reuse. When programmers replace a code token in duplicated code with a new code token, they may complete the new code token using code completion, instead of manually typing it. Unfortunately, programmers may accidentally insert a wrong code token because method or variable names are sometimes similar to each other—programmers may mistake one for another and end up with replacing the code token with a wrong one.

An example of the replace-with-wrong error is shown in Figure 3.4, which was found in source code of JBoss Seam 2.2.1, an open-source web application framework, in MailExternalContextImpl.java file. The figure shows 6 of 11 similar methods for delegating a method call to one class to another. A code editing error exists in the last method in the figure, in which a wrong method name *getRequestHeaderValuesMap* is inserted after *delegate.* instead of *getRequestParameterValuesMap*. It appears that a programmer used code completion to replace a duplicated method name with a new method name. The programmer probably intended to insert *getRequestParameterValuesMap*, but he or she mistakenly chose *getRequestHeaderValuesMap* from code completion candidates because the two method names were relatively long and looked similar. Note that there is a small possibility that the program bug may have been caused by the forget-to-replace error; the second method, which uses *getRequestHeaderValuesMap*, could have been used as a template for the last method. However, given the long vertical distance between the second and last methods—five methods between them—the replace-with-wrong error is more likely the case.

```
public Map getRequestHeaderMap()
{
  return delegate.getRequestHeaderMap();
}

public Map getRequestHeaderValuesMap()
{
  return delegate.getRequestHeaderValuesMap();
}

                    ⋮ (2 similar methods omitted)

public Map getRequestMap()
{
 return delegate.getRequestMap();
}

public Map getRequestParameterMap()
{
  return delegate.getRequestParameterMap();
}

public Iterator getRequestParameterNames()
{
  return delegate.getRequestParameterNames();
}

public Map getRequestParameterValuesMap()
{
  return delegate.getRequestHeaderValuesMap();
}
                   ↳getRequestParameterValuesMap

                    ⋮ (2 more similar methods)
```

Figure 3.4: The second type of common code editing error: replace-with-wrong. In this example, it appears that a programmer used code completion to replace a method name right after *delegate.* with a new method name. The programmer probably intended to insert *getRequestParameterValuesMap*, but he mistakenly chose *getRequestHeaderValuesMap* from code completion candidates because the method names were relatively long and looked similar.

3.2.3 Characterization of Program Bugs Caused by Common Editing Errors

This section presents properties of program bugs caused by common editing errors. It is virtually impossible to generalize all relevant properties of program bugs just from the two common types of code editing errors. The properties of program bugs were actually developed through iterative process, in the first iteration of which the two common types of code editing errors were used to derive potentially naïve properties of program bugs. The first version of properties was used to develop the first version of bug detection method, which fortunately found a small number of program bugs that appeared to have been introduced by code editing errors. Through the iterative process, program bugs found by different versions of bug detection methods were accumulated, and they were used to derive more relevant properties of program bugs. Section 3.4.2 presents a detailed discussion of the iterative development

process.

The following four properties were derived to guide the development of a bug detection method:

P1. **Structural equivalence of similar code fragments:** similar code fragments have the same code token structure. That is, all non-alphanumeric characters and programming language reserved keywords in one code fragment appear at equivalent locations in other similar code fragments. Alphanumeric code tokens may also appear commonly in all similar code fragments, but there are always one or several alphanumeric code tokens that are different from one code fragment to another.

P2. **Adjacency of similar code fragments:** similar code fragments generated by code template reuse are located next to each other. Although programmers may sometimes copy a code fragment and paste it into a different file or a disjointed code region in the same file, programmers tend to put similar code fragments adjacently because of the syntactic constraint of program constructs, as in if-else or case-switch statements, or for the convenience of code writing and maintenance. Note that this property is important because it allows us to locate short similar code fragments accurately with a small number of false positives.

P3. **A small number of unique code token sequences:** code tokens that are located at equivalent locations, but are different from each other can be viewed to form a code token sequence across the code fragments. The number of unique code token sequences tends to be small, 1 or 2 in most cases. The number of unique code token sequences does not double-count code token sequences that are equivalent. A pair of code token sequences is considered equivalent if their tokens match at most indexes (if the number of mismatched indexes is smaller than a threshold) after normalizing capitalization and trimming a common prefix and suffix.

P4. **A small number of duplicate code tokens:** Code token sequences extracted from similar code fragments with a forget-to-replace or replace-with-wrong error always include at least one code token sequences with all different code tokens (all-different sequence) and at least one code token sequence with duplicated code tokens (some-duplicates sequence). The number of extra duplicates is calculated by subtracting the number of unique code tokens from the number of code tokens. The number of extra duplicates tends to be small, 1 or 2 in most cases. For example, code token sequences extracted from the buggy code fragments in Figure 3.3 include one all-different sequence, (fdiv frem fneg f2d f2i f2l), and one some-duplicates sequence (FDIV FREM FNEG F2D F2D F2L). The number of extra duplicates is 1, calculated by subtracting 5 from 6. Note that

the replace-with-wrong error also introduces duplicate code tokens because there may be a similar code fragment that uses the same wrong code token in a correct consistent way, as in the second method in Figure 3.4.

3.3 Model and Algorithm

This section describes models and algorithms to locate source code fragments satisfying the four properties of program bugs introduced in the previous section. Models and algorithms are generally applicable to any text-based programming language although they will be introduced using code examples in the Java language.

3.3.1 Sequence-based Source Code Model

Sequence-driven code refers to structurally equivalent code fragments that are adjacent to each other. As mentioned in the third property of the program bug, similar code fragments in sequence-driven code are only different by code token sequences crossing the code fragments. Figure 3.5 shows an example of sequence-driven code consisting of three structurally equivalent code fragments that are different by two code token sequences—*(top middle bottom)* and *(TOP MIDDLE BOTTOM)*. By normalizing capitalization of the code token sequences and trimming their common prefixes and suffixes when it is possible, code token sequences may be merged into a smaller set of code token sequences. For instance, the two code token sequences with different capitalization in the figure can be merged into a single code token sequence, *(top middle bottom):2*. The number of occurrences of a merged code token sequence is annotated on the merged code token sequence and is displayed in a small circle (Figure 3.5). This way, sequence-driven code compactly summarizes the differences of similar code fragments using a set of code token sequences and the number of occurrences of each code token sequence.

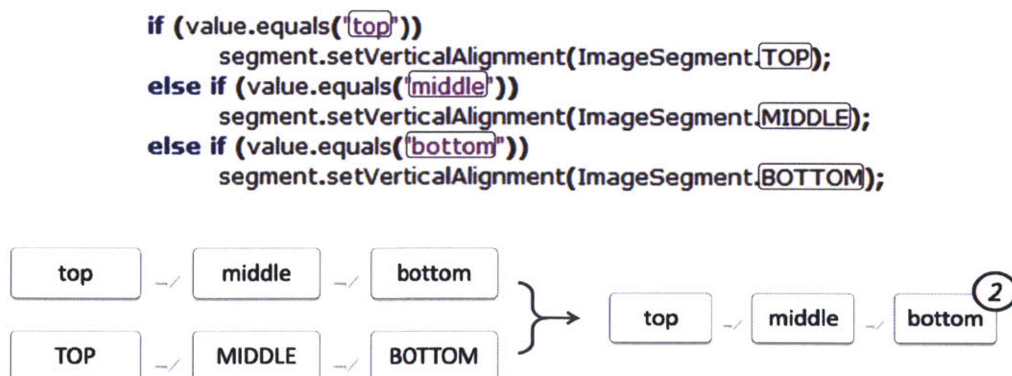


Figure 3.5: An example of sequence-driven code consisting of three structurally equivalent, adjacent code

fragments.

3.3.2 Target Code Regions for the Search of Sequence-driven Code

To find sequence-driven code, we analyze the following code regions, which are common targets for code template reuse. The code regions are selected based on our observation that sequence-driven code usually occurs with certain language constructs. Note that the syntax of the language constructs may vary depending on the programming language, but the language constructs are common in many programming languages.

- **A *switch-case* statement:** The *case* blocks in a *switch-case* statement always occur adjacently. When several *case* blocks have the same code token structures, programmers may take a code template reuse approach to complete the code. An example is shown in Figure 3.6.

```
switch (keyCode) {
    case SWT.ARROW_DOWN :
        AbstractSectionForm.scrollVertical(scomp, false);
        break;
    case SWT.ARROW_UP :
        AbstractSectionForm.scrollVertical(scomp, true);
        break;
    case SWT.ARROW_LEFT :
        AbstractSectionForm.scrollHorizontal(scomp, true);
        break;
    case SWT.ARROW_RIGHT :
        AbstractSectionForm.scrollHorizontal(scomp, false);
        break;
}
```

Figure 3.6: An example of sequence-driven code found in a *switch-case* statement.

- **An *if-else-if* statement:** The *if* and *else-if* blocks in an *if-else-if* statement always occur adjacently. When an *if* block is followed by several *else-if* statements that are structurally equivalent except the *else* keyword, programmers may write the code using code template reuse. An example is shown in Figure 3.7.

```
if (value.equals("top"))
    segment.setVerticalAlignment(ImageSegment.TOP);
else if (value.equals("middle"))
    segment.setVerticalAlignment(ImageSegment.MIDDLE);
else if (value.equals("bottom"))
    segment.setVerticalAlignment(ImageSegment.BOTTOM);
```

Figure 3.7: An example of sequence-driven code found in an *if-else-if* statement.

- **Consecutive *if* statements:** Programmers may write code that conditionally performs some, not just one, of the tasks. In such cases, programmers may write consecutive *if* statements, like the

one shown in Figure 3.8 using code template reuse.

```
if (element.getOS() != null && !element.getOS().equals(Config.ANY)) {
        for (Iterator iter = result.iterator(); iter.hasNext();) {
                Config config = (Config) iter.next();
                if (! isMatching(element.getOS(), config.getOs()) )
                        iter.remove();
        }
}
if (element.getWS() != null && !element.getWS().equals(Config.ANY)) {
        for (Iterator iter = result.iterator(); iter.hasNext();) {
                Config config = (Config) iter.next();
                if (! isMatching(element.getWS(), config.getWs()) )
                        iter.remove();
        }
}
```

Figure 3.8: An example of sequence-driven code found in consecutive if statements.

- **Consecutive method declarations:** Programmers may write a list of methods that perform almost the same operations except that they are implemented using different variables, methods, string or numeric literals, or classes. For the convenience of code writing and code maintenance, programmers usually place the list of methods together, as shown in Figure 3.9.

```
public void ishr() {
    getMethodVisitor().visitInsn(ISHR);
}

public void ishl() {
    getMethodVisitor().visitInsn(ISHL);
}

public void iushr() {
    getMethodVisitor().visitInsn(IUSHR);
}
```

Figure 3.9: An example of sequence-driven code found in consecutive method declarations.

- **Consecutive statement expressions:** A statement expression refers to an assignment expression or primary expression ending with a statement terminator, a semicolon in Java. Sometimes, a few lines of statement expressions are reused as a template for similar code fragments. In an example shown in Figure 3.10, three lines of statement expressions performing variable assignment and method invocation are reused three times to perform the same tasks on three different columns. Programmers are likely to write such code using code template reuse.

```
TableColumn featureNameColumn = new TableColumn(table, FEATURE_NAME_COLUMN);
featureNameColumn.setText(UpdateUIMessages.TargetPage_FeatureNameColumn);
featureNameColumn.setWidth(75);

TableColumn featureVersionColumn = new TableColumn(table, FEATURE_VERSION_COLUMN);
featureVersionColumn.setText(UpdateUIMessages.TargetPage_Feature_Version);
featureVersionColumn.setWidth(75);

TableColumn featureSizeColumn = new TableColumn(table, FEATURE_SIZE_COLUMN);
featureSizeColumn.setText(UpdateUIMessages.TargetPage_Feature_Size);
featureSizeColumn.setWidth(75);
```

Figure 3.10: An example of sequence-driven code found in consecutive statement expressions.

There may be other code regions, such as consecutive *while* or *for* statements, which are also written using code template reuse, but are not covered by the above code regions. Analyzing additional language constructs may increase the number of program bugs, but the increment is not expected to be significant. For instance, we tried including consecutive *while* statements, but found no additional bugs.

3.3.3 Sequence-driven Code Search Algorithm

The previous section introduced target code regions for the search of sequence-driven code. This section describes an algorithm to locate sequence-driven code in the code regions. The main function, find-sequence-driven-code() in Figure 3.11, merges sequence driven code found in five code regions. Some code regions may be enclosed in another code region, but it is actually desirable because it allows us to find a program bug enclosed in multiple levels of sequence-driven code fragments, such as a program bug in an *if-else-if* statement enclosed in one of the consecutive method declarations.

```
find-sequence-driven-code(file) {
    tree ← parse(file)
    s1 ← find-from-switch-case-statement(tree)
    s2 ← find-from-if-else-if-statement(tree)
    s3 ← find-from-consecutive-if-statements(tree)
    s4 ← find-from-consecutive-methods(tree)
    s5 ← find-from-consecutive-statement-expressions(tree)
    return s1 ∪ s2 ∪ s3 ∪ s4 ∪ s5
}

find-from-switch-case-statement(tree) {
    switch-nodes ← collect-switch-nodes(tree)
    seq-driven-code-nodes ← ϕ // initialize a multi-list (a list of node lists).
    for-each(switch-node in switch-nodes) {
        case-nodes ← collect-case-nodes(switch-node)
        seq-driven-code-nodes ← seq-driven-code-nodes ∪ find-from-consecutive-elements(case-nodes)
    }
    return seq-driven-code-list
}
```

```
find-from-if-else-if-statement(tree) {
    if-else-if-nodes ← collect-if- else-if-nodes(tree)
    seq-driven-code-nodes ← ϕ // initialize a multi-list (a list of node lists).
    for-each(if-else-if-nodes in if-else-if-nodes) {
        if-nodes ← collect-if-nodes(if-else-if-node) // both if and else if nodes are collected as if nodes
        seq-driven-code-nodes ← seq-driven-code-nodes ∪ find-from-consecutive-elements(if-nodes)
    }
    return seq-driven-code-list
}

find-from-consecutive-if-statements(tree) {
    if-nodes ← collect-if-nodes (tree)
    return find-from-consecutive-elements(if-nodes)
}

find-from-consecutive-method-declarations(tree) {
    method-nodes ← collect-method-nodes (tree)
    return find-from-consecutive-elements(method-nodes)
}

find-from-consecutive-statement-expressions(tree) {
    expression-nodes ← collect-statement-expressions(tree)
    existing-repeat-regions ← ϕ // initialize a list of code regions. code region = <start-index, end-index>.
    for (n = M to 1) { // start from the largest n-gram of size M = 15 and get down to unigram
        // generate-n-gram() generates n-gram code regions of a given size from a given code regions.
        // for example, generate-n-gram("stmtA; stmtB; stmtA; stmtB; stmtC;", 2)
        // returns { "stmtA; stmtB;", "stmtB; stmtA;", "stmtA; stmtB;", "stmtB; stmtA;", "stmtA; stmtC;" }
        n-gram ← generate-n-gram(expression-nodes, n)
        hash-values ← get-hash-values(n-gram) // insert block boundary hash values
        // find-repeat-regions-in-ngram() has the same goal as find-repeat-regions() but makes sure that
        // repeat-regions are constructed from non-intersecting n-gram code regions.
        new-repeat-regions ← find-repeat-regions-in-ngram(hash-values)
        for-each (new-repeat-region in new-repeat-regions) {
            // add new-repeat-region if it is a non-intersecting repeat region
            // or overwrite existing-repeat-region with new-repeat-region if the new one encloses the existing one
            if (does-not-intersect-any-of-existing-repeat-regions(new-repeat-region, existing-repeat-regions)
                || encloses-one-of-the-existing-repeat-regions(new-repeat-region, existing-repeat-regions)) {
                existing-repeat-regions ← existing-repeat-regions ∪ { new-repeat-region } // add or overwrite
            }
        }
    }
    return expression-nodes[existing-repeat-regions] // returns a multi-list (a list of node lists).
}
```

Figure 3.11: The sequence-driven code search algorithm. It merges search results from five different code regions.

Supporting functions used by the sequence-driven code locator are defined in Figure 3.12. Notably, the *find-from-consecutive-elements*() function is commonly used for finding sequence driven code in a *switch-case* statement, an *if-else-if* statement, consecutive *if* statements, and consecutive method declarations. The function returns a collection of sequence driven code nodes extracted from given element nodes. The element nodes are of the same kind of language construct, but may or may not have the same code token structure. To efficiently find all sub-lists of the element nodes with the same code token structure, element nodes are transformed into a list of hash values. Then find-repeat-regions()

function is used to find sub-lists with the same, repeated hash values. The find-repeat-regions() returns a list of start and end index pairs, which can be associated with the element nodes to return a desired collection of sequence-driven code nodes.

A similar technique that utilizes hash values computed from normalized code text to find disjointedly located, structurally equivalent code fragments was applied to code clone detector systems, such as CCFinder [6] and CP-Miner [7]. A key difference of our algorithm is on the search strategy. The algorithm aims to find adjacently located, structurally equivalent code fragments, called sequence-driven code. Therefore, desired code fragments can be extracted in a single pass from a list of language construct nodes, as implemented in find-repeat-regions(), the second function in Figure 3.12. The algorithm takes linear time proportional to the number of target code regions, which is a desirable property as it helps the bug detection method scale up to handle large source code repositories.

```
find-from-consecutive-elements(element-nodes) {
    seq-driven-code-nodes ← ϕ // initialize a multi-list (a list of node lists).
    hash-values ← get-hash-values(element-nodes)
    repeat-regions ← find-repeat-regions(hash-values)
    seq-driven-code-nodes ← seq-driven-code-nodes ∪ element-nodes[repeat-regions]
}

// input: hash-values = { 1,2,2,2,3,4,4,5 }
// output: repeat-regions = { <1,3>, <5,6> }
find-repeat-regions(hash-values) {
    n ← length(hash-values)
    start-index ← 0;
    previous-value ← -999
    repeat-counter ← 0
    repeat-regions ← ϕ // collect <start-index, end-index> of repeated regions
    for (i = 0; i < n; i++) {
        current-value ← hash-values [i];
        if (previous-value!= current-value) { // new start
            if (repeat-counter > 1) {
                repeat-regions ← repeat-regions ∪ { <start-index, i – 1> }
            }
            repeat-counter ← 1
            start-index ← i
            previous-value ← current-value;
        } else {
            repeat-counter++
        }
    }

    if (repeat-counter > 1) {
        repeat-regions ← repeat-regions ∪ { <start-index, n – 1> }
    }
}

// input: { node("if (value.equals("top") { align(Top); }"), node("else if (value.equals("bottom") { align(Bottom); }")) }
```

```
// output: { hash-value("if(K.K(S){K(K);}"), hash-value("if(K.K(S){K(K);}") }
get-hash-values(nodes) {
    hash-values ← ϕ
    for-each(node in nodes) {
        // normalize text by replacing: identifier → "K"; string literal → "S"; numbers → "N"; "true" or "false" → "B";
        // redundant-white-space → ""; "else if" → "if"; line-comment → "//"; and block-comment →"/**/".
        node-signature ← get-path(node) + normalize-text(node)
        hash-values ← hash-values ∪ { hash-value(node-signature) }
    }
    return hash-values
}

// input: a node in an if block in a for loop in myMethod() in MyClass
// output: "MyClass.myMethod()[2]", where 2 indicates the number of enclosing parent code blocks.
get-path (node) {
    return class-name(node) + "." + method-name(node) + "()[" + number-of-enclosing-parent-blocks(node) + "]"
}
```

Figure 3.12: Supporting functions used by the sequence-driven code search algorithm. The find-repeat-regions() function is used to locate structurally equivalent, adjacent code fragments with time complexity linear in the number of target code regions.

3.3.4 Bug Detection Algorithm Based on Code Token Sequence Analysis

The sequence-driven code search algorithm returns a list of sequence-driven code fragments in a file. The algorithm exploits the first and second properties of program bugs described in Section 3.2.3. This section introduces an algorithm for filtering in only suspicious sequence-driven code fragments that are likely to have program bugs introduced by code editing errors. The algorithm is based on the third and fourth properties of program bug: a small number of unique code token sequences and a small number of duplicate code tokens.

3.3.4.1 Suspicious code token pattern

The algorithm is relatively long (Figure 3.14) because it takes four steps to generate bug warnings. First, it extracts code token sequences that will be analyzed for bug detection. Second, it collects all potential bug candidates whose code token sequences have a small number of duplicated code tokens (β_1). Third, it filters in suspicious bug candidates whose code token sequences have only a few mismatches (β_2). Fourth, it filters in the most suspicious bug candidates whose potentially buggy code tokens can be fixed by a valid code token located in the same program context as the original code token.

The values of threshold parameters, β_1 and β_2, are empirically determined by running the algorithm on open source projects. $\beta_1 = 1$ and $\beta_2 = 1$ are found to be most effective; many program bugs are detected with high precision using the threshold values. It is noteworthy that, with the particular thresholds, the algorithm can be viewed as looking for a particular *suspicious code token pattern* in sequence-driven

code. Given a pair of code token sequences extracted from sequence-driven code, the suspicious code token pattern is satisfied when the following two conditions are met:

- First, one code token sequence has all different code tokens. The other code token sequence has only one duplicate code token ($\beta_1 = 1$).
- Second, there is one mismatch between the two code token sequences at a duplicate code token ($\beta_2 = 1$).

The suspicious code token pattern is visualized in Figure 3.13 using examples of program bugs introduced in Figure 2.4, Figure 3.3, and Figure 3.4.

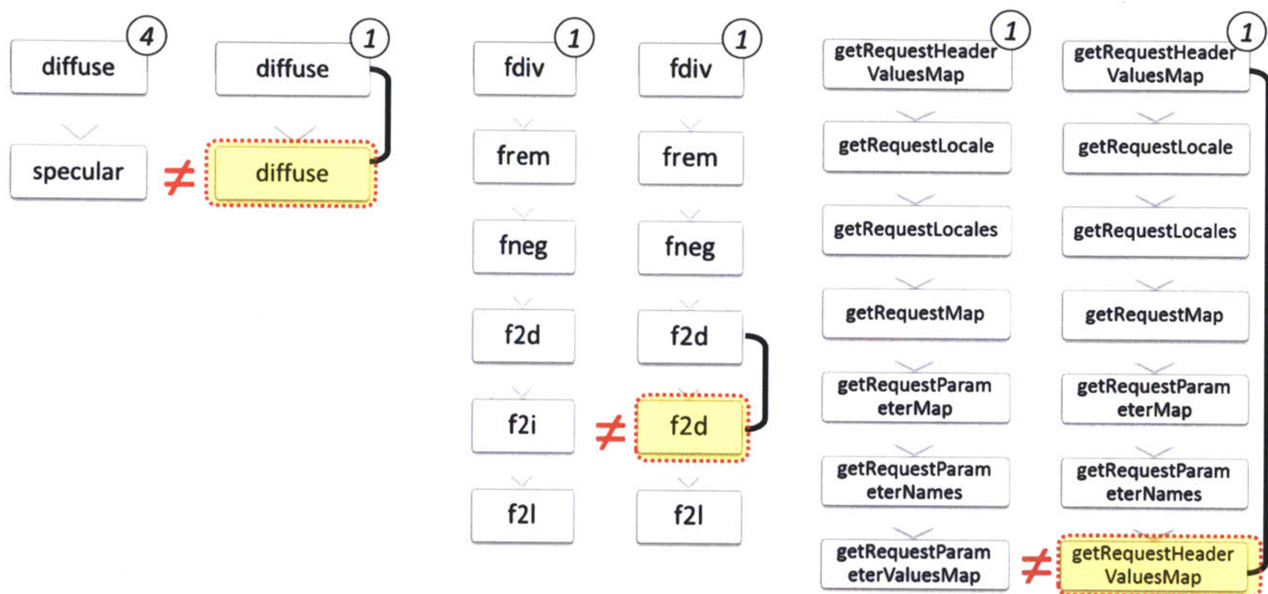


Figure 3.13: The bug detection algorithm can be viewed as looking for a certain suspicious code token pattern when it is instantiated with particular threshold values that were found effective through bug finding experiments.

3.3.4.2 Bug detection algorithm

The bug detection algorithm is described in Figure 3.14. The following four steps are taken to locate suspicious code token sequences:

First, it extracts code token sequences from each block of sequence-driven code using extract-sequences() function. If the number of unique code token sequences extracted from a block of sequence-driven code is larger than a certain threshold value, specified by max-allowed-unique-sequences(), the sequence-driven code is pruned. Also when collecting code token sequences, it groups them into two categories: one with all different code tokens (called *all-diff-sequences* in the algorithm) and the other with duplicated code

tokens (*potential-bug-sequences*). The code token sequences with duplicated tokens are called potential bug sequences because their duplicated tokens could have been introduced by a forget-to-replace or replace-with-wrong error.

Second, for each potential-bug sequence, the algorithm calculates the number of extra duplicates (*extra-dup-count*). The number of extra duplicates is calculated by subtracting the number of unique code tokens from the number of code tokens. The number of extra duplicates sets a lower bound on the number of mismatches between the potential-bug sequence and an all-different sequence; if one of the duplicated code tokens matches a code token in the all-different sequence, all other duplicated code tokens cannot match any code token in the all-different sequence. In the sense, if we can assume that the extra duplicates were introduced by code editing errors, the number of extra duplicates should be the minimum number of forget-to-replace or replace-with-wrong errors. It is less likely that a programmer introduces many code editing errors occur in a single block of sequence-driven code. So, if the number of extra duplicates is higher than a certain threshold, it would conflict with the assumption. Then the extra duplicates are likely to be intended and not errors. Therefore, if the number of extra duplicates is larger than a threshold specified by max-allowed-mismatches(), the potential-bug sequence is pruned.

Third, for potential-bug sequences with a small number of extra duplicates, the algorithm calculates the number of mismatches between a potential-bug sequence and an all-different sequence. Based on the same reasoning behind pruning potential-bug sequences with many extra duplicates, potential-bug sequences with many mismatches are pruned.

At this point, the algorithm has filtered in potential-bug sequences with a small number of extra duplicates and a small number of mismatches. In the fourth step, the algorithm tests if it is possible to come up with a name fix, which is required to satisfy program-context-based equivalence. It is straightforward to generate a name fix for a potential-bug sequence by looking at correctly matched code tokens. For instance, given potential-bug sequence (*ALIGN_TOP, ALIGN_TOP*) and all-different sequence (*top, bottom*), a name fix, *ALIGN_BOTTOM,* for a buggy code token, *ALIGN_TOP,* is generated by applying a name transformation inferred from *top* and *ALIGN_TOP*. Note that a name transformation describes which prefix and suffix need to be removed from and appended to a source code token and which case type should be used. For example, a name transformation from *top* and *ALIGN_TOP* is inferred as that *ALIGN_* is appended and the case type is changed to upper case. Applying the name transformation to *bottom* outputs a desired name fix, *ALIGN_BOTTOM.*

Once a name fix is generated by generate-name-fix(), a fixed code token and an original code token are checked for program-context-based equivalence. If the fixed code token and original code token are located in the equivalent program context, the potential-bug sequence is added to suspicious sequences (*suspicious-seqs*); the fixed code token sequence is also added to fixed sequences (*fixed-seqs*). The following tests is performed in is-equivalent-program-context() to determine whether a fixed code token and original code token are located in the equivalent program context:

- If original-code-token is a member variable of a class, fixed-code-token must be a member variable of the same class.
- If original-code-token is a method of a class, fixed-code-token must be a method of the same class.
- If original-code-token is a class in a package, fixed-code-token must be a class in the same package.
- If original-code-token is a parameter of a method, fixed-code-token must be a parameter of the same method.
- If original-code-token is a local variable in a code block, fixed-code-token must be a local variable in the same code block.

Finally, find-buggy-sequences-and-fixes() returns a list of suspicious sequences and a list of fixed sequences. A bug detection system generates a bug warning report based on the return information. The bug warning report allows users to look up the source code corresponding to a suspicious sequence and review a possible fix for the bug warning.

```
// Description: find suspicious sequences and possible fixes
// Input:     { node("if (val.equals("top") { align(Top); fireEvent(ALIGN_TOP); }"),
//              node("else if (val.equals("bottom") { align(Bottom); fireEvent(ALIGN_TOP); }")) }
// Output:    suspicious-seqs = { { "ALIGN_TOP" "ALIGN_TOP" } },
//            fixed-seqs = { { "ALIGN_TOP" "ALIGN_BOTTOM" } }
find-suspicious-sequences-and-fixes(seq-driven-code-list) {
    suspicious-seqs ← ϕ
    fixed-seqs ← ϕ
    all-diff-seqs ← ϕ
    potential-bug-seqs ← ϕ
    for-each (seq-driven-code-nodes in seq-driven-code-list) {
        { new-all-diff-seqs, new-potential-bug-seqs } ← extract-sequences(seq-driven-code-nodes)
        n ← token-count(new-all-diff-seqs [0])
        number-of-unique-seqs ← number-of-unique-sequences(new-all-diff-seqs ∪ new-potential-bug-seqs)

        // By default, max-allowed-unique-sequences() returns 1 if n = 2 and 2 if n > 2.
        // This is based on the property #3 of program bugs. We expect that the number of unique code token sequences
        // would be small in sequence-driven code. depending on the length
        if (number-of-unique-seqs ≦ max-allowed-unique-sequences(n)) {
            all-diff-seqs ← all-diff-seqs ∪ new-all-diff-seqs
            potential-bug-seqs ← potential-bug-seqs ∪ new-potential-bug-seqs
        }
    }

    for-each (potential-bug-seq in potential-bug-seqs) {
        extra-dup-count ← token-count(trimmed-pot-bug-seq) – unique-token-count(trimmed-pot-bug-seq)
        if (extra-dup-count > max-allowed-mismatches())
            continue

        // If n = 2, find-related-all-diff-seqs() returns all-diff-seqs found in the same sequence driven code
        // If n > 2, find-related-all-diff-seqs() returns all-diff-seqs found in the same file
        related-all-diff-seqs ← find-related-all-diff-seqs(potential-bug-seq, all-diff-seqs)

        // removes a prefix and suffix common in all code tokens of new-seq
        trimmed-pot-bug-seq ← trim-common-prefix-suffix(potential-bug-seq)
        is-mismatch-count-acceptable ← true
        for-each(all-diff-seq in related-all-diff-seqs) {
            trimmed-all-diff-seq ← trim-common-prefix-suffix(all-diff-seq [i])
            match-count ← 0
            for (token-index = 0; token-index < n; token-index++) {
                sub-words-in-potential-bug-seq ← sub-words(trimmed-pot-bug-seq[token-index])
                sub-words-in-all-diff-seq ← sub-words(trimmed-all-diff-seq[token-index])
                if (sub-words-in-potential-bug-seq ⊃sub-words-in-all-diff-seq)
                    match-count ++
                }
            }

            mismatch-count ← n – match-count
            // by default, max-allowed-mismatches() returns1
            if (mismatch-count > max-allowed-mismatches())
                is-mismatch-count-acceptable ← false
                continue
            }

            if (is-mismatch-count-acceptable) {
                mismatch-token-index ← mismatched-code-token-index(all-diff-seq, potential-bug-seq)
                original-code-token← potential-bug-seq[mismatch-token-index]
                fixed-code-token ← generate-name-fix(mismatch-token-index, all-diff-seq, potential-bug-seq)

                if (is-equivalent-program-context(original-code-toke, fixed-code-token)) {
```

```
                    suspicious-seqs ← suspicious-seqs ∪ { potential-bug-seq }
                    fixed-seq ← create-fixed-seq(potential-bug-seq, mismatch-token-index, fixed-code-token)
                    fixed-seqs ←fixed-seqs ∪ { fixed-seq }
                }
            }
        }
    }
    return { suspicious-seqs, fixed-seqs }
}

number-of-unique-sequences(sequences)
    unique-sequences ← ϕ
    for-each(sequence in sequences) {
        if (! is-equivalent(sequence, unique-sequences)) {
            unique-sequences ← unique-sequences ∪ { sequence }
        }
    }
    return size(unique-sequences)
}

is-equivalent(new-seq, existing-seqs) {
    if (token-count(new-seq) != token-count(existing-seqs[0]))
        return false

    n ← token-count(new-seq)

    // removes a prefix and suffix common in all code tokens of new-seq
    trimmed-new-seq ← trim-common-prefix-suffix(new-seq)

    for (i = 0; i < length(existing-seqs); i++) {
        // removes a prefix and suffix common in all code tokens of an existing-seq
        trimmed-existing-seq ← trim-common-prefix-suffix(existing-seqs [i])
        left-to-right-match-counter ← 0
        right-to-left-match-counter ← 0
        for (token-index = 0; token-index < n; token-index++) {
            sub-words-in-new-seq ← sub-words(trimmed-new-seq[token-index])
            sub-words-in-existing-seq ← sub-words(trimmed-existing-seq[token-index])
            if (sub-words-in-new-seq ⊃ sub-words-in-existing-seq)
                left-to-right-match-counter ++
            if (sub-words-in-new-seq ⊂ sub-words-in-existing-seq)
                right-to-left-match-counter ++
        }
        mismatch-counter ← n – max(left-to-right-match-counter, right-to-left-match-counter)
        // by default, max-allowed-mismatches() returns1 for any n.
        if (mismatch-counter ≦ max-allowed-mismatches(n))
            return true
    }
    return false
}


// Description: extract code token sequences from a group of sequence driven code fragments by collecting code tokens that are
//              equivalently located, but have different values across the code fragments.
// Input: {     node("if (val.equals("top") { align(Top); fireEvent(ALIGN_TOP); }"),
//              node("else if (val.equals("bottom") { align(Bottom); fireEvent(ALIGN_TOP); }")) }
// Output: { { "top" "bottom" }, { "Top" "Bottom" }, { "ALIGN_TOP" "ALIGN_TOP" } }
extract-sequences(seq-driven-code-nodes) {
    sequences ← ϕ
    // Description: get-token-row() tokenizes nodes ignoring whitespaces and non-alphanumeric characters
    // Input:   {   node("if (val.equals("top") { align(Top); fireEvent(ALIGN_TOP); }"),
```

```
    //            node("else if (val.equals("bottom") { align(Bottom); fireEvent(ALIGN_TOP); }")) }
    // Output: {  { "if" "value"  "equals" "top" "align" "Top" "fireEvent" "ALIGN_TOP" },
    //            { "if" "value"  "equals" "bottom" "align" "Bottom" "fireEvent" "ALIGN_TOP" } }
    rows ← get-token-rows(seq-driven-code-nodes)
    m ← column-count(rows)
    for (col-index = 0; col-index < m; col-index++) {
        column ← get-column(col-index, rows)
        if (all-diff (column))
            all-diff-sequences ← all-diff-sequences ∪ { column }
        else if (row-count(rows) > 2 and has-duplicates(column))
            potential-bug-sequences ← potential-bug-sequences ∪ { column }
    }

    if (row-count(rows) = 2) {
        for (col-index = 0; col-index < m; col-index++) {
            column ← get-column(col-index, rows)
            if (all-same(column) and is-potential-bug-seq(column, all-diff-sequences))
                potential-bug-sequences ← potential-bug-sequences ∪ { column }
        }
    }
    return { all-diff-sequences, potential-bug-sequences }
}

// Description: determines whether sub-words of all-same-seq row have a meaningful intersection with sub-words of any all-diff-seq row.
//              If there is one, all-same-seq could have been introduced by a text editing error.
// Input: all-same-seq = { "ALIGN_TOP" "ALIGN_TOP" }, all-diff-seqs = { { "top" "bottom" }, { "Top" "Bottom" } }
// Output: true
// Input: all-same-seq = { "val" "val" }, all-diff-seqs = { { "top" "bottom" }, { "Top" "Bottom" } }
// Output: false
is-potential-bug-seq(all-same-seq, all-diff-seqs) {
    all-same-token ← all-same-seq [0]
    // sub-word() returns sub-words in a compound word token.
    // input: ALIGN_TOP, output = { "ALIGN" "TOP" }
    sub-words-in-all-same-token ← sub-words(all-same-token)
    n ← row-count(not-all-same-seqs)
    for (i = 0; i < length(all-diff-seqs); i++) {
        // removes a prefix and suffix common in all code tokens of an all-diff-seq
        trimmed-all-diff-seq ← trim-common-prefix-suffix(all-diff-seqs [i])
        for (row-index = 0; row-index < n; row-index++) {
            if (sub-words-in-all-same-token ⊃ sub-words(trimmed-all-diff-seq [row-index]))
                return true
        }
    }
    return false
}
```

Figure 3.14: Bug detection algorithm based on code token sequence analysis.

3.4 Implementation of the Bug Detection Method

A bug detection system, called *Bugsy Finder*, implementing the sequence-driven code search and code-token-sequence-based bug detection algorithms was developed through iterative development process.

3.4.1 Implementation

The bug detection system is implemented in the Java language. It takes the location of root directory of

Java source code and generates a bug warning report in a spreadsheet file format (Figure 3.1). Optionally, the location of external libraries referred in source code can be specified. When the algorithm checks the program-context-based equivalence of a fixed code token and an original code token, it first tries to resolve data types of code tokens based on a parse tree generated from the source code. If it fails to find a necessary class definition from the parse tree, it then queries external libraries to resolve the data types. Note that Java Standard Edition libraries[2] are automatically included in the external libraries. The system builds a parse tree of Java source code using an open source parser generator called JavaCC[3]; this part of implementation can be extended to support other programming languages.

3.4.2 Iterative Development Process

Development of the bug detection system was particularly challenging because it had to be designed to find hypothetical program bugs. No known program bugs were available for algorithm design or for performance evaluation. Bug tracking systems and revision control systems, which are in wide use and contain a large amount of information related to program bugs, were considered as a source for building a database of relevant program bugs. However, the approach seemed too expensive to execute for several reasons. Bug tracking systems, such as Bugzilla[4], do not provide a fine-grained categorization or a computer-friendly structured description of program bugs, which could be useful for filtering in program bugs related to text editing errors. Revision control systems, such as Subversion[5], can query source code modifications related to bug fixes, but it is still difficult to filter in relevant code modifications because actual descriptions of bug fixes and program bugs are usually stored in unstructured text, scattered in various locations, such as commit messages, source code comments, and records in bug tracking systems.

Given no existing database of program bugs related to text editing errors, an iterative approach was taken to develop both a bug detection algorithm and a database of program bugs together, with advances on one side helping to advance on the other side at each step of iteration. We first started by developing an algorithm that detected several particular sequential patterns of code tokens that might occur in similar code fragments with code editing errors. The algorithm was then run on source code of Java open source projects (introduced in Section 3.5.1) and generated bug warnings, some of which were found true program bugs through manual verification. This way, the first, relatively small set of program bugs related to text editing errors was found. A next version of algorithm is developed based on properties learned

[2] http://download.oracle.com/javase/
[3] http://javacc.java.net/
[4] http://www.bugzilla.org
[5] http://subversion.tigris.org

from previously found program bugs. The next version of algorithm was then run on the same open source projects to generate an updated set of bug warnings. Again, the bug warnings were verified manually to find new true program bugs, which were used, together with all other previously found bugs, for the next iteration of algorithm development.

3.5 Evaluation of the Bug Detection Method

The new bug detection method was evaluated on the basis of the number of program bugs and precision through bug finding experiments on seven well-maintained open source projects.

3.5.1 Selection of Open Source Projects

Well-known and presumably well-maintained 7 open source projects are selected for evaluation of the bug detection method. The seven open source projects are:

- Eclipse 3.2 (01/08/2007): a Java development tool. 2.2M LOC (million lines of code; excluding blank lines and comments.).
- NetBeans 6.9.1: a Java development tool. 3.6M LOC.
- JRuby 1.1.5: a Java implementation of the Ruby programming language. 98K LOC.
- Jython 2.5.1: a Java implementation of the Python programming language. 98K LOC.
- JBoss Seam 2.2.1: a Web application framework written in Java. 74K LOC.
- Tomcat 6.0.29: a Servlet and Java Server Pages web server. 171K LOC.
- jMonkeyEngine 2.0.1 (jME): a 3D game engine written in Java. 153K LOC.

3.5.2 Bug Finding Experiment Procedure

The following procedure is repeated for each open source project:

1. Specify the locations of source code and library files.
2. Run the bug detection system. A list of bug warnings is reported in a spreadsheet file.
3. Examine source code of each bug warning to verify whether it is a true bug or false warning. Count the number of true bugs and false warnings.

An example of bug warning report is shown in Figure 3.15. Using the file path and code line information in the spreadsheet, a human verifier can review source code associated with a bug warning and determine whether it is a true bug or false warning. Verification of bug warnings was performed by the thesis author. The list of program bugs found from the experiment is publicly available on the Web: http://www.sangmok.org/bugsy-finder.

A large pool of about 200 JAR files, most of which related to Java EE standards[6], were specified as library files on all open source projects. Note that it is okay to leave unreferred JAR files in the library because they will not be loaded during the bug detection process.

left sequence	*right sequence*	*file path & code line*			*occur.*	
[OSGI_WS OSGI_ARCH OSGI_NL]	[ws arch arch]	C:\eclipse-	534	534	1	1
[highButton normalButton lowButton]	[PRIORITY_HIGH PRIORITY_NORMAL PRIORITY_NORMAL]	C:\eclipse-	246	443	1	1
[Dispose Move Resize]	[disposeColumnListener resizeListener resizeListener]	C:\eclipse-	301	301	1	1
[clientX clientY]	[clientX clientX]	C:\eclipse-	111	111	1	1
[addListener removeListener]	[add add]	C:\eclipse-	50	50	1	1
[getSignerInfo getVerifierInfo]	[signerInfo signerInfo]	C:\eclipse-	193	193	2	1

Figure 3.15: An example of bug warnings reported in a spreadsheet file. Using the file path and code line information, a human verifier can review source code associated with a bug warning and determine whether it is a true bug or false warning.

3.6 Results and Discussion of the Bug Detection Method

Bug finding experiments found many previous unknown program bugs in the 7 open source projects. The results provide an empirical support for our program bug hypothesis. This section discusses the impact of algorithm parameters on accuracy in terms of precision and recall. Finally, a comparison with a previous method for finding copy-paste-related program bugs is discussed.

3.6.1 Empirical Support for the Hypothesis

A total of 108 program bugs related to code editing errors during code template reuse were discovered from 7 open source projects, as shown in Table 3.1. The number is a count of all known program bugs that were discovered through iterative development process of the bug detection method (Section 3.4.2). The number is considered high enough to support our hypothesis that code editing errors during code template reuse may be a significant source of program bug. Although such program bugs are introduced by tedious code editing errors, they pose a risk to software quality.

Table 3.1: The number of program bugs found in the 7 open source projects. A total of 108 program bugs were found through the iterative development process of the bug detection method.

Project Name	*Eclipse*	*NetBeans*	*JBoss Seam*	*Tomcat*	*JRuby*	*Jython*	*jME*	*Sum*
Number of all known program bugs	35	56	6	1	3	0	7	**108**

[6] http://download.oracle.com/javaee/

3.6.2 Evaluation of Accuracy

Accuracy of the bug detection method was evaluated on the basis of precision and recall, which are defined as follows in our context:

$$\text{Precision} = \frac{\text{number of true bugs}}{\text{number of bug warnings}}$$

$$\text{Recall} = \frac{\text{number of known bugs reported as bug warnings}}{\text{number of all known bugs}}$$

Note that the recall is evaluated based on all known bugs, which is a subset of all program bugs related to code template reuse. Therefore, the recall values reported in this section must be higher than the true value of recall based on the all program bugs. For this reason, the recall should not be interpreted as an absolute measure of the method's recall performance; however, the recall value can be useful for comparing the effects of different parameter values used with the bug detection algorithm.

The bug detection algorithm described in Section 3.3.4 involves three parameters that need to be determined empirically. The first two—the maximum number of extra duplicates (β_1) and the maximum number of token mismatches (β_2)—are essentially a single parameter because they reflect the number of code editing errors that are expected to occur in a block of sequence-driven code. For instance, if a programmer makes one forget-to-replace error while writing a block of sequence-driven code, the sequence-based analysis will find a sequence with two code tokens with the same text (one extra duplicate) and one mismatch between the duplicate-token sequence and all-different code token sequence. For these two parameters, two possible values—1 and 2—were examined: $\beta_1 = \beta_2 = 1$ and $\beta_1 = \beta_2 = 2$.

The third parameter is the maximum number of unique code token sequences in a block of sequence-driven code (β_3). It reflects how good the alignment among code token sequences is. Generally, duplicated and mismatched code tokens found in well-aligned sequence-driven code are a positive sign of code editing error (likely to be a code editing error). Because the mechanism for determining equivalence of code token sequences (is-equivalent function in Figure 3.14) is implemented such that it consider a pair of code token sequences as equivalent if the number of mismatched tokens is equal to or smaller than the maximum number of token mismatches (β_1), the number of unique code token sequences can be still one when some sequences have duplicate tokens and mismatches.

Setting the maximum number of unique code token sequences (β_3) to 1 will strictly filter in well-aligned sequence-driven code and thus may help increase precision of bug detection. However, it may not be an ideal threshold because there some sequence-driven code with program bugs may have two or more unique code token sequences for several reasons. In other words, using a too strict threshold value on β_3 may cause missing some of the existing program bugs and hurt the recall performance. Some sequence-driven code may have two unique code token sequences: one for variable names and the other for constant values. For example, Figure 3.16 shows sequence-driven code with a program bug found in the NetBeans open source project. The sequence-driven code has two unique code token sequences as there are two groups of equivalent code token sequences: { (*ERR_RCS ERR_VERS*) (*ERR_RCS ERR_RCS*) } and { (*repositoryName repositoryRevision*) (set*RepositoryFileName* set*RepositoryRevision*) }. Two unique code token sequences may also occur when a block of sequence-driven code has one code token sequence for variable names and the other for data types associated with the variables. To evaluate the impact of β_3 on precision and recall, two possible strategies of setting the threshold was evaluated: the first strategy is to use 1 for shorter code token sequences of length 2 and use to 2 for all other longer code token sequences; the second strategy is to use 2 for all code token sequences of any length.

```
else if (line.startsWith(ERR_RCS)) {
   if (fileInformation != null) {
      String repositoryName =
            line.substring(ERR_RCS.length()).trim();
      fileInformation.setRepositoryFileName(repositoryName);
   }
}
else if (line.startsWith(ERR_VERS)) {
   if (fileInformation != null) {
      String repositoryRevision =
            line.substring(ERR_RCS.length()).trim();
      fileInformation.setRepositoryRevision(repositoryRevision);
   }
}
```

Figure 3.16: Sequence-driven code with a program bug found in NetBeans. It involves two unique code token sequences: one for variable names and the other for constants names.

The following 4 configurations of the bug detection method were evaluated:

- $\beta_1 = \beta_2 = 1$, $\beta_3 =$ 1-or-2 (1 for length = 2; 2 for length > 2)
- $\beta_1 = \beta_2 = 1$, $\beta_3 = 2$
- $\beta_1 = \beta_2 = 2$, $\beta_3 =$ 1-or-2 (1 for length = 2; 2 for length > 2)
- $\beta_1 = \beta_2 = 2$, $\beta_3 = 2$

Table 3.2 summarizes the precision and recall of the bug detection method. It is interesting to see that configurations with $\beta_1 = \beta_2 = 1$ gave a better performance over corresponding configurations with $\beta_1 = \beta_2 = 2$, regardless the value of β_3. In other words, changing $\beta_1 = \beta_2 = 1$ to $\beta_1 = \beta_2 = 2$ has no impact on recall, but only decreases precision either when using β_3 = 1-or-2 or $\beta_3 = 2$.

Not surprisingly, the maximum number of unique code token sequences, β_3, had an impact on both precision and recall. When using a more strict β_3 = 1-or-2, accuracy was higher than when using $\beta_3 = 2$ (46.9% vs. 39.4%). However, using the more strict β_3 value resulted in detecting 11 less program bugs than using $\beta_3 = 2$.

We notice that precision and recall values varied considerably across the open source projects. However, it is noteworthy that precision values of two large open source projects, Eclipse and NetBeans, were very similar, about 40%. Given that the precision values were based on 135 sample points (bug warnings), we think that the precision values should closely reflect the true performance of the bug detection method.

Table 3.2: Precision and recall accuracy of the bug detection method. The recall values are calculated from the number of true bugs and the number of all known program bugs in Table 3.1.

$\beta_1 = \beta_2 = 1$ (one extra duplicate and one mismatch) and β_3 = 1-or-2 (one or two unique code token sequences)								
Project Name	*Eclipse*	*NetBeans*	*JBoss Seam*	*Tomcat*	*JRuby*	*Jython*	*jME*	***Average***
True bugs	25	37	4	1	3	0	6	**76**
False warnings	24	49	0	5	1	1	6	**86**
Precision (%)	51.0%	43.0%	100.0%	16.7%	75.0%	0.0%	50.0%	**46.9%**
Recall (%)	71.4%	66.1%	66.7%	100.0%	100.0%	N/A	85.7%	**70.4%**

$\beta_1 = \beta_2 = 1$ (one extra duplicate and one mismatch) and $\beta_3 = 2$ (maximum two unique code token sequences)								
Project Name	*Eclipse*	*NetBeans*	*JBoss Seam*	*Tomcat*	*JRuby*	*Jython*	*jME*	***Average***
True bugs	30	43	4	1	3	0	6	**87**
False warnings	46	65	0	5	3	3	12	**134**
Precision (%)	39.5%	39.8%	100.0%	16.7%	50.0%	0.0%	33.3%	**39.4%**
Recall (%)	85.7%	76.8%	66.7%	100.0%	100.0%	N/A	85.7%	**80.6%**

$\beta_1 = \beta_2 = 2$ *(max. two extra duplicates and max. two mismatches) and* β_3 *= 1-or-2 (one or two unique code token sequences)*								
Project Name	*Eclipse*	*NetBeans*	*JBoss Seam*	*Tomcat*	*JRuby*	*Jython*	*jME*	***Average***
True bugs	25	37	4	1	3	0	6	**76**
False warnings	34	60	0	5	1	1	7	**108**
Precision (%)	42.4%	38.1%	100.0%	16.7%	75.0%	0.0%	46.2%	**41.3%**
Recall (%)	71.4%	66.1%	66.7%	100.0%	100.0%	N/A	85.7%	**70.4%**

$\beta_1 = \beta_2 = 2$ *(max. two extra duplicates and max. two mismatches) and* β_3 *= 2 (maximum two unique code token sequences)*								
Project Name	*Eclipse*	*NetBeans*	*JBoss Seam*	*Tomcat*	*JRuby*	*Jython*	*jME*	***Average***
True bugs	30	43	4	1	3	0	6	**87**
False warnings	56	76	0	5	3	3	13	**156**
Precision (%)	34.9%	36.1%	100.0%	16.7%	50.0%	0.0%	31.6%	**35.8%**
Recall (%)	85.7%	76.8%	66.7%	100.0%	100.0%	N/A	85.7%	**80.6%**

3.6.3 Comparison with a Previous Method for Copy-paste-related Bug Detection

A relatively old version of Eclipse source code (Version 3.2 on 01/08/2007) was selected as one of the open source projects for evaluation because the version of Eclipse source code was used in previous work on bug detection of copy-pasted-related program bugs by Jiang et al. [9]. The previous work reported the number of program bugs introduced by inconsistent identifier replacement errors and the accuracy of detecting such program bugs. This section discusses differences between the previous method and new bug detection method, Bugsy Finder, in terms of accuracy of bug detection, the types of program bugs, and the number of program bugs.

Table 3.3 summarizes the differences of the two bug detection methods. The most noticeable difference is in the precision of bug detection: precision of the new method is about 9 times higher than that of the previous method (51% vs. 5.7%). In other words, the previous method requires verifying about 18 bug warnings to find one true bug; the new method requires verifying only two bug warnings to find one true bug. High precision is a critical requirement for a wide adoption of a bug detection method because bugs verification is difficult and time-consuming. Although there is still much room for improvement, Bugsy Finder makes a significant improvement over the previous method in terms of precision.

The recall values were equivalent in the two bug detection methods (71.4%). The recall value is

calculated as a ratio of the number of true bugs detected by a bug detection method to the number of all program bugs found by researchers who developed the bug detection method. The recall should not be interpreted as an absolute measure of each method's recall performance, but may be useful to compare estimated recall performance of the two methods. Also note that both bug detection methods used different techniques to filter out false positive warnings. Bugsy Finder used parameter configuration of $\beta_1 = \beta_2 = 1$ and β_3 = 1-or-2; the previous method used six filtering heuristics described in [9].

Table 3.3: Program bugs found by two different bug detection methods in the source code of Eclipse project.

Bug Type	*New Method (Bugsy Finder)*	*Previous Method [9]*
missed conditional check	-	8
inappropriate conditions	-	3
off-by-one	-	1
inappropriate logic for corner cases	-	2
unhandled cases or exceptions	-	3
wrong local variable	11	4
wrong method	7	-
wrong member variable	9	-
wrong member constant	7	-
Wrong class	1	-
total number of program bugs	35	21
precision (%) with filter	**51.0% (25/49)**	**5.7% (15/265)**
recall (%) with filter	**71.4% (25/35)**	**71.4% (15/21)**

Another noteworthy difference is in the types of program bugs. Program bugs found by the new bug detection method are classified based on the bug types defined in the previous work. Each bug detection method was found effective at finding different bug types. Only one bug type, the wrong local variable error, was commonly found by the two methods. This is a positive aspect of the new bug detection method because it enables programmers to find program bugs that were difficult to find using the previous method.

Lastly, the new bug detection method found 66.7% (14/21) more program bugs than the previous method. This is noteworthy because it supports that the hypothesized source of program bugs—code editing errors during code template reuse—is as significant a risk as previously known sources of program bugs—the inconsistent evolution of copy-pasted code fragments.

3.7 User Interface for Error-preventive Code Editing

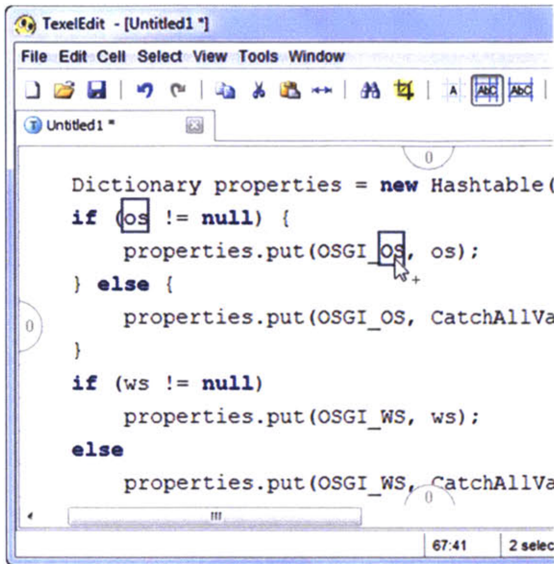
A bug detection method is useful at finding program bugs that have already been introduced, but does not help with preventing new program bugs. This section presents an error-preventive code editing approach

that aims to proactively reduce program bugs during code template reuse. The new code editing approach, called *Texel Editing*, reduces the number of code editing steps by supporting efficient multiple-selection based on cell-based text editing and by automating code template reuse through a new code generation user interface for sequence-driven code.

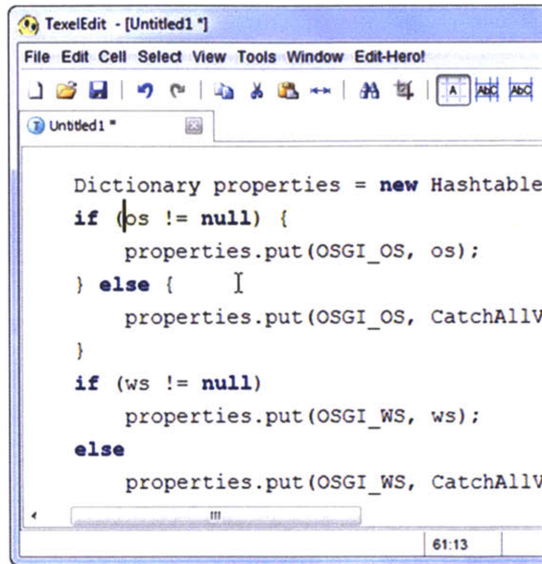
3.7.1 Efficient Multiple-Selection User Interface

Most selection targets during code template reuse are words or sub-words (words in a compound word), which often have the same text. Unfortunately, in normal text editing, keyboard and mouse operations for performing such selection tasks are far from optimal. A precise mouse drag is necessary for selecting a sub-word; a mouse double-click is necessary for selecting a word; complex Arrow, Ctrl, and Shift key combinations are necessary for selecting a word or sub-word using a keyboard.

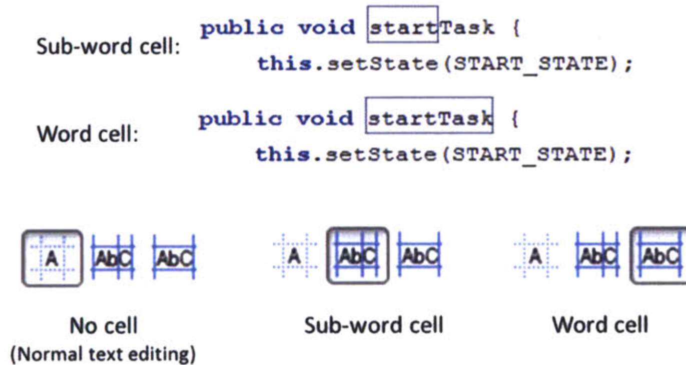
Cell-based text editing is proposed to improve efficiency of code template use by optimizing keyboard and mouse interactions for common selection tasks during code template reuse. The key idea is to transform source code into a spreadsheet-like representation in which individual word (or sub-word) tokens are contained in separate cells. Then users can select or modify multiple word (or sub-word) tokens through efficient yet familiar mouse and keyboard operations supported in a spreadsheet. Users have control over the size of cells, so they can instantly switch between different cell sizes (word or sub-word) or between different modes of editing (normal or cell-based text editing). *TextEdit* is an experimental text editor supporting both cell-based text editing and normal text editing (Figure 3.17). A cell size selector in the toolbar is used to switch between different cell sizes as well as editing modes.



(a) TexelEdit in the cell-based text editing mode



(b) TexelEdit in the normal text editing mode



(c) Cell size selector

Figure 3.17: Screenshots of TexelEdit, an experimental text editor supporting cell-based text editing. Users have an option to show or hide cell borders. Screenshots in (a) and (c) were taken with cell borders visible to make it easy for readers to see how source code is split into cells.

3.7.1.1 Mouse click for Selecting a Word or Sub-word

The most basic and efficient way to select a word or sub-word in cell-based text editing is using a mouse click. Users also use Ctrl + mouse-click to add word or sub-word selections, like they select multiple files in a file browser, as shown in Figure 3.18.

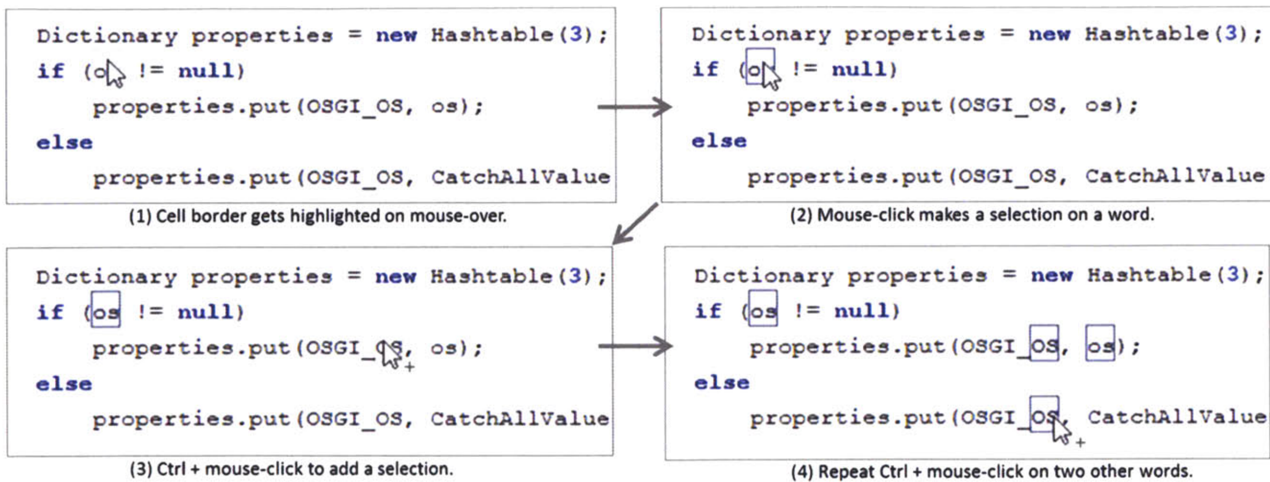


Figure 3.18: Mouse-click selection in cell-based text editing: a mouse click on a word or sub-word makes the cell of the word or sub-word selected.

3.7.1.2 Mouse Double-click for Selecting Words or Sub-words with the Same Text

To exploit the fact that code tokens with the same text are commonly selected during code template, cell-based text editing performs text-search-based selection when a word or sub-word cell is double-clicked. For example, the four selections in Figure 3.18 can also be made by a double click on any of the four code tokens. The user interface also incorporates visual aids to inform users of off-screen matches as well as a selection crop mechanism to filter in only desired cells (Figure 3.19).

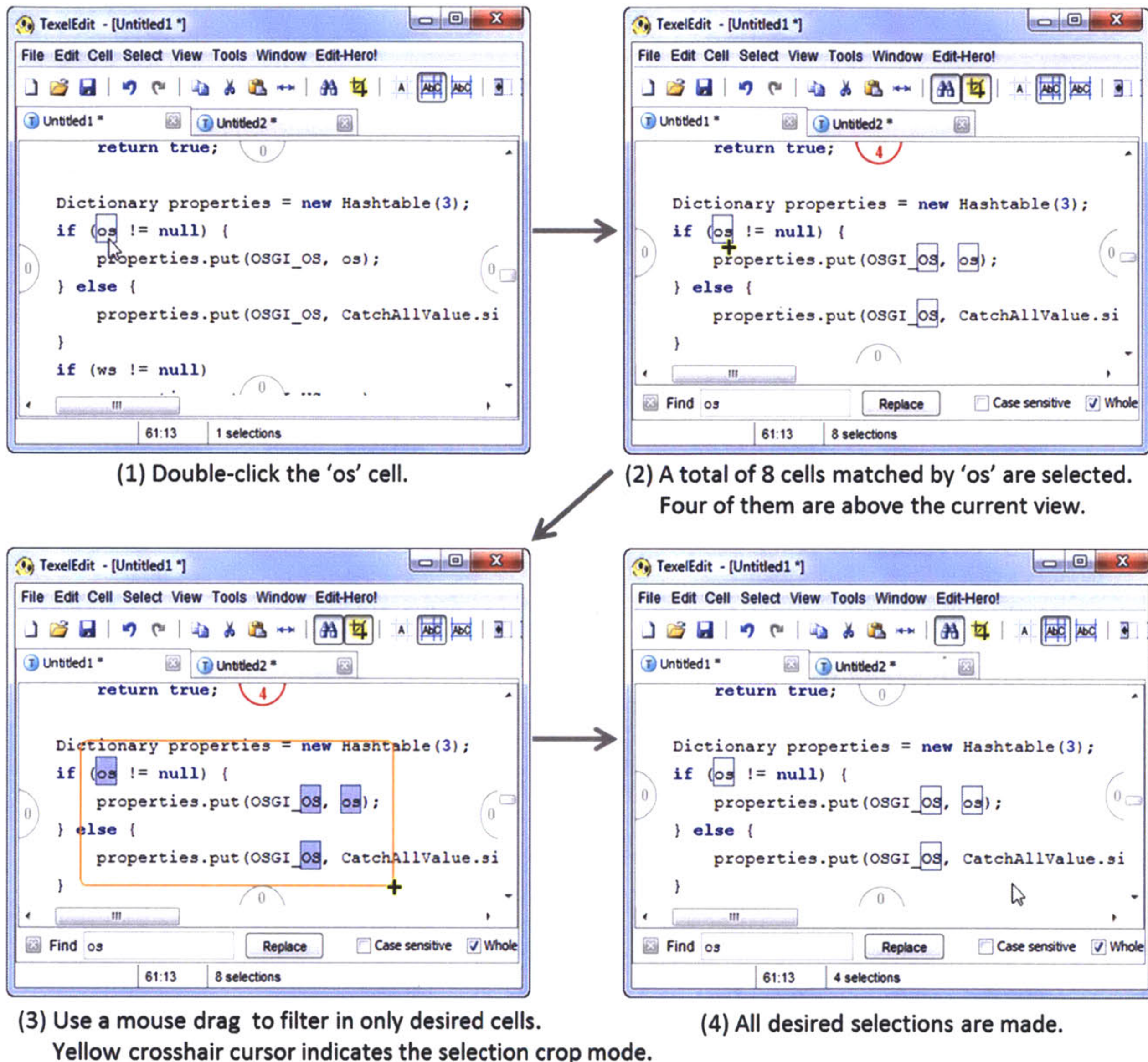
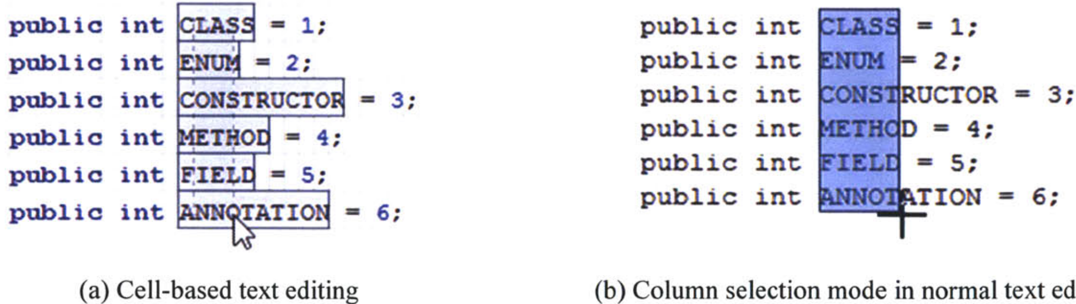


Figure 3.19: Mouse-double-click selection in cell-based text editing: A mouse double click on a word or sub-word performs text-search-based selection. To make the selection approach generally applicable to scenarios with off-screen matches, the user interface also incorporates an off-screen match indicator and a selection crop mechanism.

3.7.1.3 Mouse Drag for Selecting Semi-aligned Multiple Words or Sub-words

In cell-based text editing, a mouse drag is used to select multiple cells intersecting a rectangular marquee. Because selections are made by cell, the mouse-drag selection is useful for selecting semi-aligned code tokens, such as the ones shown in Figure 3.20. Selecting semi-align code tokens is necessary when users want to copy multiple code tokens that will be used for generating sequence-driven code—an exemplary usage scenario is described in Figure 3.28. It is also used with a multiple-selection replace command

when users want to generate sequence-driven code that involves more than one unique code token sequences, as described in Section 3.8.4

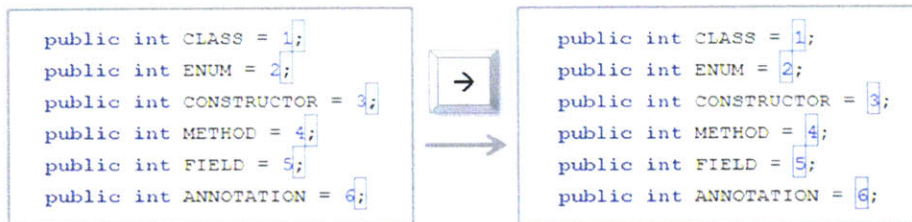


(a) Cell-based text editing (b) Column selection mode in normal text editing

Figure 3.20: Mouse-drag selection in cell-based text editing: a mouse drag performs marquee-based cell selection. (a) Because selections are made by cell, not by character, the mouse-drag selection is useful for selecting semi-aligned code tokens. (b) The column selection mode of normal text editing also supports marquee-based cell selection; however, it is applicable only when code tokens are completely aligned.

3.7.1.4 Arrow Keys for Moving Selections by Word or Sub-word

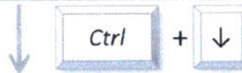
Cell-based text editing also extends keyboard-based selection manipulation to multiple selections of text cells. In particular, a Left/Right Arrow key press moves all selections by one word (or sub-word) to the left or right. When all existing selections need to expand or shrink by word (or sub-word) on the right edge, the Shift + Arrow key is used. The keyboard-based selection manipulation is useful for selecting code tokens that do not have the same text, but are located at equivalent positions relative to certain reference points. For example, all number tokens just one-word before the semi-colons can be selected by selecting all semi-colons and then moving all selections to the left using the Arrow key (Figure 3.21). It is also possible to transfer all selections in one code fragment to another structurally equivalent code fragment just above or below it by pressing Ctrl + Up/Down Arrow key (Figure 3.21). The selection transfer followed by a multiple-selection replace command, introduced Section 3.8.4, is useful for making consistent modifications on each similar code fragment.



(1) Double-click the ';' to select all its occurrences. (2) Press Left Arrow key to move selections to the left

```
public JLabelOperator lblCategories() {
    if (_lblCategories==null) {
        _lblCategories = new JLabelOperator(Bundle.getString("CTL_Categories"));
    }
    return _lblCategories;
}
public JLabelOperator lblProject() {
    if (_lblProject==null) {
        _lblProject = new JLabelOperator(Bundle.getString("CTL_Project"));
    }
    return _lblProject;
}
```

(1) Press Ctrl + Down Arrow to transfer selections to a lower code fragment.




```
public JLabelOperator lblCategories() {
    if (_lblCategories==null) {
        _lblCategories = new JLabelOperator(Bundle.getString("CTL_Categories"));
    }
    return _lblCategories;
}
public JLabelOperator lblProject() {
    if (_lblProject==null) {
        _lblProject = new JLabelOperator(Bundle.getString("CTL_Project"));
    }
    return _lblProject;
}
```

(2) Now users can replace selected tokens with a new token relevant to the second code fragment.

Figure 3.21: Keyboard-based selection in cell-based text editing: keyboard-based selection manipulation is extended to support multiple selections in cell-based text editing.

3.7.2 Efficient Code Generation User Interface for Code Template Reuse

The bug finding experiment (Section 3.4) found that many instances of sequence-driven code have only one unique code token sequence. This section presents a code generation user interface that can further reduce the number of editing steps by automating code template reuse required for writing such sequence-driven code.

In essence, writing sequence-driven code with only one unique code token sequence requires three pieces of information. First, it requires a code fragment, called an original code fragment, which is be used as a structural template for other similar code fragments. Second, it requires information about which code tokens in the original code fragment need to be different in each of the similar code fragments, which are called pivot tokens. Third, it requires information about new code tokens that should replace pivot tokens in each duplicated code fragment. Figure 3.22 presents a code generation user interface that can collect the three pieces of information from multiple selections and a line of text input. The code region that will be used as an original code fragment is inferred from multiple selections by looking up the smallest code block enclosing all multiple selections, visualized as a gray box in the figure. Pivot tokens are specified by multiple selections themselves. Finally, a list of new code tokens that will replace pivot tokens is specified in a text input box in a comma separated format.

```
Dictionary properties = new Hashtable(3);
    os,ws,arch,nl
if (os != null)
    properties.put(OSGI_OS, os);
else
    properties.put(OSGI_OS, CatchAllValue.singleton);
```

Figure 3.22: A user interface for collecting required information for automatically generating sequence-driven code.

Once the user presses the Enter key to execute the code generation, the user interface passes collected information to a code generation engine, which performs the following operations:

- First, it generates a code template from a given original code fragment. The code template replaces each pivot token with a placeholder token, which keeps case-type information of the pivot token (Figure 3.23).
- Second, it retrieves a list of new code tokens from the comma separated text and then duplicates the code template as many times as the number of new code tokens.
- Third, for each duplicated code template, replace all placeholder tokens with a corresponding new code token. The new code token must be transformed to case types marked in the placeholder tokens.
- Finally, duplicated code templates are merged into a single code block and returned as sequence-driven code, such as the one in Figure 3.24.

```
if ($WORD:9$ != null)
      properties.put(OSGI_$WORD:3$, $WORD:9$);
else
      properties.put(OSGI_$WORD:3$, CatchAllValue.singleton);
```

Figure 3.23: An example of code template generated from the user input in Figure 3.22. Note that $WORD:9$ is a placeholder for a word token marked as the lower-camel-case type; $WORD:3$ is a placeholder for a word token marked as the all-upper-case type.

```
if (os != null)
    properties.put(OSGI_OS, os);
else
    properties.put(OSGI_OS, CatchAllValue.singleton);

if (ws != null)
    properties.put(OSGI_WS, ws);
else
    properties.put(OSGI_WS, CatchAllValue.singleton);

if (arch != null)
    properties.put(OSGI_ARCH, arch);
else
    properties.put(OSGI_ARCH, CatchAllValue.singleton);

if (nl != null)
    properties.put(OSGI_NL, nl);
else
    properties.put(OSGI_NL, CatchAllValue.singleton);
```

Figure 3.24: Sequence-driven code generated from the user input.

3.7.3 Predictive Evaluation

This section presents a predictive evaluation of the proposed user interfaces, which will be referred to as cell-based text editing. Estimated usage time of two code template reuse tasks were decreased by 45% and 76% when using the proposed approach compared to the normal text editing.

3.7.3.1 Task Description

Efficiency of the proposed user interfaces for multiple-selection and code generation of sequence-driven code is evaluated based on the Keystroke Level Model [31], which provides an estimated time for performing a series of keyboard, mouse, and mental operations. Two code fragments were randomly selected for evaluation from 91 sequence-driven code fragments in Eclipse and NetBeans found by the bug detection method. The goal of each code editing task is to produce bug-free sequence-driven code

using either normal text editing or cell-based text editing.

- *Task #1*: Write the second code fragment that is the same as the first code fragment except that all occurrences of *Input* must be replaced with *Output*.
    - The snippet was adapted from BasicMethoid.java in NetBeans.

```
public int getIndexOfInputField(IField field) {
    return mInput.indexOf(field);
}

public int getIndexOfOutputField(IField field) {
    return mOutput.indexOf(field);
}
```

- *Task #2*: Write the second, third, and fourth code fragments that are the same as the first code fragment except that all occurrences of *os* must be replaced with *ws*, *arch*, and *nl*, respectively.
    - The snippet was adapted from ClasspathComputer3_0.java in Eclipse.

```
if (os != null)
    properties.put(OSGI_OS, os);
else
    properties.put(OSGI_OS, CatchAllValue.singleton);

if (ws != null)
    properties.put(OSGI_WS, ws);
else
    properties.put(OSGI_WS, CatchAllValue.singleton);

if (arch != null)
    properties.put(OSGI_ARCH, arch);
else
    properties.put(OSGI_ARCH, CatchAllValue.singleton);

if (nl != null)
    properties.put(OSGI_NL, nl);
else
    properties.put(OSGI_NL, CatchAllValue.singleton);
```

3.7.3.2 Predictive Evaluation

Keyboard, mouse, and mental operations for completing the code template reuse tasks are generated for each text editing method. Then estimated time usage is calculated based on the following coefficients [31]: Keystroke (K) = 0.28 sec, Mental (M) = 1.35 sec, Mouse/Keyboard Homing (H) = 0.36 sec, Mouse Button (B) = 0.10, Mouse Pointing (P) = 1.10 sec, and Typing T(n) = n * K sec.

- Normal text editing
    - Method: Use a double click to select a word token. Select a sub-word token using a mouse drag. Use Ctrl-C/V shortcuts for copy-paste.
    - *Task #1*

Plan	Select the first code block using mouse drag. Copy and paste the code block. Select 'Input' using mouse drag. Type 'Output'. Select the next 'Input' using mouse drag. Type 'Output'.
Operation	M(find) P(to-block-start) M(verify) B(press) M(find) P(to-block-end) M(verify) B(release) M(verify) K(Ctrl) K(C) M(verify) M(find) P(to-insert-start) M(verify) BB(click) M(verify) K(Ctrl) K(V) M(verify) M(find) P(to-input-start) M(verify) B(press) M(find) P(to-input-end) M(verify) B(release) M(verify) M(prepare) H(to-kbd) T(7:Output) M(verify) M(find) H(to-mouse) P(to-input-start) M(verify) B(press) M(find) P(to-input-end) M(verify) B(release) M(verify) M(prepare) H(to-kbd) T(7:Output) M(verify)
Est. Time	K x 18 + M x 20 + P x 7 + B x 8 + H x 3 = **41.62 sec**

- *Task #2*

Plan	Select the first code block using mouse drag. Copy and paste the code block three times. { Select 'os' using mouse double click. Type 'ws'. Copy 'ws'. Select the next 'os' using mouse double click. Paste 'ws'. Select 'OS' using mouse double click. Type 'WS'. Copy 'WS'. Select the next 'OS' using mouse double click. Paste 'WS'. } Repeat the steps within the curly braces for 'arch' and 'nl'.
Operation	M(find) P(to-block-start) M(verify) B(press) M(find) P(to-block-end) M(verify) B(release) M(verify) K(Ctrl) K(C) M(verify) M(find) P(to-insert-start) BB(click) M(verify) K(Ctrl) K(V) M(verify) K(V) M(verify) K(V) M(verify) { M(find) P(to-os) M(verify) BBBB(double-click) M(verify) H(to-kbd) T(2:ws) M(verify) M(find) H(to-mouse) P(to-ws) M(verify) BBBB(double-click) M(verify) K(Ctrl) K(C) M(find) P(to-os) M(verify) BBBB(double-click) M(verify) K(Ctrl) K(V) M(find) P(to-OS) M(verify) BBBB(double-click) M(verify) H(to-kbd) T(2:WS) M(verify) M(find) H(to-mouse) P(to-WS) M(verify) BBBB(double-click) M(verify) K(Ctrl) K(C) M(find) P(to-OS) M(verify) BBBB(double-click) M(verify) K(Ctrl) K(V) } M(verify) } Repeat the steps within the curly braces for 'arch' and 'nl'.
Est. Time	K x 39 + M x 74 + P x 21 + B x 76 + H x 12 = **144.44 sec**

- Cell-based text editing
  - Method: Click the cell size selector in the toolbar to enter the cell-based text editing mode and select a word or sub-word. Use Ctrl-C/V for copy and paste. Use the Esc key to exit the cell-based text editing mode.
  - *Task #1*

Plan	Click the toolbar button to switch to the cell-based editing mode. Ctrl-click two 'Input' tokens. Type 'output' and press the Enter key. Exit the cell-based editing mode using the 'Esc' key.
Operation	M(find) P(to-button) M(verify) BB(click) M(verify) M(find) P(to-input) M(verify) BB(click) M(verify) M(find) P(to-input) M(verify) K(Ctrl) BB(click) M(verify) H(to-kbd)

	T(6:,output) M(verify) K(enter) M(verify) K(esc) M(verify)
Est. Time	K x 10 + M x 12 + P x 3 + B x 6 + H x 1 = **22.98 sec**

- *Task #2*

Plan	Click the toolbar button to switch to the cell-based editing mode. Ctrl-click four 'os' tokens. Type ',ws,arch,ln' and press the Enter key. Exit the cell-based editing mode using the 'Esc' key.
Operation	M(find) P(to-button) M(verify) BB(click) M(verify) M(find) P(to-os) M(verify) BB(click) M(verify) M(find) P(to-os) M(verify) BB(click) M(verify) M(find) P(to-os) M(verify) BB(click) M(verify) M(find) P(to-os) M(verify) BB(click) M(verify) H(to-kbd) T(11:,ws,arch,nl) M(verify) K(enter) M(verify) K(esc) M(verify)
Est. Time	K x 13 + M x 18 + P x 5 + B x 10 + H x 1 = **34.80 sec**

3.8 Discussion of Error-preventive Code Editing

3.8.1 Estimated Time Savings

Estimated time savings by cell-based text editing were 45% and 76% in the first and second tasks respectively (Figure 3.25). The time savings were larger on the second task because the task requires writing three additional similar code fragments while the first task requires writing only one similar code fragment. It reflects the fact that, in cell-based text editing, only keystrokes for typing an additional token are all that is necessary to add an additional similar code fragment. Therefore, the new user interface should be more attractive when programmers have a longer block of similar code fragments to write.

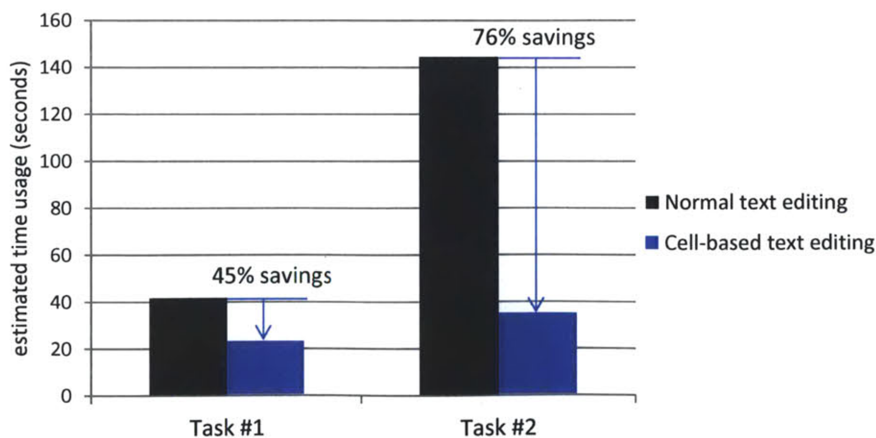


Figure 3.25: A comparison of estimated time usage between normal and cell-based text editing.

3.8.2 Comparison with Existing Code Generation Methods

Programmers use various code generation tools to work around repetitive text editing. Most of them are specific to certain predefined code generation scenarios, such as generating getter/setter methods or generating a placeholder class implementing a certain interface; therefore, they cannot be generally used for automating various forms of source code that programmers generate through code template reuse. However, tools based on a markup-language-based approach, such as Editor Template of Eclipse [14], are worthy of comparison because they can generate the same source code as the proposed user interface for code template reuse. When using the markup-language-based approach, programmers generate code by first writing a *Code-Template,* which is a piece of source code annotated with expressions in a markup language, and then passing the Code-Template to a code generator, which evaluates annotated expressions and outputs text of code.

There are notable differences in efficiency between the proposed approach and the mark-up-language-based approach. The markup-language-based approach requires users explicitly to write a Code-Template, which is a time-consuming task involving many editing steps. It is typically achieved by copy-pasting the first block of similar code fragments into a Code-Template editor and replacing code tokens that need to be different in each code fragment with annotations in a markup language. For instance, to perform the second task in the predictive evaluation, users would have to write a Code-Template similar to the one in Figure 3.26. Creating code by passing the Code-Template and parameter values to a code generator is also time-consuming. To perform the step in Eclipse [14], users need to select a desired Code-Template from a list of available Code-Templates, invoke the Insert Template command to open the dialog in Figure 3.27, and finally specify all parameter values by clicking and typing in the text fields for individual parameters. In addition, the markup-language-based approach requires that users should learn the syntax of a markup language.

```
if (${name_1} != null)
    properties.put(OSGI_${name_2}, ${name_1});
else
    properties.put(OSGI_${name_1}, CatchAllValue.singleton);
```

Figure 3.26: An example of Code-Template necessary for code template reuse when using the markup-language-based approach.

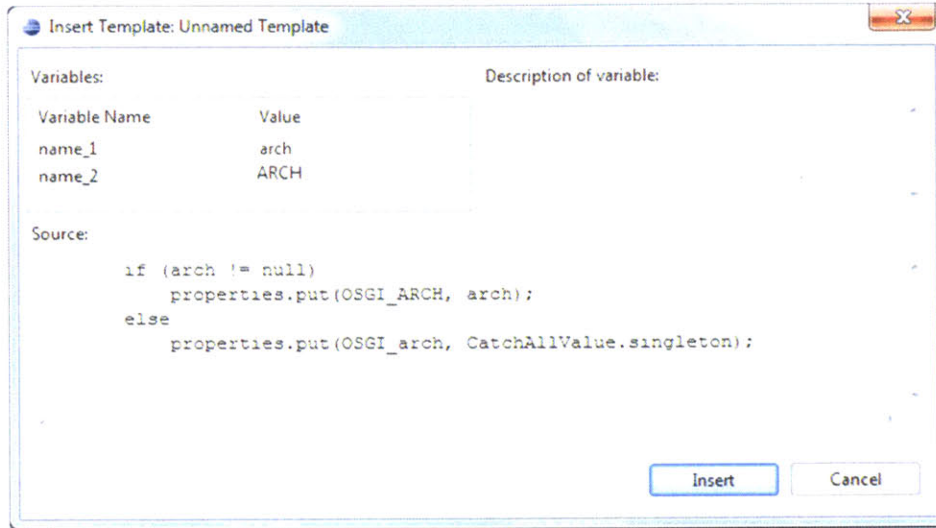


Figure 3.27: A dialog for inserting a Code-Template in Eclipse.

Because of the overhead of creating and inserting a Code-Template, the markup-language-based approach is considered not effective for code template reuse. The proposed code generation user interface addresses the efficiency problem because it does not require explicitly writing any code template, handles case-type conversion automatically, and takes all new code tokens through a single text field.

3.8.3 Multiple-Selection Copy-Paste Support for Handling Many New Code Tokens

Manually typing many new code tokens in a comma separate format is difficult and prone to errors. The proposed code generation user interface supports multiple-selection copy-paste to address the usability issue. The copy-paste-based approach is particularly effective because new code tokens may already exist in the source code. For example, in the Task #2, text for the new code tokens, *ws, arch, nl*, could have been copy-pasted from source code. The copy-paste procedure is shown in Figure 3.28. Note that the mouse-drag multiple-selection technique introduced in is Section 3.7.1.3 is used in the first step to select semi-aligned variable name tokens.

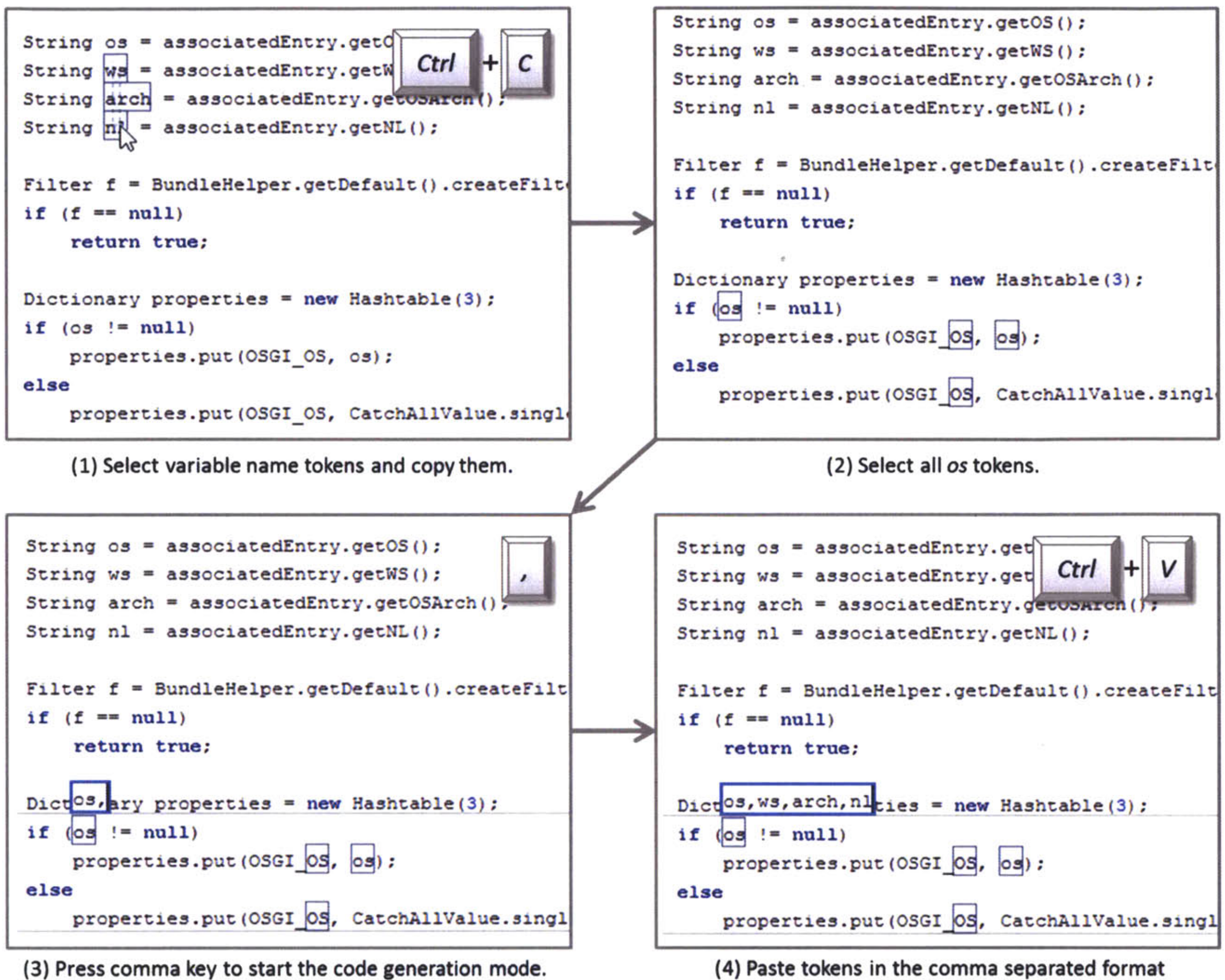


Figure 3.28: New code token text can be copy-pasted from multiple selections in a comma separated format.

3.8.4 Generation of Sequence-driven Code with More than One Code Token Sequences

The proposed user interface for code template reuse can also be used for generating sequence-driven code with more than one code token sequence. In such a case, users need to generate sequence-driven code based on one of the code token sequences. Then users need to modify code tokens corresponding to each remaining code token sequence. For example, source code in Figure 3.29 has two code token sequences: (*add remove*) and (*attach detach*). Because (*add remove*) appears more frequently than (*attach detach*), a user may choose to generate sequence-driven code based on the frequent code sequence. Then users can replace *attach* in the method name with *detach* to complete the task.

```
public void attachButton (AbstractButton button) {
    button.addActionListener(this);
    button.addMouseListener(this);
    button.addMouseMotionListener(this);
}

public void detachButton (AbstractButton button) {
    button.removeActionListener(this);
    button.removeMouseListener(this);
    button.removeMouseMotionListener(this);
}
```

Figure 3.29: An example of sequence-driven code with two unique code token sequences, adapted from original code found in NetBeans. A program bug in the original code has been fixed in this example.

Modifying code tokens corresponding to each remaining code token sequence can be error-prone if many code tokens need to be modified consistently. The cell-based text editing supports a multiple-selection replace command, called *type-to-replace*, which can accelerate common code token replacement tasks during code template reuse. Figure 3.30 shows how the replace command can be used to replace three code tokens in two different case types in a consistent and efficient manner. The *type-to-replace* command is activated when a user starts typing over existing selections. When the user has completed typing text for replacement, a user presses the Enter key to apply the change. The case types of destination code tokens are preserves by the replacement command.

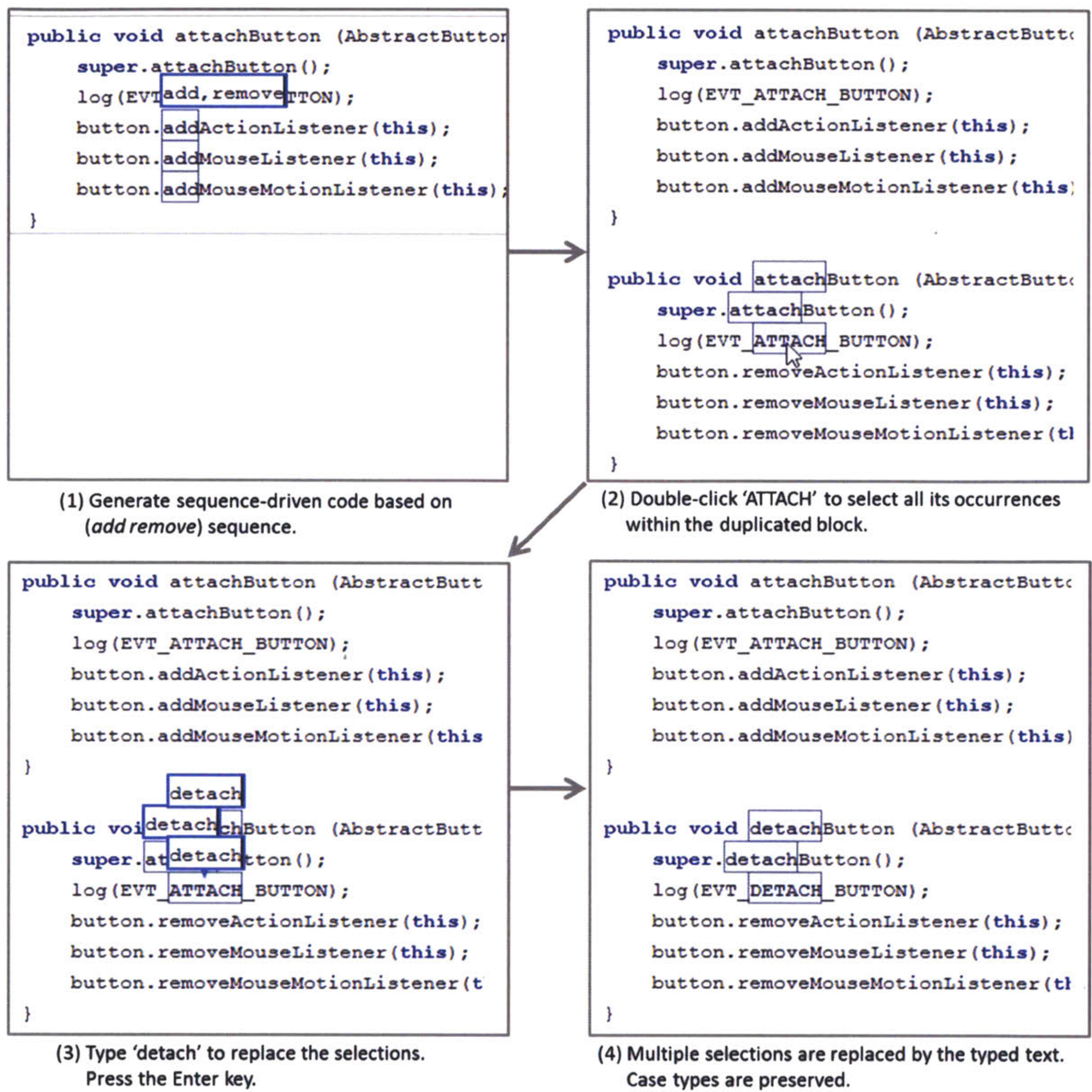


Figure 3.30: Cell-based text editing supports a multiple-selection replace command, called type-to-replace, which can accelerate common text replacement tasks during code template reuse.

Chapter 4

Effective Code Phrase Reuse: Code Completion of Multiple Keywords from Abbreviated Input

Chapter 1 has mentioned that reuse phrase code by typing and code completion poses a threat to productivity. This chapter presents a code phrase completion method that can facilitate efficient reuse of code phrases and address the productivity limitation.

4.1 Overview

Many programmers use code completion to accelerate code-writing through reduced keystrokes—avoiding typing the whole character sequence of keywords [15]. This chapter describes a method for accelerating code completion still further by completing multiple keywords at a time based on non-predefined abbreviated input, utilizing frequent keyword patterns learned from a corpus of existing code. For example, Figure 4.1 shows a user entering *ch.opn(n);*, which is translated into a list of code completion candidates that includes *chooser.showOpenDialog(null)* as the most likely candidate. Entering slightly different abbreviations such as *cho.opdlg(nl);* or *cs.sopd(nu);* should also lead to the same best candidate because the system accepts non-predefined abbreviations of keywords.

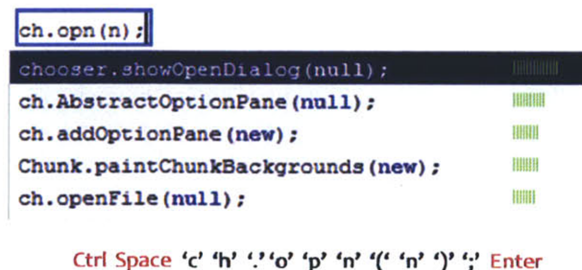


Figure 4.1: Abbreviation Completion can complete multiple keywords from abbreviated input in a single code completion dialog.

In addition to code completion by disabbreviation, the new code completion system supports prediction of the next keywords and non-alphanumeric characters of a code completion candidate, a technique called code completion by extrapolation. A user can extrapolate the current selection in a code completion candidate list by pressing the Tab key. For example, Figure 4.2 shows a user extrapolating one of the code completion candidates, *SwingUtilities.*, to get a list of extended code completion candidates, such as *SwingUtilities.invokeLater(new Runnable()*.

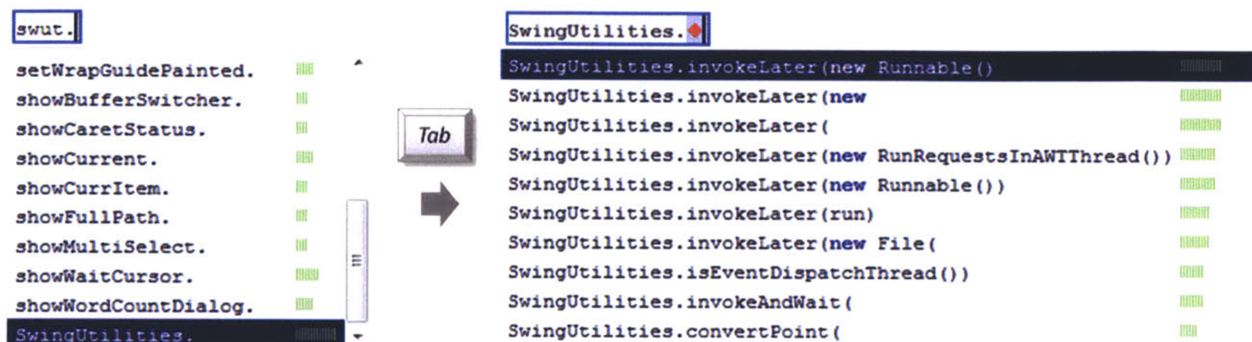


Figure 4.2:A user can extrapolate a code completion candidate by pressing the Tab key. The system updates the list of code completion candidates to display possible extensions of the code completion candidate. Note that the system sorts the candidate list in an alphabetical order when there is only one keyword being code completed, as shown in the candidate list on the left; otherwise, the system sorts the candidate list by the likelihood, as shown in the candidate list on the right. In any sort order, a candidate with the highest likelihood gets the default selection.

The system aims to address the following limitations of the conventional code completion systems:

- First, conventional systems complete only one keyword at a time. Therefore, the number of extra keystrokes increases in proportion to the number of completed keywords.
- Second, conventional code completion systems find a keyword based on an exact match of leading characters. Because the leading parts of keywords are often identical among candidates while the ending parts are more distinguishable, as in *showOpenDialog* and *showSaveDialog*, programmers often need to type potentially lengthy sequences of leading characters before they type distinctive characters close to the end.
- Third, conventional code completion systems put the default selection of code completion candidate on the first candidate in an alphabetically sorted candidate list. It leads to additional Up/Down Arrow keystrokes to adjust the selection to a correct candidate. This problem might get worse when using multiple-keyword code completion systems because of the large number of potential multiple-keyword code completion candidates.

Here are a few motivating examples in which the new code completion system is used to expand an abbreviated expression into a full expression:

what you type...		*what you get...*
`pv st nm`	→	`private String name`
`gval(r,c)`	→	`getValueAt(row,col)`
`f(i i=0;i<ls.sz();i++)`	→	`for(int i = 0; i < list.size(); i++)`
`pb st v m(st[] ag)`	→	`public static void main(String[] args)`
`swut.later(n run()`	→	`SwingUtilities.invokeLater(new Runnable()`

This chapter presents new models and algorithms, based on a Hidden Markov Model (HMM) and an *n*-gram language model, which have been integrated into a new code completion system called *Abbreviation Completion*. An HMM and an *n*-gram language model have been applied to various engineering domains [12]; however, Abbreviation Completion is believed to be their first application to the code completion domain. This chapter also presents a new user interface for interactive multiple-keyword code completion because different usability concerns arise than in single keyword completion, such as manually overriding some of the keywords suggested by the system and displaying the system's confidence on code completion candidates.

Key contributions of the new multiple-keyword code completion method are:

- A new application of an HMM to support code completion by disabbreviation of multiple keywords. An HMM has been extended in consideration of two distinctive characteristics of our problem domain.
- Development of a regression model for estimating a probabilistic distance between an abbreviated keyword and an original keyword. This model is essential for the system to handle non-predefined abbreviations. A total of 4857 abbreviations collected from human volunteers were used to train the regression model.
- Development of an *n*-gram model of programming language for code completion by extrapolation. The *n*-gram model is used to find the most likely next keywords and non-alphanumeric characters of a given code completion candidate.
- A user interface for interactive multiple-keyword code completion. A code editor supporting Abbreviation Completion has been implemented to demonstrate the new user interface. It can be downloaded from: http://www.sangmok.org/abbreviation-completion.
- Evaluation of the system's accuracy, time savings, and keystroke savings. Accuracy of code completion by disabbreviation was measured by counting how many of the code lines sampled

from open source projects could be reconstructed from abbreviated input. The system achieved average 99.3% accuracy against 4919 lines of code from six open source projects. Accuracy of code completion candidate extrapolation was also evaluated; the system achieved 95.7% and 81.8% accuracy in the prediction of the next one or two keywords using code completion by extrapolation. Time savings and keystroke savings were measured by comparing time usage and the number of keystrokes of the system with those of a conventional code completion system. The system achieved average 30.4% savings in time and 40.8% savings in keystrokes in a user study with eight participants.

The next section presents statistics collected from open source projects that demonstrates a large number of code phrases are potential targets of code reuse by typing and code completion, motivating the development of a multiple-keyword code completion system for efficient code phrase reuse. Then the following section describes models and algorithms for multiple-keyword code completion. An HMM-based model and algorithm for expanding abbreviated input is discussed first. A discussion of an n-gram model for extrapolating a code completion candidate follows. We then present estimation methods to learn probability distributions required by the models. We describe a user interface for Abbreviation Completion. The setup and results of two evaluations follow.

4.2 Groundwork: Value and Feasibility Estimation

This section presents two corpus studies that let us estimate practical value and feasibility of an efficient code phrase reuse system, which was done as groundwork before the actual development of algorithms and models.

4.2.1 Reuse Potential of Code Phrases

Programmers reuse some code phrases—those code phrases could be what they think of as idiomatic expressions in a natural language or could be what they have to write to follow usage patterns of common programming resources—more frequently than others. As a result, a relatively small set of frequently reused code phrases may account for a high percentage of code phrase occurrences. To quantitatively evaluate the degree of code phrase reuse, we defined *reuse potential* as a percentage of code phrase occurrences that can be accounted for by a certain small fraction of the most frequent code phrases and measured reuse potential of six open source projects in Java, described in Section 4.6.1.1. The reuse potential provides a rough estimate of how useful a system that can facilitate efficient code phrase reuse would be.

Based on the number of unique code phrases and the number of code phrase occurrences reported in Table 4.1, reuse potential values averaged on six open source projects are presented in Table 4.2. Reuse potential is expected to vary depending on the length of code phrases, so the table (Table 4.2) reports reuse potential for the three different lengths of code phrases: 2-keyword, 3-keyword, and 4-keyword code phrases. Note that, in this thesis, the term *keyword* refers to any alphanumeric token in source code, which can be any of the identifier, reserved word, numeric literal, or string literal. The three lengths were chosen based on the assumption that reuse potential for code phrases with the three lengths would be most interesting because they are more likely to be targets of code phrase reuse than longer code phrases, which programmers may choose to reuse by copy-paste. Reuse potential also varies depending on the fractional size of the most frequent code phrases; the larger the fractional size is, the higher percentage of code phrase occurrences will be accounted for by a fraction of the most frequent code phrases of that size. The table reports reuse potential for three different fractional sizes of 1%, 3%, and 5%.

The results shows that a significant portion of code phrase occurrences can be accounted for by a small fraction of frequent code phrases; for instance, 35.9% of all 3-keyword code phrase occurrences are accounted for by 3% of the most frequent 3-keyword code phrases. Because high reuse potential can result from retyping and copy-paste of code phrases, as well as from other ad-hoc code reuse approaches, the results should not be interpreted as all of code phrases having been created solely by retyping. However, high reuse potential still motivates development of a system for efficient code phrase reuse. In many code phrase reuse scenarios, copy-paste-based code reuse is difficult and time-consuming because copy-paste-based code reuse not only requires programmers to recall the location of desired code phrases, but it also incurs overheads for source code navigation and text selection.

Table 4.1: Number of unique code phrases and the number of code phrase occurrences, whose values were averaged on six open source projects.

Code phrase length	Number of unique code phrases	Number of code phrase occurrences	Occurrences by 1% of the most frequent code phrases	Occurrences by 3% of the most frequent code phrases	Occurrences by 5% of the most frequent code phrases
2-keywords	52,125	163,375	59,257	76,877	86,184
3-keywords	58,997	122,437	33,061	43,981	50,173
4-keywords	49,086	87,133	19,816	26,898	31,128

Table 4.2: Reuse potential averaged on six open source projects.

Code phrase length	Reuse potential by 1% of the most frequent code phrases	Reuse potential by 3% of the most frequent code phrases	Reuse potential by 5% of the most frequent code phrases
2-keywords	36.3%	47.1%	52.8%
3-keywords	27.0%	35.9%	41.0%
4-keywords	22.7%	30.9%	35.7%

4.2.2 N-Gram Entropy Comparison: Java vs. English

An n-gram entropy of a natural language is defined as an average number of binary digits required per word given previous *N* words [27] . The *n*-gram entropy can be used to measure predictability of a natural language: a low *n*-gram entropy value indicates high predictability. By modeling source code as a sequence of the keyword-connector pairs, the *n*-gram entropy definition can be extended to a programming language—Section 4.3.4.1 presents a detailed description of this modeling technique. The extended definition of an n-gram entropy makes it possible to compare predictability of a natural language with that of a programming language.

A corpus study of six Java open source projects and two English novels shows that Java has lower 2- and 3-gram entropy than English, as shown in Table 4.3 and 4.4. Given the large difference—on average, 3-gram entropy of English is 46% larger than that of Java—such a trend may hold for other programming languages, especially those which have similar syntax as Java. The finding that a programming language may be no less predictable, if not more predictable, than a natural language, seems promising for development of an efficient code phrase completion system. It may be feasible for a well-engineered statistical model of source code to infer idiomatic code phrases from a small but essential amount of ambiguous input, just like statistical models of English words enable speech recognition from noisy voice signals and enables auto-complete and auto-correction for text entry on mobile devices.

Table 4.3: *n*-gram entropy of two English novels.

English	2-gram entropy	3-gram entropy
Alice In Wonderland by Lewis Carroll	5.3	1.6
Far from the Madding Crowd by Thomas Hardy	5.3	1.6
Average	5.3	1.6

Table 4.4: *n*-gram entropy of six Java open source projects.

Java	2-gram entropy	3-gram entropy
DNSJava	3.3	0.9
Carol	2.9	0.9
RSS Owl	3.2	1.1
JEdit	3.9	1.0
JRuby	3.8	1.3
TV-Browser	3.8	1.2
Average	3.5	1.1

4.3 Model and Algorithm

This section first describes an HMM-based model and algorithm to resolve the most likely keyword sequences from abbreviated input. Two structural extensions of an HMM and a modified backtracking mechanism of the Viterbi algorithm will be highlighted. Then an *n*-gram model for extrapolating a code

completion candidate is discussed.

4.3.1 Code Completion of Multiple Keywords as a Decoding Problem of a Hidden Markov Model

An HMM is a graphical model that concerns two sequences. Only one of these sequences, called the observation symbol sequence, is observable while the other sequence, called the hidden state sequence, is the one of interest. Decoding of an HMM refers to a process of finding the most likely state assignment to a hidden state sequence based on an observation symbol sequence. The Viterbi algorithm [12] is an efficient dynamic programming algorithm for decoding an HMM.

Code completion of multiple keywords based on abbreviated input can be modeled as a decoding problem of a particular HMM. Figure 4.3 shows an example of the HMM for decoding *str nm = th.gv(r,c)*. Given two pieces of information, abbreviated input by a user and a set of keywords collected from a corpus of existing code, the two sequences of the HMM are created as follows:

First, an observation symbol sequence is created by splitting abbreviated input into a sequence of string tokens that have non-alphanumeric characters, such as spaces, dots or commas, between each one. The non-alphanumeric characters between each keyword are referred to as a *connector*. For example, the result of splitting abbreviated input *str nm = th.gv(r,c)* is shown in Figure 4.3. Note that, in our implementation for Java language code completion, the underscore character was not treated as a splitting non-alphanumeric character because it is a valid character for the identifier name.

Second, a hidden state sequence is created by sequentially arranging keywords that have the same non-alphanumeric characters between each one, as in the observation symbol sequence. Given the setup for creating the two sequences from abbreviated input and keywords collected from a corpus of existing code, finding the most likely code completion becomes a problem of finding the most likely assignment of keywords to a hidden state sequence.

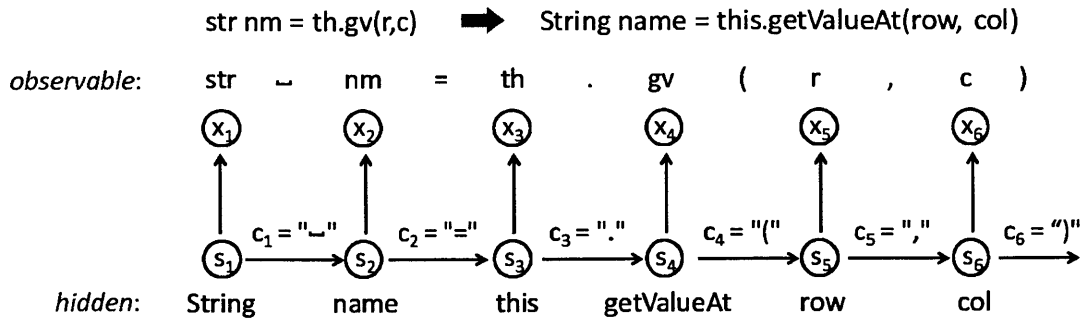


Figure 4.3: Resolution of multiple keywords is solved as a decoding problem of Hidden Markov Model.

4.3.2 An Extended Hidden Markov Model for Code Completion by Disabbreviation

We have extended a standard HMM [12] in two ways to take the following characteristics of our problem domain into account:

- Keyword transition always occurs through non-alphanumeric characters, called a connector. For example, in Figure 4.3, transition from *String* to *name* occurs through a space character, and transition from *name* to *this* occurs through an equal character.
- The number of observation symbols is not finite because users are allowed to abbreviate keywords in their own non-predefined way.

The first extension is intended to exploit the keyword transition characteristics by modeling transition probability distributions in a more sophisticated way than conventional HMMs. Unlike conventional HMMs that employ a single transition probability distribution for any type of transition, the HMM for code completion by disabbreviation employs different transition probability distributions depending on the type of transition, identified by non-alphanumeric characters at the transition.

As a result, the transition probability distribution becomes dependent not only on a previous keyword but also on non-alphanumeric characters. Generally, allowing transition probabilities to be conditioned by a greater number of surrounding states results in better prediction, although more data is necessary for training. We assume that users can find training data that is large enough to take advantage of this extension because they have access to source code in their existing projects or in open source projects.

The second extension is to resolve issues with estimating emission probabilities, the probability of a hidden state generating an output symbol. Because the number of possible observation symbols is

infinitely many, it is impossible to develop an estimation model that is guaranteed to produce a valid probability distribution—a valid emission probability distribution must sum to one when it is integrated over all possible observation symbols. However, it is possible to work around this issue by modifying the structure of an HMM as shown in Figure 4.4. The trick is to introduce a match indicator node, whose value is one if an observation symbol (an abbreviated keyword) is a correct match of a hidden state (an original keyword) and zero otherwise.

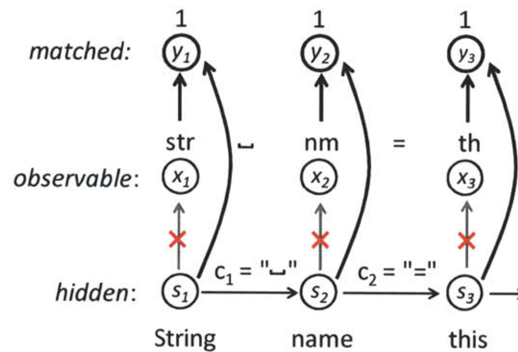


Figure 4.4:A modified HMM with match indicator nodes.

Then, we define a match probability as a probability of a match indicator becoming one given an observation symbol and hidden state pair. The match probability is equivalent to the emission probability in the sense that both measure a probabilistic distance between an abbreviated keyword and an original keyword. As we will see in the next section (Section 4.3.3.1), the match probability is computationally equivalent to the emission probability, so it can replace the emission probability in the Viterbi algorithm. Once we have replaced the emission probability with the match probability, it is straightforward to develop a valid estimation model for the match probability because it is a probability of finite, binary events. For a formal description, we will use the following notation to characterize the HMM for code completion by disabbreviation:

s_t	The value of a state at time t. The value of a state is a keyword.
S	A set of all possible values of a state.
x_t	The value of an observation symbol at time t. The value of an observation symbol is an abbreviation of a certain keyword.
y_t	The value of a match indicator at time t. The value of a match indicator is one if abbreviated keyword x_t and original keyword s_t are a correct match. The value is zero

otherwise.

c_t Non-alphanumeric characters between s_t and s_{t+1}, called a connector at time t.

$T(s)$ Start probability of state s.

$T_c(s_{t+1} \mid s_t)$ Transition probability from state s_t to state s_{t+1} through connector c.

$E(x \mid s)$ Emission probability of state s emitting observation symbol x.

$M(y \mid s, x)$ Match probability. $M(y=1 \mid s, x)$ is a probability that the value of a match indicator becomes one, which indicates given state s and observation symbol x are a correct match.

T The length of an observation symbol sequence, which must be equal to the length of a hidden state sequence.

4.3.3 Modified Viterbi Algorithm

This section presents a modified version of the Viterbi algorithm for code completion by disabbreviation. Notably, two modifications have been made: First, the emission probability is replaced with the match probability; second, the backtracking part of the original Viterbi algorithm has been replaced with a new backtracking algorithm that retrieves *N*-best candidates, instead of the single best candidate of the original algorithm.

4.3.3.1 Match Probability

The Viterbi algorithm finds the most likely hidden state sequence by calculating the joint probabilities of possible assignments of state values to a hidden state sequence. Because the joint probability distribution of the extended HMM is different from that of the original HMM (without match indicators), the Viterbi algorithm needs to be modified to accommodate the difference. The modification just replaces the emission probability with the match probability. The difference is apparent comparing the joint probability distribution of the original HMM in (4.1) with the joint probability distribution of the extended HMM in (4.2). The only difference is that the emission probability $E(x_t \mid s_t)$ is replaced with the match probability $M(y_t = 1 \mid s_t, x_t)$.

$$P(s_1 \ldots s_T, x_1 \ldots x_T, c_1 \ldots c_T) = T(s_1) \prod_{t=2}^{T} T_{c_{t-1}}(s_t \mid s_{t-1}) \prod_{t=1}^{T} E(x_t \mid s_t) \tag{4.1}$$

$$P(s_1 \ldots s_T, x_1 \ldots x_T, c_1 \ldots c_T) = T(s_1) \prod_{t=2}^{T} T_{c_{t-1}}(s_t \mid s_{t-1}) \prod_{t=1}^{T} M(y_t = 1 \mid s_t, x_t) \tag{4.2}$$

4.3.3.2 Finding the *N*-Best Candidates of Code Completion by Disabbreviation

The second modification to the Viterbi algorithm aims to enhance the way it handles backtracking, the final step used to construct the most likely state sequence using data structures calculated from a dynamic programming step. The original backtracking algorithm returns only one code completion candidate because it finds the single most likely sequence. However, a code completion system is expected to provide *N*-best code completion candidates.

Several backtracking approaches that find *N*-most likely sequences have been proposed [21, 22]. We decided to take a tree-trellis based backtracking algorithm [21] because it is an exact algorithm that requires a negligible amount of computation when compared to computation for the dynamic programming step of the Viterbi algorithm: $O(T \cdot n^2 \log(n)) << O(T \cdot N^2)$. The lowercase *n*, the number of the *N*-best candidates, is generally much smaller than the capitalized *N*, the number of all possible states.

4.3.4 An *n*-Gram Model for Code Completion by Extrapolation

This section describes an *n*-gram model of programming language for code completion by extrapolation. We first describe an extension to *n*-gram model of natural language for predicting both keywords and non-alphanumeric characters. Then we describe a search algorithm for finding the *N*-best code completion candidates. We also discuss the reasoning behind setting $N = 3$ for the *n*-gram model used in the prototype system.

4.3.4.1 An *n*-Gram Model of Programming Language

An *n*-gram model is a probabilistic model, which assumes that the probability for the next item in a sequence of items depends only on the previous $N-1$ items. An *n*-gram model is commonly used for modeling natural languages [27]. In an *n*-gram word model of natural language, a sentence is modeled as a sequence of words, in which the probability for the next word is assumed to be dependent only on the previous $N-1$ words. An *n*-gram word model can be used to predict the next word [27] or the next several words [25] of a given sequence of words.

We introduce an *n*-gram model of programming language, which models source code as a sequence of the keyword-connector pairs. Note that a connector refers to non-alphanumeric characters between each keyword. For example, source code *this.getValueAt(row, col)* is modeled as a sequence of *this.*,

getValueAt(, row,, and *col).* A keyword-connector pair may have an empty keyword or an empty connector because some code lines may start without a keyword or may end without a connector.

This setup allows us to use the same technique used for predicting the next word in an *n*-gram word model for predicting the next pair of a keyword and a connector. Notably, Equation (4.3) is used to find the most likely keyword-connector pair, denoted as w_{t+1}, following a sequence of keyword-connector pairs, denoted as $w_1 w_2 \cdots w_t$.

$$\underset{w_{t+1}}{\operatorname{argmax}} \mathrm{P}(w_{t+1}|w_1 \cdots w_t) = \underset{w_{t+1}}{\operatorname{argmax}} \mathrm{P}(w_{t+1}|w_{t-N+2} \cdots w_t) \quad (4.3)$$

4.3.4.2 Finding the *N*-Best Candidates of Code Completion by Extrapolation

When using code completion by extrapolation, the system is expected to display the *N*-most likely extensions as code completion candidates. A depth-limited beam search algorithm has been implemented to generate the *N*-most likely code completion candidates.

The algorithm generates code completion candidates by appending the most likely keyword-connector pairs to a given code completion candidate. Because it is depth-limited, the algorithm does not generate a code completion candidate that is longer than a depth limit. The depth limit is 3 by default, so a maximum of 3 keyword-connector pairs can be appended to a code completion candidate. The depth limit exists to prevent the system from generating potentially inaccurate code completion candidates that are also longer than a user would normally expect. Should a user want to extend a code completion candidate further, he or she can extrapolate the candidate one more time by pressing the Tab key.

The algorithm also implements a beam search with the default beam width of 4: only the specified number of the most likely keyword-connector pairs, not the whole set of the most likely keyword-connector pairs, are appended to a code completion candidate. The beam search aims to prevent a situation in which all code completion candidates are similar to each other, given a limited number of code completion candidates, which is 10 by default, that are proposed to a user. For example, when not using a beam search, if there are more than 10 possible keyword-connector pairs that can follow *SwingUtilities.*, the most likely 10 of them should be included in code completion candidates. It will make all code completion candidates extrapolated by one keyword-connector pair and have the same length. When using a beam search with the beam width of 4, only the most likely 4 keyword-connector pairs are appended to the most likely code completion candidate of a certain length; as a result, the list of code

completion candidates can be populated with code completion candidates with several different lengths as shown in Figure 4.2.

4.3.4.3 Selecting the *N* for the n-Gram Model

We set the N for the n-gram model to 3 for the prototype system implementing code completion of extrapolation. Based on statistics collected from a corpus of source code, we expect that the system would achieve acceptable performance if it predicts the next keyword-connector pairs by looking at the previous two keyword-connector pairs. Let us use *K* and C to denote a keyword and a connector, respectively. Table 4.5 shows the numbers of all possible unique *KC* sequences, *KCKC* sequences, and *KCKCKC* sequences found in source code of six open source projects, which will be introduced in Section 4.6.1.1 for an artificial corpus study.

Table 4.5: The number of all possible unique *KC*, *KCKC*, and *KCKCKC* sequences in source code of six open source projects. *K* and *C* denote a keyword and a connector, respectively.

Project	*KC* size	*KCKC* size	*KCKCKC* size	$\frac{KCKC\text{size}}{KC\text{size}}$	$\frac{KCKCKC\text{size}}{KCKC\text{size}}$
DnsJava	6144	16735	18749	2.72	1.12
CAROL	4759	12739	13746	2.68	1.08
RSS Owl	15027	42332	49088	2.82	1.16
JEdit	29482	86670	101067	2.94	1.17
JRuby	42270	130108	161435	3.08	1.24
TV-Browser	41192	116629	138302	2.83	1.19
Average	23146	67536	80398	2.84	1.16

We first notice that the ratio between the number of all possible *KCKC* sequences and the number of possible *KC* sequences is 2.84 on average, implying that a *KC* sequence is expected to be followed by 2.84 different *KC* pairs on average. Some KC sequences, such as *this.* may be followed by many different *KC* pairs, but the low average value indicates that many of the *KC* sequences are followed by a small number of *KC* pairs. The ratio between the number of all possible *KCKCKC* sequences and the number of all possible *KCKC* sequences is 1.16, implying that a *KCKC* sequence is expected to be followed by only 1.16 different *KC* pairs on average. It suggests that looking at a *KCKC* sequence, two previous keyword-connector pairs, would provide a good clue in predicting the next keyword-connector pair. Based on this reasoning, we decided to use a 3-gram model in prototyping the extrapolation feature. Section 4.6.2.2 presents the evaluation of the prototype system.

4.4 Parameter Estimation

Given models and algorithms to compute the *N*-most likely code completions, this section describes a method to estimate model parameters. Three model parameters need to be estimated for the extended HMM with match indicators: start probability distribution, transition probability distribution, and match probability distribution. The first two probability distributions, which describe how keyword transition occurs at the beginning and in the remaining part of a sequence, are estimated from a corpus of existing code. The third match probability distribution, which describes how likely an abbreviated keyword and an original keyword are a correct match, is estimated from examples of correct matches and incorrect matches. One parameter need to be estimated for the *n*-gram model used for code completion by extrapolation: conditional probability distribution of the next keyword-connector pair given the two previous keyword-connector pairs. The conditional probability distribution is estimated by examining a corpus of existing code.

4.4.1 Estimation of Start Probabilities and Transition Probabilities

4.4.1.1 Preparation of Training Examples

Generally, a set of known state sequences is used as training examples to estimate start probabilities and transition probabilities of an HMM. In case of the HMM for code completion, a corpus of existing code is used to generate training examples. We took the following two-step approach to convert a piece of source code into a set of state sequences:

In the first step, a lexical analyzer tokenizes source code into a sequence of lexical tokens that are one of the following types: identifiers, string literals, character literals, number literals, line breaks, non-line-breaking whitespaces, comments, and non-alphanumeric characters. For example, tokenizing *String name = null;<LF>* creates lexical tokens in Table 4.5. This step allows us to identify keyword states: A lexical token of identifier type, string literal type, character literal type, or number literal type will form a keyword state in the state sequence.

Table 4.6: An example of lexical analyzer output.

lexical token	type
String	identifiers
<space>	non-line-breaking whitespaces
name	identifiers
<space>	non-line-breaking whitespaces
equal	non-alphabetical characters
<space>	non-line-breaking whitespaces
null	identifiers
<semi-colon>	non-alphabetical characters
<LF>	line breaks

In the second step, a state sequence generator scans through lexical tokens and selectively uses them to construct state sequences. Because keyword states were already identified in the first step, this step aims to identify connectors by concatenating non-alphanumerical characters and removing redundant whitespaces. A state sequence is extended by appending keyword states and connectors, one next to the other. The beginning of a new state sequence is signaled by a line or block separator, such as a semi-colon or curly bracket character in Java, followed by a line break. For example, lexical tokens of *String name = null;<LF>* constructs the following state sequence in (4.4), in which ϕ indicates the end of a state sequence:

$$String \xrightarrow{<space>} name \xrightarrow{<equal>} null \xrightarrow{<semi-colon>} \phi \tag{4.4}$$

The implementation of the two-step approach is language-specific because it involves lexical analysis. However, it is often possible to customize a lexical analyzer to handle a different language simply by updating regular expressions so that they match language-specific lexical tokens.

4.4.1.2 Learning Maximum Likelihood Estimates

Given state sequences extracted from a corpus of existing code, the maximum likelihood estimates of start probabilities and transition probabilities are calculated by counting the number of state transitions. Let us define $\mathrm{COUNT}(i, s \xrightarrow{c} s')$ as the number of transitions from state s to state s' through connector c in the ith training example and $\mathrm{COUNT}(i,s)$ as the number of occurrences of state s in the ith training example.

The maximum likelihood estimate of transition probabilities from state s to state s' through connector c is calculated as the ratio of the number of transitions from state s to state s' through connector c to the number of transitions from state s through connector c:

$$\hat{T}_c(s'|s) = \frac{\sum_i \mathrm{COUNT}(i, s \xrightarrow{c} s')}{\sum_i \sum_{s'} \mathrm{COUNT}(i, s \xrightarrow{c} s')} \tag{4.5}$$

The maximum likelihood estimate of start probabilities at state s is calculated from the ratio of the number of occurrences of state s to the number of occurrences of any state, as shown in (4.6). Notice that the number of occurrences is used instead of the number of beginning transitions. This is justified by the fact that a user can invoke code completion at any location in a code line. Therefore, the first keyword in a code completion does not necessarily represent its occurrence at the beginning of a code line, but can represent its occurrences at any location.

$$\hat{T}(s) = \frac{\sum_i \mathrm{COUNT}(i,s)}{\sum_i \sum_s \mathrm{COUNT}(i,s)} \tag{4.6}$$

In practice, transition probabilities based on exact counting, as in equation (4.5), may not generalize well because the number of training examples may not be sufficiently large to estimate transition probabilities of all possible state combinations. A popular technique to address this issue is to use the weighted sum of transition probabilities and occurrence probabilities as transition probabilities. A weight coefficient λ is set to 0.7 by following a common practice. Finally, the actual transition probability in the system is shown in equation (4.7).

$$\tilde{T}_c(s'|\, s) = \lambda \cdot \hat{T}_c(s'|\, s) + (1-\lambda)\hat{T}(s) \tag{4.7}$$

4.4.2 Estimation of Match Probabilities

4.4.2.1 An Estimation Model For Match Probabilities

The match probability indicates the probability of a match event, which occurs when an abbreviated keyword and an original keyword are a correct match. The estimation of match probabilities can be challenging because there are infinitely many combinations (when counting both correct and incorrect combinations) of abbreviated keywords and original keywords; therefore, should someone try to estimate match probabilities by directly counting the occurrences of match events, the person may need an infinite number of training examples.

We propose an approach based on a logistic regression model that predicts match probabilities from a set of similarity features between an abbreviated keyword and an original keyword. In this approach, an abbreviation pair—an abbreviated keyword and an original keyword—is represented as a feature vector, elements of which describe different aspects of similarity, such as the number of consonant matches or the percentage of matched letters. Table 4.7 shows a list of similarity features used in the latest

implementation of Abbreviation Completion. Given a feature vector encoding similarities between an abbreviated keyword and an original keyword, the logistic regression model is expected to give a value close to one if the abbreviation pair is a good match, and a value close to zero otherwise.

Table 4.7: Similarity features for estimating match probabilities.

feature name	feature description
$Sim_1(s,x)$	number of consonant letter matches
$Sim_2(s,x)$	number of vowel letter matches
$Sim_3(s,x)$	number of non-alphabet character matches
$Sim_4(s,x)$	number of capital letter matches
$Sim_5(s,x)$	number of letter matches with ordering ignored
$Sim_6(s,x)$	percentage of matched consonant letters
$Sim_7(s,x)$	percentage of matched vowel letters
$Sim_8(s,x)$	percentage of matched capital letters
$Sim_9(s,x)$	percentage of matched letters with ordering enforced
$Sim_{10}(s,x)$	percentage of matched letters with ordering ignored

The feature-vector-based representation of abbreviation pairs allows us to use standard machine learning techniques to develop an estimation model of match probabilities. Notably, a logistic regression model fits well in our problem because it can learn probabilities of a binary event, like the match event.

4.4.2.2 Preparation of Training Examples

To train a logistic regression model, a set of positive training examples and a set of negative training examples are necessary. To prepare positive training examples, we first collected a set of 500 keywords randomly from source code of JEdit, one of the open source projects used introduced in Section 4.6.1.1 for an artificial corpus study. Then about 10 abbreviations for each keyword—4857 abbreviation pairs in total—were generated from 92 human volunteers who were recruited from an online service marketplace called Amazon Mechanical Turk (AMT; http://www.mturk.com). Additionally, 500 abbreviation pairs whose abbreviated keywords were the same as the original keywords were added to the pool of positive training examples because an unabbreviated form is actually one of the possible forms of an abbreviated keyword. Finally, 5357 abbreviation pairs of positive training examples were converted into a feature vector representation based on similarity features described in Table 4.7.

The procedure for collecting abbreviations on the AMT is as follows. The randomly chosen 500 keywords were divided into 50 subsets each containing 10 keywords. For each subset, we posted a work request on the AMT for writing abbreviations of 10 keywords. Each work request was performed by 10 unique AMT users to generate 10 abbreviations for each keyword. For example, for keyword *getProperty*, we could collect various abbreviations such as *gp*, *gprt*, *gprop*, *getprop*, and *gtproprty*.

Note that the work requests instructed that AMT users could abbreviate keywords any way they like as long as an abbreviation met the following two criteria: (1) an abbreviation consists of letters that appear in the original keyword and (2) an abbreviation is at least two characters long. In about 24 hours, a total of 92 AMT users completed all work requests, and 5000 abbreviation pairs were collected. Then, the collected abbreviation pairs were inspected to ensure that they met the specified criteria. 4857 valid abbreviation pairs were saved after discarding invalid abbreviation pairs that did not meet the criteria.

Negative training examples were generated by pairing an original keyword with a randomly selected, wrong abbreviation. A total of 5357 wrong abbreviation pairs were generated to be used as negative training examples and converted into a feature vector representation.

4.4.2.3 Learning Maximum Likelihood Estimates

Having collected 10714 positive and negative training examples, 25% of the examples (2679 abbreviation pairs) were held for testing and 75% (8035 abbreviation pairs) were used for training. The logistic regression model has 11 unknown parameters, denoted as $\beta_0, \beta_1, \cdots, \beta_{10}$, because 10 similarity features are included in the model. The logistic regression model for match probabilities is shown in equation (4.8):

$$M(y=1 \mid s,x) = g(\beta_0 + \beta_1 \cdot Sim_1(s,x) + \cdots + \beta_{10} \cdot Sim_{10}(s,x)) \text{ where} g(z) = \frac{1}{1+e^{-z}} \quad (4.8)$$

The maximum likelihood estimates of $\beta_0, \beta_1, \cdots, \beta_{10}$ were calculated using a generalized linear model regression function in a statistics package. The values of the explanatory variables were normalized before calculating the estimates to be in the range within 0 and 1, so the size of the regression coefficient of each explanatory variable can be interpreted as the size of the effect that variable has on the match probability. The result of parameter estimation is shown in Table 4.8. The train error and test error of the logistic regression model were recorded as 2.60% and 2.42%, respectively. Because the test error is slightly lower than the train error, the model is not expected to have an issue with overtraining.

Table 4.8: Estimated parameters of the logistic regression model of the match probability.

β_0	β_1	β_2	β_3	β_4	β_5	β_6	β_7	β_8	β_9	β_{10}
-11.2	17.5	14.6	13.7	11.1	-15.1	2.0	-2.4	8.8	38.5	-36.0

4.4.2.4 Discussion of The Training Result

The training result reflects the relative importance of different similarity features, although it is specific to a certain style of abbreviation exhibited by our training data. Let us first look at the regression coefficients of the first five similarity features ($\beta_1, \cdots, \beta_5$) that are based on the number of letter matches. Among four positive regression coefficients, we notice that the regression coefficient of the number of consonant matches (β_1) is the largest, indicating that an abbreviation pair with many consonant letter matches would be given a high match probability value. It reflects a common practice of abbreviating a word by eliminating vowels.

The regression coefficient for the number of letter matches with ordering ignored (β_5) has a relatively large negative value. It reflects the fact that the number of letter matches with ordering ignored can be falsely large for incorrect abbreviation pairs. Note that other numbers of letter matches for $\beta_1, \cdots, \beta_4$ are counted with ordering enforced. For example, given keyword *getProperty*, abbreviation *pg* has only one consonant matches because the first letter of the abbreviation, *p*, is matched by the capital *P* in the keyword, but the second letter of the abbreviation, *g*, is not matched by any letters following the capital *P*. In contrast, abbreviation *pg* has two letter matches when the ordering is ignored. Note that we have not introduced a similarity feature for the number of letter matches with ordering enforced in the regression model because the number of letter matches is a sum of already introduced features: the number of vowel matches and the number of consonant matches.

Then let us look at regression coefficients for the last five similarity features, which are all percentage based. β_9 for the percentage of matched letters with ordering enforced has the largest positive value while β_{10} for the percentage of matched letters with ordering ignored has the largest negative value. Again, this reflects the fact that the percentage of matched letters can get falsely large if the ordering is ignored. We also notice that regression coefficient β_8 for the percentage of matched capital letters is larger than β_6 and β_7, regression coefficients for the percentage of matched consonant and vowel letters. This implies that people tend to include many of the capital letters in an original keyword when they generate an abbreviated form of the keyword.

It is noteworthy that different users may have different preferred ways of abbreviating keywords, and even the same user may not abbreviate consistently over time. If such differences necessitate retraining of match probability parameters for each user or at any point of time, it would make the Abbreviation

Completion system very expensive to use. However, the parameter values in Table 4.5 were found to work reasonably well for an artificial corpus study and a user study in Section 4.6 and 4.7 without further retraining. This is promising because it indicates that the parameter values can serve as system defaults, which may attract users to the system. The default parameter values can later be updated from actual abbreviation examples collected from system usage. This updating is to be considered in future work.

4.4.3 Estimation of Conditional Probability Distribution of the *n*-Gram Model

We estimate conditional probability distribution of the *n*-gram model of keyword-connector pairs by counting keyword-connector sequences in a corpus of source code. The lexical analyzer and state sequence generator, described in Section 4.4.1.1, are used to convert source code into sequences of keyword-connector pairs. Let us use COUNT $(i, w_1 w_2 \cdots w_{n-1} w_n)$ to denote the number of times that keyword-connector sequence $w_1 w_2 \cdots w_{n-1} w_n$ occurs in the ith training example. Then the maximum likelihood estimate for $P(w_{t+1} \mid w_{t-N+2} \cdots w_t)$ can be expressed as (4.9), adapted from [27]. Since the prototype system sets $N = 3$ for the *n*-gram model, (4.9) can be written as (4.10).

$$\hat{\mathrm{P}}(w_{t+1} \mid w_{t-N+2} \cdots w_t) = \frac{\sum_i \mathrm{COUNT}(i, w_{t-N+2} \cdots w_t w_{t+1})}{\sum_i \sum_{w'} \mathrm{COUNT}(i, w_{t-N+2} \cdots w_t w')} \tag{4.9}$$

$$\hat{\mathrm{P}}(w_{t+1} \mid w_{t-1} w_t) = \frac{\sum_i \mathrm{COUNT}(i, w_{t-1} w_t w_{t+1})}{\sum_i \sum_{w'} \mathrm{COUNT}(i, w_{t-1} w_t w')} \tag{4.10}$$

A user may sometimes want to extrapolate a code completion candidate when there is only one keyword-connector pair in the candidate. A *2*-gram model of keyword-connector pairs has also been included in the prototype system to handle such an extrapolation request. The maximum likelihood estimate for conditional probability distribution of the *2*-gram model is shown in (4.11).

$$\hat{\mathrm{P}}(w_{t+1} \mid w_t) = \frac{\sum_i \mathrm{COUNT}(i, w_t w_{t+1})}{\sum_i \sum_{w'} \mathrm{COUNT}(i, w_t w')} \tag{4.11}$$

A user may also want to extrapolate a code completion candidate without having to type an ending

connector. For example, code completion candidate *System.out* may be extrapolated by a user—notice that the last keyword-connector pair, the keyword part of which is *out*, is missing a connector, which would normally be a period character. The prototype system incorporates another *n*-gram model for predicting the next connector based on the previous one keyword and two connectors to handle such missing connectors. The system automatically adds the most likely connector to the code completion candidate and then uses the fixed candidate for extrapolation.

Finally, based on the same reasoning behind using a weighted sum of transition probabilities and occurrence probabilities as transition probabilities, we use a weighted sum of conditional probabilities of the 3-gram model and conditional probabilities of the 2-gram model as conditional probabilities of the 3-gram model, as shown in (4.12). A weight coefficient is set to 0.9 for the prototype system, which is higher than commonly used 0.7, because we wanted the weighted conditional probabilities to be largely determined by the original conditional probabilities of the 3-gram model when the 3-gram model probabilities are non-zero.

$$\tilde{\mathrm{P}}(w_{t+1} \mid w_{t-1}w_t) = \lambda \cdot \hat{\mathrm{P}}(w_{t+1} \mid w_{t-1}w_t) + (1-\lambda) \cdot \hat{\mathrm{P}}(w_{t+1} \mid w_t) \tag{4.12}$$

4.5 User Interface and Implementation

This section describes a user interface for multiple-keyword code completion. Three design requirements have been identified for the user interface: acceptance of abbreviated input that may include non-alphanumeric characters including spaces; allowance for users to override the system's suggestion; and effective sorting of code completion candidates. The user interface has been implemented on a demonstrational code editor using Java Swing GUI framework.

4.5.1 User Interface

It is important to support a scenario in which some part of a code line is typed, while another part is completed using Abbreviation Completion. For example, a user may first type *this.getValueAt()*and then try to complete *row, col* inside the parentheses from abbreviated input *r,c*. The demonstrational code editor utilizes an input popup that floats over the code editing area, shown as a light blue rectangle in Figure 4.5. The area accepts abbreviated input, including non-alphanumeric characters. The input popup appears at the current caret position when a user presses Ctrl+Space, so it can be used to insert completed

code at any location in the code. Since the input popup initializes its content with highlighted text in the code editing area, a user may type abbreviated input in the code editing area and expand it by highlighting and pressing Ctrl+Space. The user may also use Ctrl+Shift+Space, a shortcut for highlighting text in the current code line and opening the input popup simultaneously.

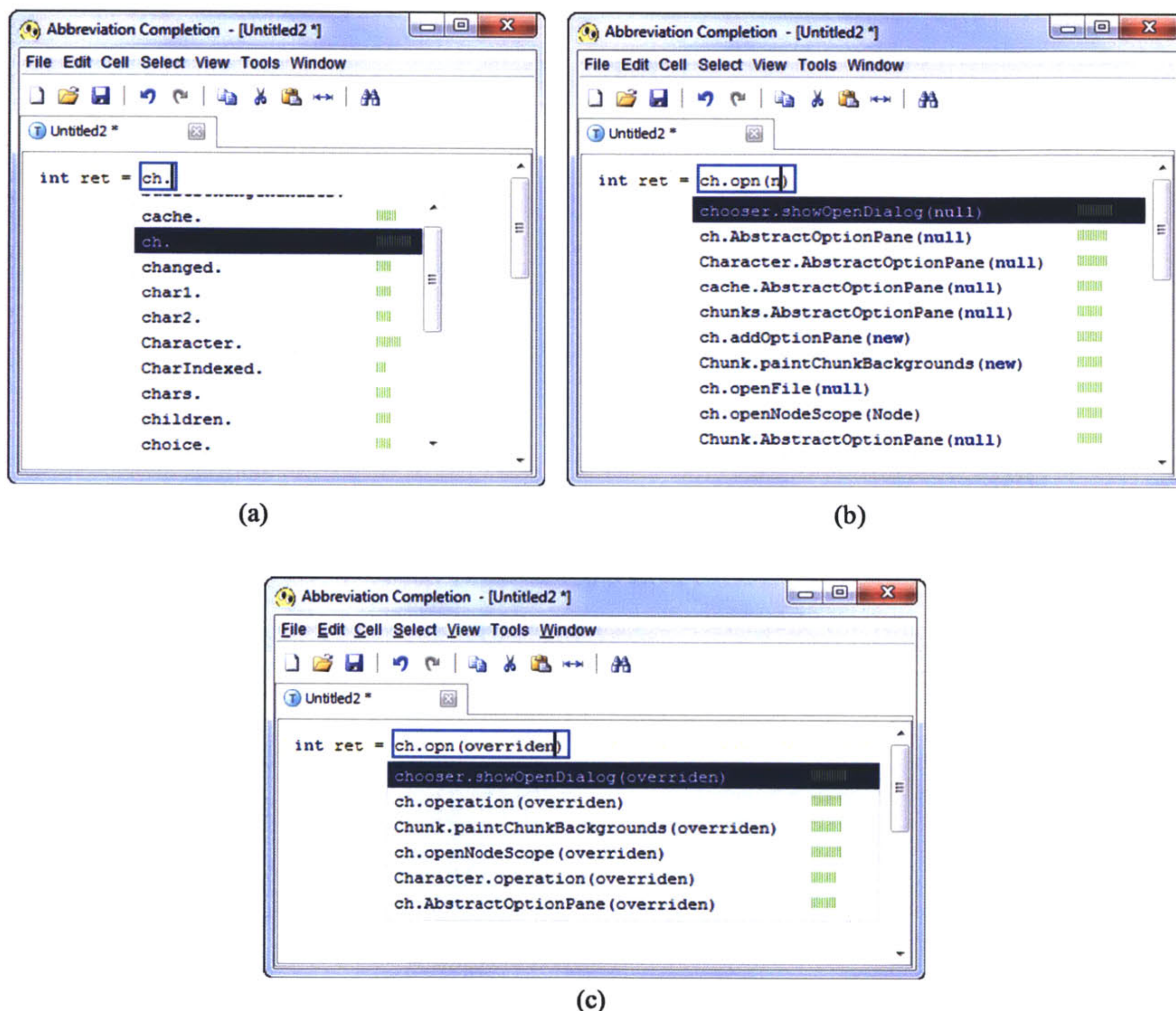


Figure 4.5: A user interface for multiple-keyword code completion: (a) the list is alphabetically sorted when there is one keyword, (b) the list is sorted by the likelihood when there are more than one keyword, and (c) a user can override the system's suggestion.

The second requirement is handled by a keyword-pinning capability, which is invoked using a keyboard shortcut Ctrl+B. The system highlights a pinned keyword in the input popup by making it boldface with a light gray background, as shown in Figure 4.5c. Internally, the system treats a pinned keyword as an

observation symbol that matches all possible states with an equal probability. Once code completion candidates are generated using the match probabilities, they are displayed to users with keyword tokens at a pinned location replaced with overriding text.

Regarding the third design requirement, users are assumed to have two concerns when using the code completion candidate list. First, users want to know which candidates are more likely to be correct and should be examined first. A list sorted by the likelihood is useful for this purpose. Second, users want to navigate the list of candidates efficiently in a predictable manner. An alphabetically sorted list is useful for this purpose. We implemented the code completion candidate list such that it can be sorted one of the two ways depending on the number of keywords in the input popup. When there is only one keyword, the list is sorted alphabetically; the system's default suggestion can be often incorrect because the transition pattern cannot be utilized. When there is more than one keyword, the list is sorted by the likelihood; two keywords are often enough to locate the correct candidate in the top-10 list, so users may want to examine candidates from the most likely one. Figure 4.5a and Figure 4.5b demonstrates the behavior of the candidate list. In any sort order, the likelihood of a code completion candidate is displayed using a green bar-shaped icon next to each code completion candidate: a longer bar indicates higher likelihood of a code completion candidate.

4.5.2 Incremental Feedback of Code Completions

Responsive incremental feedback is essential for the usability of the multiple-keyword code completion system. We applied a filtering technique to improve responsiveness of the system. The filtering technique effectively reduces the search space of the Viterbi algorithm by removing some of the states that are impossible to appear in the most likely candidates. Filtering based on characters and connectors (the non-alphanumeric characters between keywords) has been implemented. For example, given abbreviated input "sys.", the system first applies character-based filtering so that only the states that have both 's' and 'y' characters remain. Then the system applies connector-based filtering so that only the states that have a transition through '.' remain.

4.5.3 Incremental Indexing of Source Code

The demonstrational code editor supports incremental indexing of source code using a background thread. The code editor monitors changes of source code, which may occur inside the code editor or on the file system, and updates the HMM to reflect the changes. As a result, Abbreviation Completion can be used to

complete code lines that may include recently introduced variable or method names. A full indexing of 400 source code files (3000 kilobytes in size) usually takes less than 3 seconds on a laptop computer with Intel Core 2 Duo P8400 CPU and 3 gigabyte ram. Therefore, an incremental indexing of a few changed entries can be processed in a negligible time compared to a normal lag between code edits. A configuration file is used to specify the target directories for incremental indexing in the prototype system.

4.6 Artificial Corpus Study

The artificial corpus study aims to evaluate the accuracy of the Abbreviation Completion system. Accuracy of two code completion methods has been evaluated: code completion by disabbreviation and by extrapolation.

To evaluate code completion by disabbreviation, a total of 4919 frequent code lines were collected from 6 open source projects. The code lines were converted into acronym-like abbreviations by applying a particular text transformation rule. We measured how many of the original code lines could be completed from the abbreviated code lines. We report top-*N* accuracy, which refers to the rate of finding the original code line within the top-*N* candidates of code completion.

To evaluate code completion by extrapolation of a code completion candidate, we converted the 4919 frequent code lines into a set of code completion candidates by truncating keyword-connector pairs at the end. We measured how many of the original code lines could be completed by extrapolating the code completion candidates. Top-*N* accuracy is measured for the code completion by extrapolation.

4.6.1 Study Setup

4.6.1.1 Selection of Open Source Projects

Six open source projects were selected from 14 open source projects that were used in a previous artificial corpus study of Keyword Programming in [16].

The six open source projects are: DNSJava, an implementation of DNS in Java; CAROL, a library for using different RMI implementations; JEdit, a source code editor implemented in Java; JRuby, a Java implementation of the Ruby programming language; RSSOwl, a news reader application for RSS feeds;

and TV-Browser, a Java-based TV guide application.

4.6.1.2 Preparation of Original Code Lines

The 1000 most frequent code lines, which are at least 20 characters long, include at least 2 keyword tokens, and appear at least 2 times in an open source project, were collected from each open source project; a total of 4919 code lines was collected from six open source projects. The total of code lines is less than 6000 because two of the open source projects, DNSJava and CAROL, had only 455 and 464 code lines meeting the given criteria.

Frequent code lines were selected since they are more likely to be a target of multiple-keyword code completion. The minimum length requirement is introduced to exclude short code lines that are not likely to be targets of multiple-keyword code completion. It also helps to include more of the challenging, longer code lines in the test set. Code lines are required to have at least 2 keyword tokens because we are interested in evaluating the code completion of multiple keywords.

4.6.1.3 Preparation of Abbreviated Code Lines

Given the large number of code lines, we decided to generate their abbreviations using an artificial abbreviation generator, implemented as a computer program. The artificial abbreviation generator creates acronym-like abbreviations with the maximum length of three characters, such as *bao* abbreviated from *ByteArrayOutputStream*; *th* abbreviated from *throw*; and *i* abbreviated from *if*. The artificial abbreviation generator takes the following steps to transform a keyword into an abbreviation:

- First, it determines the target length of an abbreviation to be created, which is always between one and three characters based on (4.13):

$$\text{Target length} \leftarrow \max(1, \min(3, \text{ceil}(\text{Keyword length} \times 40\%))) \tag{4.13}$$

- Second, it appends the first letter and as many capitalized letters (0, 1, or 2 letters) as possible to the abbreviation within the limit of the target length. All characters are appended in lower case.
- Third, if the abbreviation is still shorter than the target length, append letters following the first letter one by one until the abbreviation has the target length (for *Object*, *obj*).

Table 4.9 shows examples of abbreviated code lines generated using the abbreviation generator.

Table 4.9: Examples of abbreviated code lines generated using the abbreviation generator.

Original Code Line	Abbreviated Code Line
final Ruby runtime = context.getRuntime()	*fi ru run = con.ger()*
catch(NumberFormatException nf)	*ca(nfe n)*
if (TraceCarol.isDebugJndiCarol())	*i (trc.idj())*
URLConnection connection = url.openConnection()	*url con = ur.opc()*
buttons.add(Box.createHorizontalStrut(6))	*but.ad(bo.chs(6)*
return SetResponse.ofType(SetResponse.NXDOMAIN)	*ret ser.oft(ser.nxd)*

Using this computer-based approach, we limit ourselves to testing the system against the particular style of abbreviation with potential biases. However, acronym-like abbreviation is believed to be one of popular ways of abbreviating keywords and therefore the result of this study may provide a reasonable estimate of the system's performance against human-generated abbreviations. The claim is supported by the fact that 26.1% (1269 of 4587) of abbreviations collected from human participants in Section 4.4.1.1 were exactly the same as abbreviations generated by the computer-based abbreviation generator.

Because we imposed the maximum length limit of three characters on the abbreviation generator as an effort to make a conservative estimate, computer-generated abbreviations tended to be shorter than human-generated abbreviations. For example, the abbreviation generator abbreviated *isMultipleSelectionEnabled* to *ims* while some human participants abbreviated it to *imse*. Actually, 39.5% (1917 of 4957) of abbreviations collected from human participants were the same as or prefixed by computer-generated abbreviations.

4.6.1.4 Preparation of Code Completion Candidates for Extrapolation

We evaluate the accuracy of the code completion by extrapolation in two scenarios, which are different by the number of keyword-connector pairs to be predicted. The first is a scenario in which a user completes the next one keyword-connector pair based on the two previous keyword-connector pairs. The second is the same as the first scenario except that a user completes the next two keyword-connector pairs.

We generate code completion candidates for extrapolation using the 4919 frequent code lines collected from six open source projects. Code completion candidates are generated by truncating different numbers of keyword-connector pairs at the end of a code line. A total of 8181 and 5362 code completion candidates were generated for the first and second scenarios respectively.

The procedure of generating code completion candidates for extrapolation can be summarized as follow:

- For each code line, repeat the following two steps after converting the code line into a sequence

of keyword-connector pairs. Lexical analysis for the conversion is described in Section 4.4.1.1.

- Second, scan the sequence with a sliding window of size 3 from left to right. Code completion candidates for the first scenario are generated by concatenating all keyword-connector pairs left to the sliding window and the first two keyword-connector pairs in the sliding window. The third keyword-connector pair in the sliding window is saved to be used as a correct prediction of the next one keyword-connector pair.
- Third, scan the sequence with a sliding window of size 4. Code completion candidates for the second scenario are generated in the same way as those for the first scenario by concatenating all keyword-connector pairs left to the sliding window and the first two keyword-connector pairs in the sliding window. The last two keyword-connector pairs in the sliding window are saved to be used as a correct prediction of the next two keyword-connector pairs.

For example, given code line *private CellConstraints cc = new CellConstraints(),* three code completion candidates are generated for the first scenario: *private CellConstraints*, *private CellConstraints cc =* and *private CellConstraints cc = new* . Two code completion candidates are generated from the code line for the second scenario: *private CellConstraints* and *private CellConstraints cc =*. Note that The prototype system is based on a 3-gram model and only the ending two keyword-connector pairs have an effect on the prediction; therefore, the extrapolation of *private CellConstraints cc = new* gives the same prediction as the extrapolation of *cc = new* .

After generating code completion candidates, we remove some of them from a test set if the last two keywords of a code completion candidate are both reserved keywords in Java, such as *public*, *int*, and *while*. We assume that users use code completion by extrapolation when they think that a code completion candidate is specific enough to be extrapolated into a useful expression. Based on the assumption, code completion candidates that end with two reserved keywords, such as *static void* or *private int*, are excluded from a test set because they are not likely to be a target of extrapolation.

4.6.1.5 Test Procedure

To measure top-*N* accuracy of code completion by disabbreviation, we repeated the following steps for each of the six open source projects:

- Train an HMM from the source code.
- Expand abbreviated code lines using the HMM. Count the number of successful disabbreviation.
- Calculated top-*N* accuracy by dividing the occurrences of successful disabbreviation within top-*N*

candidates by the number of code completion invocations.

Top-*N* accuracy of code completion by extrapolation is measured in a similar way. The following steps were repeated for each open source project:

- Train an *n*-gram model from the source code.
- Extrapolate code completion candidates for the first scenario using the *n*-gram model. Count the number of successful extrapolation.
- Extrapolate code completion candidates for the second scenario using the *n*-gram model. Count the number of successful extrapolation.
- Calculated top-*N* accuracy by dividing the occurrences of successful extrapolation within top-*N* candidates by the number of code completion invocations.

4.6.2 Results and Discussion

4.6.2.1 Top-*N* Accuracy of Code Completion by Disabbreviation

The top-10 accuracy of code completion by disabbreviation against 4919 code lines from six open source projects was 99.3%. The top-5, top-3 and top-1 accuracies were 97.7%, 95.6%, and 80.1%, respectively. Table 4.10 shows accuracy values of individual open source projects.

There are two positive findings about the system's performance on accuracy. First, the accuracy itself is remarkably high, over 99%. Although the accuracy is measured against a particular style of acronym-like abbreviations, such a high accuracy shows potential for achieving similarly high accuracy against human-generated abbreviations. Second, the system's accuracy is consistent across the six open source projects, which may involve different class libraries, use different naming conventions, and exhibit different code patterns. Figure 4.6 shows that there is no noticeable difference in top-*N* accuracy across the projects.

Table 4.10: Top-*N* accuracy of code completion by disabbreviation.

Project	top-10	top-5	top-3	top-1	test set size	time per code completion
DnsJava	100.0%	98.9%	97.8%	83.5%	455	26 ms
CAROL	99.6%	98.1%	95.9%	78.7%	464	32 ms
RSS Owl	99.8%	98.1%	95.7%	82.0%	1000	68 ms
JEdit	99.2%	97.9%	96.0%	83.3%	1000	139 ms
JRuby	98.4%	96.3%	94.5%	79.1%	1000	278 ms
TV-Browser	98.8%	96.8%	93.4%	74.1%	1000	437 ms
Average	99.3%	97.7%	95.6%	80.1%	820	163 ms

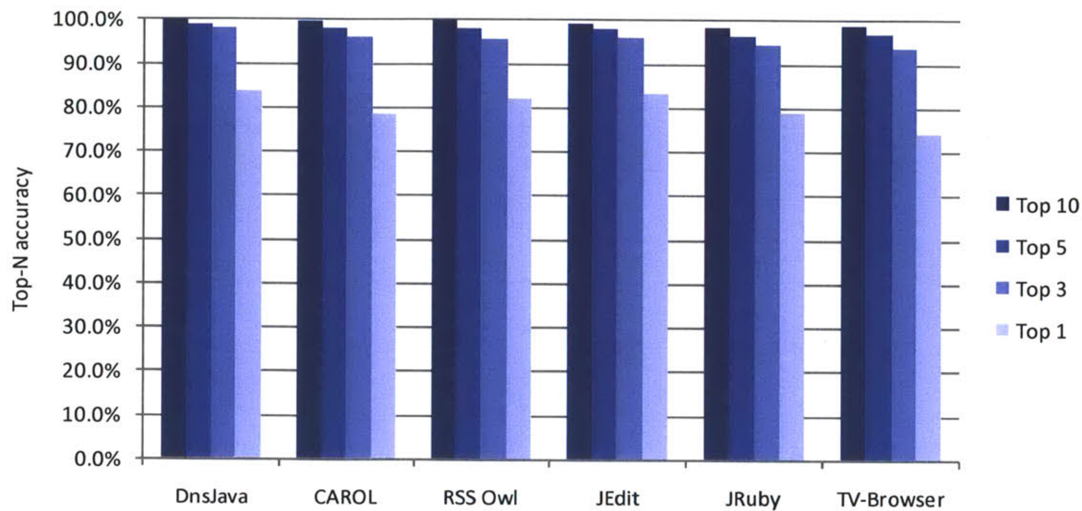


Figure 4.6: The system gives consistent top-N accuracy across the open source projects in code completion by disabbreviation.

4.6.2.2 Accuracy of Code Completion by Extrapolation

The top-10 accuracy of code completion by extrapolation was 95.5% on average in the prediction of the next one keyword-connector pair and 81.6% on average in the prediction of the next two keyword-connector pairs. The accuracy was measured by extrapolating 8181 and 5362 code completion candidates in the prediction of the next one and two keyword-connector pairs, respectively. Table 4.11 and Table 4.12 show accuracy values of individual open source projects. Like the accuracy of disabbreviation, we find no noticeable difference in top-*N* accuracy across the projects (Figure 4.7). Note that top-1 accuracy in the prediction of the next two keyword-connector pairs in Table 4.12 is zero because the top-1 slot is always occupied by a code completion candidate that has been extended by one keyword-connector pair.

Overall, the accuracy is considered high enough to make the extrapolation feature useful for speeding code completion of frequently used code lines. Although the system may not be very accurate (81.6% accuracy) at predicting two keyword-connector pairs at a time, the system's high accuracy in predicting one keyword-connector pair may be useful to work around a situation in which the system fails to propose desired two keyword-connection pairs. The first of the two keyword-connector pairs is likely to be found in top-10 code completion candidates with 95.5% accuracy on average. A user may extrapolate the code completion candidate with the first keyword-connector pair one more time to complete the desired two

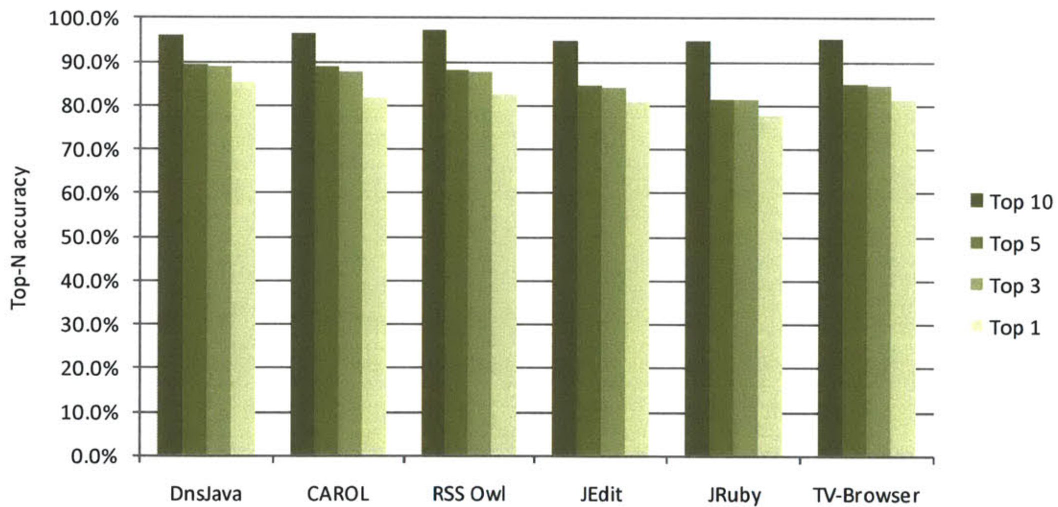
keyword-connector pairs. More than two keyword-connector pairs can be completed one by one using the approach, too.

Table 4.11: Top-*N* accuracy of code completion by extrapolation in predicting next one keyword-connector pair.

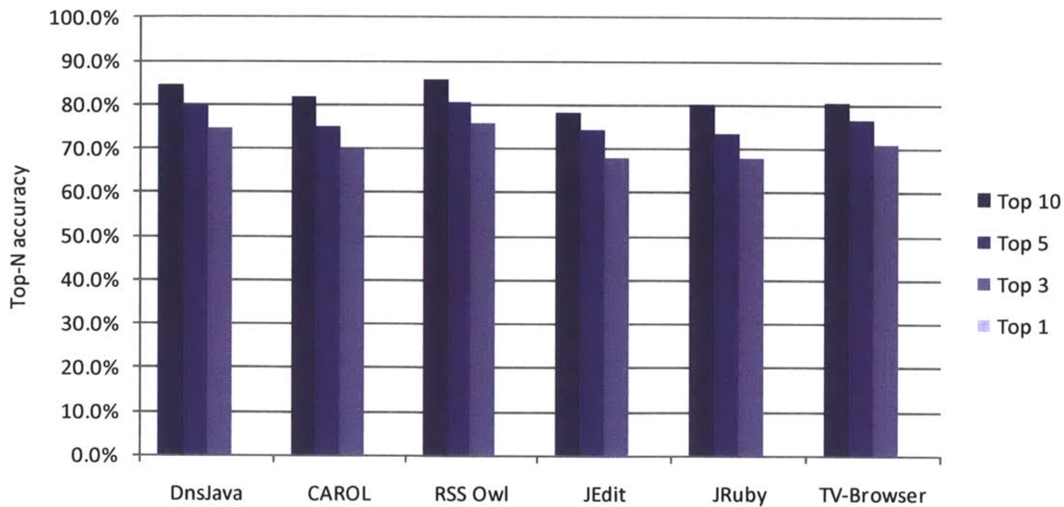
Project	top-10	top-5	top-3	top-1	test set size	time per code completion
DnsJava	95.8%	89.3%	88.9%	85.1%	840	15 ms
CAROL	96.5%	88.9%	87.7%	81.9%	855	13 ms
RSS Owl	97.2%	87.9%	87.6%	82.6%	1501	41 ms
JEdit	94.8%	84.3%	84.2%	80.7%	1641	70 ms
JRuby	94.9%	81.4%	81.3%	77.9%	1929	103 ms
TV-Browser	95.2%	84.9%	84.7%	81.5%	1415	109 ms
Average	95.7%	86.1%	85.7%	81.6%	1364	59 ms

Table 4.12: Top-*N* accuracy of code completion by extrapolation in predicting next two keyword-connector pairs.

Project	top-10	top-5	top-3	top-1	test set size	time per code completion
DnsJava	84.6%	79.6%	74.8%	0.0%	539	15 ms
CAROL	81.7%	75.1%	69.9%	0.0%	575	13 ms
RSS Owl	85.8%	80.4%	75.8%	0.0%	997	40 ms
JEdit	78.3%	74.0%	67.7%	0.0%	1058	68 ms
JRuby	80.0%	73.3%	67.9%	0.0%	1280	99 ms
TV-Browser	80.5%	76.5%	71.1%	0.0%	913	101 ms
Average	81.8%	76.5%	71.2%	0.0%	894	56 ms



(a)



(b)

Figure 4.7: Top-*N* accuracy of code completion by extrapolation in predicting (a) the next one keyword-connector pair and (b) the next two keyword-connector pairs.

4.6.2.3 Time per Code Completion

Code completion by disabbreviation took 0.17 seconds per code completion on average over 4919 code lines. The right-most column in Table 4.10 shows the average code completion time of six open source projects. Code completion time varies considerably across the projects. It is because the time complexity of code completion increases in proportion to the number of keywords and the number of transitions.

Code completion by extrapolation took less time than code completion by disabbreviation. It reflects the fact that the time complexity of code completion by extrapolation is linearly proportional to the number of all possible unique keyword-connector pairs in the prototype system. Code completion by extrapolation took 0.06 seconds per code completion on average. There is no time difference in two test scenarios—the first one predicting one keyword and the other one predicting two keywords—because the prototype system of the same configuration was used in either test scenario: code completion by extrapolation could append up to three keyword-connector pairs to the code completion candidate. Code completion time of six open source projects is shown in Table 4.11 and Table 4.12.

Both code completion methods took less than 0.5 seconds in all six open source projects on average. Therefore, the responsiveness of the current implementation is considered acceptable. However, there is some room for improvement because one of the projects, TV-Browser, required almost 0.5 second per

code completion, which is a noticeable lag.

4.6.2.4 Number of Resolved Keywords per Code Completion

When evaluating the accuracy of code completion by disabbreviation, the number of keywords in each code line was recorded to inspect how many keywords were expanded from an abbreviated code line per code completion. A histogram in Figure 4.8 shows that resolution of 3 to 5 keywords was most frequent. The average number of keywords was 4.1. In case of code completion by extrapolation, the number of keywords extrapolated from a code completion candidate was one or two by design of our test setup.

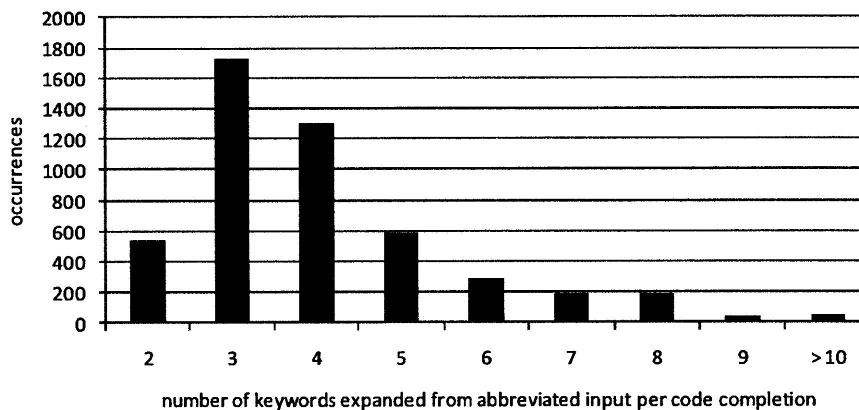


Figure 4.8: The histogram of the number of keywords expanded from abbreviated input per code completion.

4.6.2.5 Statistics about Hidden Markov Models

Table 4.13 shows statistics of HMMs trained by source code of six open source projects. The number of code lines is measured by counting effective code lines ignoring blank lines and comments. One interesting finding is that the ratio between the number of transitions and the number of keywords is about 5 in all six open source projects. This implies that a graph connecting keywords (nodes) through transitions (edges) is very sparse because a keyword is connected to just five keywords on average among many possible keywords.

Table 4.13: Statistics of HMMs for code completion by disabbreviation.

Project	Files	Code lines	Keywords (a)	Transitions (b)	Ratio $\frac{(b)}{(a)}$
DnsJava	123	35658	2963	12678	4.28
CAROL	157	44166	2601	9844	3.78
RSS Owl	412	147162	7577	32921	4.34
JEdit	394	233908	13807	66857	4.84
JRuby	677	320124	20415	101991	5.00
TV-Browser	852	348942	19953	90840	4.55
Average	436	188326	11219	52521	4.47

4.6.2.6 Inspection of Unsuccessful Code Completions

There were 40 unsuccessful code completions by disabbreviation among 4919 trials, in which none in the top-10 candidates was the original code line. Two common failure types were identified by inspecting them.

Failure Type I: This type of failure is caused by the *new* keyword in Java language. Not only many keywords (states) make a transition to the new keyword (a state) but the *new* keyword also makes a transition to many keywords. Because transition probabilities of the first-order HMM like ours are conditioned by only one previous state, such a universally connected previous state cannot provide useful guidance in decoding. This failure type applies to 11 of 40 failures.

Failure Type II: This type of failure is caused by similar keywords making similar transitions. In this study, we have restricted the maximum length of an abbreviated keyword to three characters. The HMM tries to resolve ambiguity introduced by such a short abbreviation using a transition pattern. However, there are cases in which similar keywords make transitions in a similar fashion. Then it becomes difficult for the HMM to locate the original code line within the top-*N* candidates. For example, a code line } *catch (Exception e1)* could not be resolved from its abbreviation *} ca (exc e)* because there were other similar keywords making similar transitions such as *} catch (Exception e)* or } *catch (IOException e).* This type of failure potentially applies to all failures.

4.7 User Study

The user study focuses on evaluating time savings and keystroke savings when a programmer uses code completion by disabbreviation, which is referred to as Abbreviation Completion for brevity. Note that code completion by extrapolation was introduced after we performed the user study; therefore the feature was not used by participants in the user study. The time usage and the number of keystrokes needed in the

Abbreviation Completion system are compared with those needed in a conventional code completion system in Eclipse, a popular Java development tool. We report that time savings and keystroke savings were 30.4% and 40.8% and the difference in time and keystrokes was statistically significant.

4.7.1 Participants

Eight Java programmers were recruited using flyers and a mailing list in a college campus. They were informed that the user study would take about 30 minutes and one of the participants would be awarded a $25 gift certificate. There were six males and two females among the participants. The average age of the participants was 28.1. All of them had a minimum of 5 years of general programming experience. Five people had used Eclipse for more than 3 years while three people had not used it or used it just briefly because they used different Java development tools, which they confirmed have a code completion capability similar to Eclipse.

4.7.2 Usage Scenario and Assumptions

We are interested in evaluating code completion systems in a particular test scenario, in which a programmer writes lines of code based on a concrete idea of what needs to be written. That is, a programmer can write multiple keywords without having to stop to ponder about the next keyword. To simulate such a code-writing scenario, we decided to provide our subjects with a visual reference of code lines, which was always visible on the computer screen. We assumed that such a visual reference could work as an external memory, which would enable our subjects to type multiple keywords continuously as if they had what needs to be written in their minds. We also assumed that using a visual reference would not slow down code-writing significantly as long as subjects were familiar with the code lines in the visual reference.

4.7.3 Study Setup

The user study was performed at an office area in a college campus using a computer with a full-sized keyboard. One subject, assisted by one experiment facilitator, performed a set of code-writing tasks at each run of the user study.

The two code completion systems under investigation are called Abbreviation Completion and Eclipse Code Completion. To counterbalance the effect of trying one system first and the other later, we separated subjects into two groups. The first group, named Abbreviation-First, started using Abbreviation

Completion first while the second group, named Eclipse-First, started using Eclipse Code Completion first.

4.7.4 Tasks

The user study starts with the first task of learning two code completion systems. Let us assume that a subject from the Abbreviation-First group performs the first task. After the facilitator explains how to use Abbreviation Completion, the facilitator lets the subject practice using Abbreviation Completion. A subject is allowed to ask questions during the practice. For practice, the subject is required writes code lines in Figure 4.9 using Abbreviation Completion. Once the subject finishes writing the code lines, the facilitator explains how to use Eclipse Code Completion. A same practice session follows using Eclipse Code Completion.

```
JPanel content = new JPanel(new BorderLayout())
content.add(BorderLayout.CENTER, panel)
public void actionPerformed(ActionEvent evt)
label.setHorizontalAlignment(SwingConstants.CENTER)
GridBagLayout layout = new GridBagLayout()
cons.anchor = GridBagConstraints.WEST
label.setBorder(new EmptyBorder(0,0,0,12))
fireTableRowsUpdated(row,row)
SwingUtilities.invokeLater(new Runnable())
StringBuffer buf = new StringBuffer()
```

Figure 4.9: Code lines used in the user study to measure time usage and keystrokes needed for code-writing using two code completion systems.

The second task is a recording session to record time usage and keystrokes in writing code lines in Figure 4.9. Note that a subject is asked to write the same code lines that they already have written twice because it may help simulate the usage scenario of our interest. Let us assume that a subject from the Abbreviation-First group performs this task. The subject first writes the code lines using Abbreviation Completion. The time usage and keystrokes are unobtrusively recorded using custom-built instrumentation facilities in code editors. Once the subject finishes writing the ten code lines, the subject will have a short break and then repeat the same recording process using Eclipse Code Completion.

The ten code lines were selected from JEdit, one of open source projects used in our artificial corpus study, through a random walk of its source code to find code lines that satisfy the following

characteristics: each code line should appear at least four times in the whole project; each code line should be at least 30 characters long and at most 50 characters long; and selected code lines should demonstrate various styles of code-writing such as instantiations, assignments, declarations, member access, and parameters.

4.7.5 Results

4.7.5.1 Time Savings

The overall time savings averaged for all subjects and for all code lines was 30.4%, as shown in Figure 4.10. The detailed time usage is presented in two ways, first by averaging for all code lines (Figure 4.10) and second by averaging for all subjects (Figure 4.11). Time savings were calculated by dividing the difference of time usage in two code completion systems by the time usage of Eclipse Code Completion. The standard deviation of time savings across subjects was 8.4%, indicating that there is a certain amount of variation in time savings depending on individual subjects. The standard deviation of time savings across code lines was 6.3%. The difference in the time usage between Abbreviation Completion and Eclipse Code Completion was statistically significant based on a paired t-test ($df = 79$, $p < 0.001$).

4.7.5.2 Keystroke Savings

The overall keystroke savings averaged for all subjects and for all code lines was 40.8%, as shown in Figure 4.12, which is larger than the overall time savings. The number of keystrokes is presented in two ways, first by averaging for all code lines (Figure 4.12) and second by averaging for all subjects (Figure 4.13). The standard deviation of keystroke savings was 6.7% across subjects and 7.5% across code lines. The difference in the number of keystrokes between Abbreviation Completion and Eclipse Code Completion was statistically significant based on a paired t-test ($df = 79$, $p < 0.001$).

Unlike the time usage, the number of keystrokes has a baseline value. A baseline value is the number of keystrokes when the whole character sequence in a code line is typed without using any code completion. Black dotted lines in Figure 4.12 and Figure 4.13 show the baseline values. Multiple dotted lines are shown in Figure 4.13 because each code line has its own baseline value. Comparing average keystrokes by Abbreviation Completion (20.7 keystrokes) with the baseline value (45.8 keystrokes), we see 54% of keystroke savings. Meanwhile, average keystrokes by Eclipse Code Completion (35.0 keystrokes) is just 24% less than the baseline value.

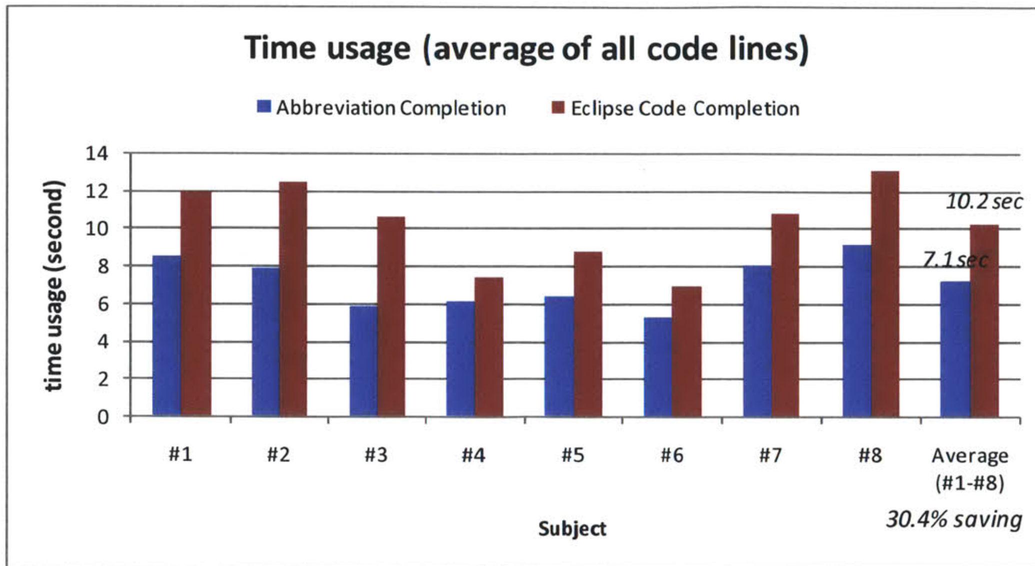


Figure 4.10:Time usage average of all code lines for each subject.

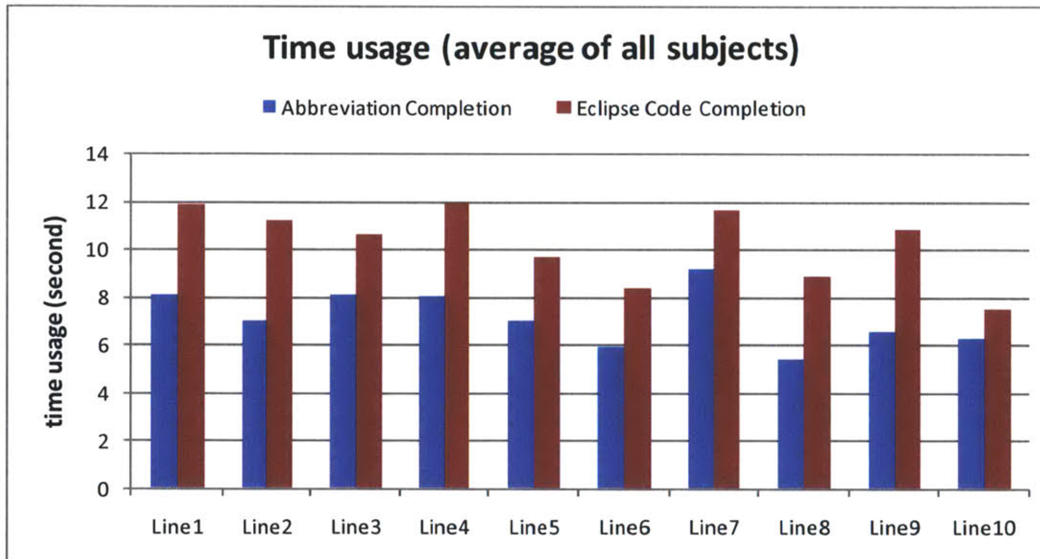


Figure 4.11: Time usage average of all subjects for each code line.

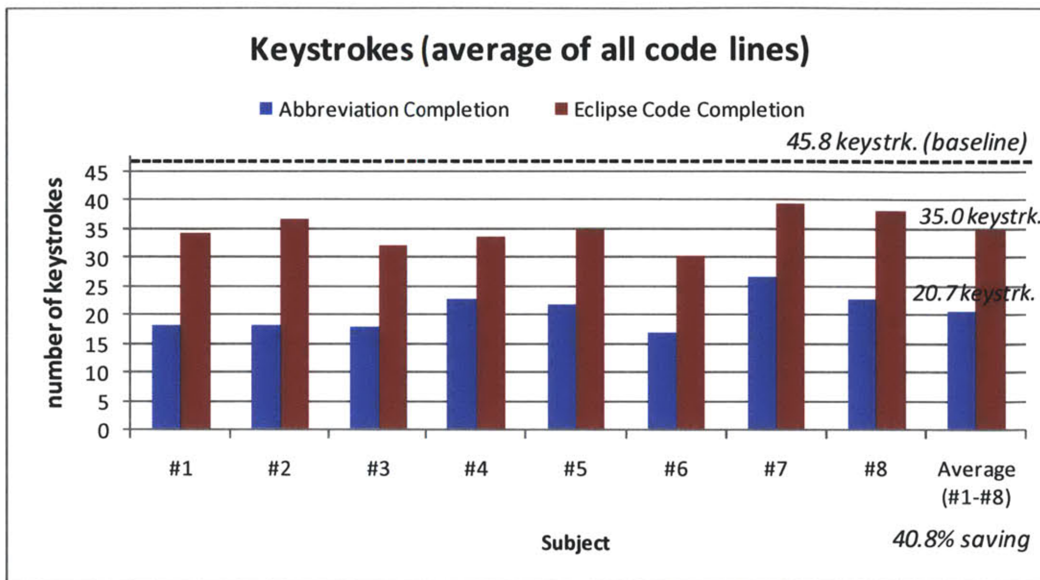


Figure 4.12:Keystrokes average of all code lines for each subject. The baseline keystrokes, also an average of all code lines, are shown as a dotted line.

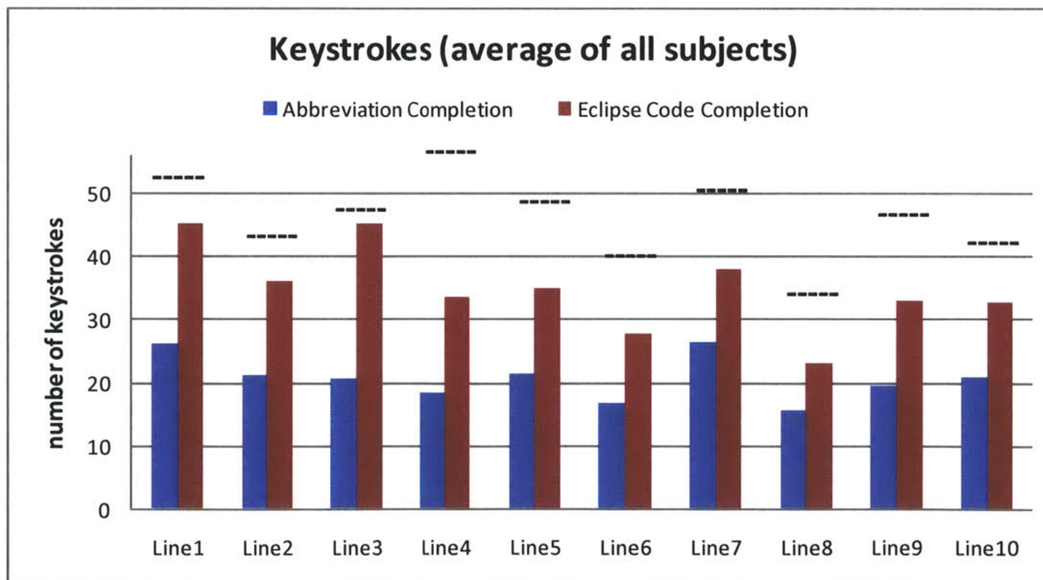


Figure 4.13:Keystrokes average of all subjects for each code line. The baseline keystrokes are shown as dotted lines.

4.7.6 Discussion

In the user study, the Abbreviation Completion system achieved substantial savings in time and keystrokes. It is noteworthy that the keystroke savings were larger than the time savings. Obviously, the time usage is not a linear function of keystrokes; it is also a function of various mental operations, which

could not be measured directly in the user study.

From our observation of subjects' behavior, one noticeably time-consuming mental operation was a validation of code completion candidates. After making some keystrokes, subjects stopped to check if the list of code completion candidates had the intended code line. Because the system showed ten code completion candidates, it could take seconds to scan through the list if the correct one did not appear near the top in the list.

One subject told us that validation was more difficult in Abbreviation Completion than in Eclipse Code Completion. The subject explained that it was because multiple keywords had to be validated all at once. Another subject told us that the subject had a desire to hit the Enter key right after typing abbreviated input without validating the candidates. He said that it was because the system's first suggestion seemed usually correct.

We think that both of subjects' comments point to a single usability issue of the Abbreviation Completion system. Information needed for validating code completion candidates is not clearly visible to users. Users only saw abbreviated input and code completion candidates. Note that, unlike the latest version of prototype system, the prototype system used for the user study did not display the likelihood of a code completion candidate. Information about how well or why a code completion candidate matches the abbreviated input was not visible to users. Also, from the very nature of code completion, it was difficult to visually compare code completion candidates with the intended code line because the intended code line was in user's mind.

To address the visibility issue, we have been working to improve the system in two ways. First, we decided to expose the system's confidence about code completion candidates. An improved version of user interface, which was implemented after the user study, displays a green bar-shaped icon next to each code completion candidate; the length of a green bar visualizes the likelihood of a code completion candidate. This allows users to compare relative likelihood of candidates and see which of the top candidates are worthy of a closer look. Second, we are investigating a way to expose information about why a code completion candidate makes a match of abbreviated input. Instead of trying to explain a complicated probabilistic model, an approximated model that can be communicated easily may serve this purpose well. A simple visualization technique, such as highlighting which part in abbreviated input matches which part in a code completion candidate, should be useful as well.

Chapter 5
Conclusions

5.1 Summary

The thesis presented models, algorithms, and user interfaces to improve effectiveness of code template reuse and code phrase reuse. The code template reuse and code phrase reuse are common ad-hoc code reuse approaches, but they pose a risk to software quality and productivity.

This thesis first presented a new bug detection method to automatically locate program bugs introduced during code template reuse. The new bug detection method, called Bugsy Finder, is developed based on two key insights. The first is that differences of structurally equivalent code fragments can be compactly expressed using a set of code token sequences crossing the code fragments. The second is that program bugs can be accurately located by suspiciously inconsistent patterns within a set of code token sequences. The code token pattern based approach to program bug detection could uncover many previously unknown program bugs in well-maintained open source projects. Specifically, 87 program bugs were found from 7 open source projects. In addition, the precision of bug detection, the ratio of the number of true bugs to the number of bug warnings, of the new bug detection method is found significantly higher (47%) than that of a previous similar method (5.7%). This thesis also presented Texel Editing, a new code editing method that can reduce the number of repetitive code editing steps for code template reuse. It introduces a cell-based text editing approach to speed selection of editing targets and supports an efficient code generation user interface for automating code template reuse. The new code editing approach was compared with normal text editing on the basis of estimated time usage in two code template reuse scenarios. Large savings in estimated editing time (45% and 76%) were reported when using Texel Editing; this is a promising finding because such savings can help programmers stay focused and make fewer errors during code template reuse.

To address the productivity limitation posed by code phrase reuse, this thesis introduced Abbreviation Completion, a new code phrase completion method that can complete multiple code tokens at a time efficiently by taking non-predefined abbreviated input and expand it into a full code phrase. The code

phrase completion method utilizes a statistical model learned from a corpus of code and abbreviation examples to infer code phrases from short, potentially ambiguous user input. The thesis also described code completion by extrapolation, a technique to further accelerate code phrase completion by predicting next code tokens of a code completion candidate. The effectiveness of the new code completion method was evaluated on the basis of accuracy and efficiency. The code completion method achieved 99.3% of top-10 accuracy on average against 4919 code phrases sampled from six open source projects. Time and keystroke savings of the new code completion method were evaluated in a user study. Time savings and keystroke savings were measured by comparing time usage and the number of keystrokes of the new code completion method with those of a conventional code completion method. The system achieved average 30.4% savings in time and 40.8% savings in keystrokes in a user study with eight participants. A key insight underlying this statistical-model-based approach to code completion is from our observation that source code in a programming language is statistically as predictable as, or even more predictable than, text in a human natural language; therefore a well-engineered statistical model of source code can infer idiomatic code phrases from a small but essential amount of ambiguous input—such as non-predefined abbreviation—just like statistical models of English words enable speech recognition from noisy voice signals and enables auto-complete and auto-correction for text entry on mobile devices.

The evaluation results suggest that this work may have a substantial impact on improving source code editing for effective ad-hoc code reuse.

5.2 Future Work

There are two important directions of future work on improving the bug detection method. The first is on increasing the precision of bug detection. Although the bug method has increased accuracy, an accuracy of 47% may not be high enough to encourage wide adoption of the bug detection method. Optimizing the parameters of suspicious code token pattern is planned as immediate future work because we have explored only a subset of many possible combinations of different parameter values. A complete exploration of parameter values may be able to find an optimal code token pattern that can achieve a higher precision. It has been difficult to automate the exploration process because each run required manual verification of bug warnings. It will be possible to automate the process and explore all necessary combinations once we have accumulated enough number of program bugs, related to code template reuse, in certain open source projects. Then the same set of program bugs can be used for evaluating bug detection accuracy without manual verification. Non-parametric changes in the suspicious code token

pattern, such as incorporating additional heuristics for determining code token equivalence based on synonyms or abbreviations, may be necessary to achieve a significant improvement of accuracy, too.

The second is on generating more informative bug reports. In particular, providing a type classification of programs bugs can be useful for prioritizing bug warnings for inspection. There are several common types of program bugs, such as inconsistent invocation of a super class method, inconsistent constant-to-variable assignment, or inconsistent identifier usage in a throw statement. Programmers may want to pay more attention to certain types of program bugs because they cause more serious problems than others.

One of the important goals of future work on Abbreviation Completion is to further improve the usability of the code phrase completion method. A user interface for validating code completion candidates deserves investigation because the user study revealed that the validation can be time-consuming and difficult. Improving efficiency is another goal because the system tended to be less responsive when the number of keywords was about 20,000. In addition, because the key benefit of our approach is keystroke savings, application to programming environments with limited input capabilities, such as mobile devices, may be worthy of investigation.

Finally, the solution approaches presented in this thesis demonstrate the power of a simple sequence-based model of source code. Analyzing vertical sequences of code tokens across similar code fragments is found useful for detecting program bugs accurately; learning to infer horizontal sequences of code tokens is found useful for efficient code completion. Therefore, another important direction of future research is to develop other sequence-based models of source code, as well as different analysis and inference techniques, which may address previously difficult software engineering problems.

References

[1] J. Sametinger, "Software engineering with reusable components," Springer, 2001.

[2] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in OOPL," Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on, IEEE, 2004, p. 83–92.

[3] A.J. Ko, H.H. Aung, and B.A. Myers, "Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks," Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on, IEEE, 2005, p. 126–135.

[4] B.S. Baker, "On finding duplication and near-duplication in large software systems," Reverse Engineering, Working Conference on, 1995, p. 86.

[5] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," ICSM, Published by the IEEE Computer Society, 1998, p. 368.

[6] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," IEEE Transactions on Software Engineering, 2002, p. 654–670.

[7] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A tool for finding copy-paste and related bugs in operating system code," Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation-Volume 6, USENIX Association, 2004, p. 20.

[8] Z. Li and Y. Zhou, "PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code," Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, ACM, 2005, p. 306–315.

[9] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ACM, 2007, p. 55–64.

[10] R.C. Miller and B.A. Myers, "Multiple Selections in Smart Text Editing," IUI, 2002.

[11] D. Hou, P. Jablonski, and F. Jacob, "CnP: Towards an environment for the proactive management of copy-and-paste programming," 2009 IEEE 17th International Conference on Program Comprehension, May. 2009, pp. 238-242.

[12] L. R. Rabiner, "Tutorial on hidden Markov models and selected applications in speech recognition," In Proc. IEEE, vol. 77, 1989.

[13] "Abbrevs," in GNU Emacs Manual. http://www.gnu.org/software/emacs/manual/emacs.html.

[14] "Editor Template," in Eclipse Ganymede Documentation. http://help.eclipse.org/ganymede/index.jsp.

[15] G. C. Murphy, M. Kersten, and L. Findlater, "How are Java Software Developers using the Eclipse IDE?," IEEE Software, vol.23, no.4, pp 78-83, 2006

[16] G. Little and R. C. Miller, "Keyword programming in Java," In Proc. ASE, vol. 16, pp. 37-71, 2007.

[17] N. Sahavechaphan and K. Claypool, "XSnippet: Mining for sample code," OOPSLA, pp 413-430, 2006.

[18] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman, "Jungloid mining: Helping to navigate the API jungle," In Proc. PLDI, 2005.

[19] R. Robbes and M. Lanza, "How Program History Can Improve Code Completion," In Proc. ASE, 2008.

[20] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," In Proc. FSE, 2006.

[21] F. K. Soong and E. F. Huang, "A tree-trellis based fast search for finding the n-best sentence hypotheses in continuous speech recognition," In Proc. ICASSP, vol. 1, pp 705-708, 1991.

[22] D. Nilsson and J. Goldberger, "An efficient algorithm for sequentially finding the n-best list," In Proc. IJCAI, 2001.

[23] S. M. Shieber and R. Nelken, "Abbreviated text input using language modeling," Natural Language Engineering, vol.13, 2007.

[24] T. Willis, H. Pain, S. Trewin and S. Clark, "Probabilistic flexible abbreviation expansion for users with motor disabilities," In: Proceedings of Accessible Design in the Digital World, 2005.

[25] S. Bickel, P. Haider, and T. Scheffer, "Predicting sentences using N-gram language models," In: Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language, 193-200, 2005.

[26] A. Nandi and H. V. Jagadish, "Effective phrase prediction," In: Proceedings of International Conference on Very Large Data Bases, 219—230, 2007.

[27] P. F. Brown, P. V. deSouza, R. L. Mercer, V. J. Della Pietra, and J. C. Lai, "Class-based n-gram models of natural language," Computational Linguistics, vol.18, 467—479, 1992

[28] "SciTE," in SciTE Documentation. http://www.scintilla.org/SciTE.html.

[29] C. E. Shannon, "Prediction and entropy of printed English," Bell System Technical Journal, 1951.

[30] S. Pini, S Han, and D. Wallace, "Text entry for mobile devices using ad-hoc abbreviation," In Proc. AVI, 2010.

[31] S. K. Card, T. P. Moran, and A. Newell, "The keystroke-level model for user performance time with interactive systems," Commun. ACM, 1980.

Acknowledgment

My deepest gratitude to my family and family-in-law: my mom and my dad, who always believed in me and what I could do, and supported me with unconditional love; my brother Jungho, who sent me pocket money to keep his older brother away from any financial trouble; my parents-in-law, who always gave me the warmest encouragement and have patiently waited almost two years without seeing their lovely daughter.

I am very grateful to Professor David Wallace for his advice on my research, comments on research papers and this dissertation, and encouragement and counseling when I feel lost or confused during my graduate studies. I also thank Professor David Wallace for giving me an opportunity to work with him in the classroom and learn many invaluable lessons from his innovative product engineering processes class.

I owe my sincere thanks to my thesis committee members: Professor Sanjay Sarma and Professor Rob Miller for their patience, encouragement, and insightful comments on my work. I am also indebted to Professor Rob Miller for having me join his research group meeting and kindly providing constructive feedback on my half-baked ideas and work-in-progress.

My heartfelt thanks to MIT CADlab members: Qing Cao and Mao Wei, who helped me settle down in early days in CADlab; James Penn, who has always been fun to be around and helped me in many ways all through the years; Sittha Sukkasi and Mika Tomczak, whose great sense of humor made me laugh and happy in the lab; Barry Kudrowitz and Monica Rush, who invited me to delightful thanksgiving dinners; Sungmin Kim, who not only sits at the closest desk to me, but has also been the closest friend in the lab; Stefano Pini, who made my first research collaboration productive and fun; Steven Keating, with whom I spent an intensive yet extremely rewarding semester as we work as TAs for 2.009 class; I am also grateful to other CADlab members for providing a friendly and creative environment to work or study.

My most special and deserved thanks to my dear wife Kyonghee Shin for her patience and love: her thoughtful care and encouraging words made me get through stressful and uncertain days during my dissertation. I am glad and grateful that we are at the end of the long journey. We made it!