Context-Sensitive Code Completion

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Muhammad Asaduzzaman

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon SK S7N 5C9

OR

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 - 110 Science Place
Saskatoon SK S7N 5C9

Abstract

Developers depend extensively on software frameworks and libraries to deliver the products on time. While these frameworks and libraries support software reuse, save development time, and reduce the possibility of introducing errors, they do not come without a cost. Developers need to learn and remember Application Programming Interfaces (APIs) for effectively using those frameworks and libraries. However, APIs are difficult to learn and use. This is mostly due to APIs being large in number, they may not be properly documented, and finally there exist complex relationships between various classes and methods that make APIs difficult to learn. To support developers using those APIs, this thesis focuses on the code completion feature of modern integrated development environments (IDEs). As a developer types code, a code completion system offers a list of completion proposals through a popup menu to navigate and select. This research aims to improve the current state of code completion systems in discovering APIs.

Towards this direction, a case study on tracking source code lines has been conducted to better understand capturing code context and to evaluate the benefits of using the simhash technique. Observations from the study have helped to develop a simple, context-sensitive method call completion technique, called CSCC. The technique is compared with a large number of existing code completion techniques. The notion of context proposed in CSCC can even outweigh graph-based statistical language models. Existing method call completion techniques leave the task of completing method parameters to developers. To address this issue, this thesis has investigated how developers complete method parameters. Based on the analysis, a method parameter completion technique, called PARC, has been developed. To date, the technique supports the largest number of expressions to complete method parameters. The technique has been implemented as an Eclipse plug-in that demonstrates the proof of the concept. To meet application-specific requirements, software frameworks need to be customized via extension points. It was observed that developers often pass a framework related object as an argument to an API call to customize default aspects of application frameworks. To enable such customizations, the object can be created by extending a framework class, implementing an interface, or changing the properties of the object via API calls. However, it is both a common and non-trivial task to find all the details related to the customizations. To address this issue, a technique has been developed, called FEMIR. The technique utilizes partial program analysis and graph mining technique to detect, group, and rank framework extension examples. The tool extends existing code completion infrastructure to inform developers about customization choices, enabling them to browse through extension points of a framework, and frequent usages of each point in terms of code examples. Findings from this research and proposed techniques have the potential to help developers to learn different aspects of APIs, thus ease software development, and improve the productivity of developers.

Acknowledgements

First of all, I would like to express my heartiest gratitude to my supervisors, Dr. Chanchal K. Roy and Dr. Kevin A. Schneider, for their constant guidance, valuable suggestions, encouragement and extraordinary patience during this thesis work. This thesis would have been impossible without them.

I am indebted to Dr. Daqing Hou (Electrical and Computer Engineering Department, Clarkson University, USA) for his continuous inspiration, expert advice and brilliant ideas that not only shape my research but also provide me the opportunity to be guided through the jungle of ideas.

I would like to thank Dr. Dwight Makaroff, Dr. Ian McQuillan, and Dr. Seok-Bum Ko for their participation in my thesis committee. Their advisements, evaluation, and insightful comments lead to the successful completion of this thesis. Thanks are also due to Dr. Tien N. Nguyen and all other members of my thesis examination committee for their helpful comments, insights, and suggestions.

I am thankful to the co-authors of the papers I have published during my graduate studies, including Chanchal K. Roy, Kevin A. Schneider, Daqing Hou, Ripon K. Saha, Massimiliano D. Penta, Michael C. Bullock, Ahmed S. Mashiyat, Samiul Monir, and Muhammad Ahasanuzzaman.

I would like to thank all the present and past members of the Software Research Lab for their support, advice, and inspiration. In particular, I would like to thank Manishankar Mondal, Jeffrey Svajlenko, Saidur Rahman, Ripon K. Saha, Minhaz Zibran, Mohammad Khan, Md. Sharif Uddin, Khalid Billah, Farouq Al. Omari, Masudur Rahman, Judith Islam, Kawser W. Nafi, Amit Mondal, Avigit K. Saha, Shamima Yeasmin, and Tonny Kar.

This work is supported in part by the First Research Excellence Fund (CFREF) under the Global Institute for Food Security (GIFS) and Natural Sciences and Engineering Research Council of Canada (NSERC). I am grateful to them for their generous financial support that helped me to concentrate more deeply on my thesis work.

I would like to thank all the staff members in the Department of Computer Science who have helped me throughout my graduate studies. In particular, I would like to thank Gwen Lancaster, Heather Webb, and Shakiba Jalal.

The most wonderful thing in my life is my son, Nubaid Ilham, whose presence inspires me in the right direction in finishing my thesis work. I am deeply indebted to my incomparable wife Nushrat Jahan for her love, patience and support during the thesis work. They are the ones who sacrificed the most for this thesis.

I express my heartiest gratitude to my family members and relatives especially my mother Ayesha Akhtara Begum, my father Md. Abdul Jalil Miah, my brother Muhammad Ahasanuzzaman, my uncle Motahar Hossain, Atahar Hossain, and Mozahar Hossain. Their constant encouragement and support have given me the confidence to complete this thesis.

This thesis is dedicated to my mother, Ayesha Akhtara Begum, whose affection and inspiration have given me mental strength in every step of my life, to my father, Md. Abdul Jalil Mia, whose guidance have helped me take the right decisions, and to my wife, Nushrat Jahan, whose continuous support and sacrifice have assisted me to complete this thesis.

Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
BCC	Better Code Completion
BMN	Best Matching Neighbors
CALCITE	Construction and Language Completion Integrated Throughout Eclipse
CSCC	Context Sensitive Code Completion
DVM	Dalvik Virtual Machine
ECC	Eclipse Code Completion
FCC	Frequency-based Code Completion
FEMIR	Framework Extension Miner and Recommender
GraLan	Graph-based Language Model for Source Code
GraPacc	Graph-based Pattern Oriented Context Sensitive Code Completion
HAPI	Hidden Markov Model for API Usages
HMM	Hidden Markov Model
IDE	Integrated Development Environment
JADEITE	Java API documentation with Extra Information Tacked-on for Emphasis
JVM	Java Virtual Machine
LHDiff	Language-independent Hybrid Diff
PBN	Pattern Based Bayesian Networks
PARC	Parameter Recommender
SLAMC	Statistical Language Model for Source Code

Chapter 1

Introduction

This chapter provides a short introduction to the thesis. After providing a motivation to this thesis in Section 1.1, research problems are discussed in Section 1.2. Section 1.3 discusses the process of discovering research problems and their solutions. The contributions of this thesis are described in Section 1.4. Finally, Section 1.5 provides an outline of the remaining chapters.

1.1 Motivation

Developers rely on frameworks and libraries of APIs (Application Programming Interface) to ease application development. While these APIs provide ready-made solutions to many complex problems, developers need to learn to use them effectively. The problem is that due to the large volume of APIs, it is practically impossible to learn and remember them completely. To avoid developers having to remember every detail, modern integrated development environments (IDE) provide a feature called Code Completion, which displays a sorted list of completion proposals in a popup menu for a developer to navigate and select. For example, when a developer types a dot (.) after a receiver name to complete a method call, Eclipse[1] offers a list of method names as completion proposals through a popup menu. When the developer selects a proposal by pressing the *enter* key, Eclipse automatically completes the method call. Omari et al. found that code completion operations are repetitive in nature [104]. In a study on the usage of the Eclipse IDE, Murphy et al. found the content assist as one of the top ten commands used by developers [90]. Surprisingly, the frequency of the execution of this command was similar to other common editing commands (such as delete, save, paste etc.). The content assist command offers the code completion feature in the Eclipse IDE. Thus, previous research indicates that code completion is crucial for today's development. As a result, it becomes an integral part of almost all modern development environments. We can thus leverage the code completion feature to help developers to learn and use those APIs. Existing code completion systems are far from perfect. This thesis advanced the state-of-the-art in code completion research in terms of context formulation and learning APIs.

[1] `https://eclipse.org/`

1.2 Research Problem

An investigation of the code completion literature has been conducted to identify both limitations of existing completion systems and opportunities to improve them. The following research problems have been identified:

- First, existing code completion systems already consider the structure of source code. To improve the performance of code completion systems, it is required to collect additional sources of information. Furthermore, a number of method call completion systems have been proposed in the literature that use different context information for recommending completion proposals. However, there is a lack of study on the performance implications of these context information.

- Second, existing method parameter completion systems are far from perfect. For example, consider the case of a parameter completion technique, called Precise [144]. The technique does not support completion of a number of parameter expression types. Precise excludes simple name, boolean variables and null literals from the recommendation. Precise also fails to detect the parameters of class instance creation type. However, a considerable number of parameter expression types fall into these categories. By integrating additional sources of information and by integrating different recommendation strategies, a system can possibly support completion of a large number of parameter expression types.

- Third, the code completion interface can be utilized to push notifications to developers to learn various aspects of APIs as they develop the code. For example, code completion interfaces can be utilized to help developers learn possible choices to customize the behavior of framework related objects. However, there is a considerable lack of study in this direction.

1.3 Addressing Research Problems

Since existing method call completion systems have already taken advantage of source code structure, there exists an opportunity of improving the accuracy of those systems by collecting additional information in the form of completion context. Careful observation of source code examples reveals that code elements that are related to each other are typically closely located within a few lines of distance. This localness property of the source can be utilized to capture the code completion context to improve the performance of existing method call completion systems. Furthermore, context-sensitive method completion requires matching of query context with the context of method calls collected from previous code examples [18]. Previous study on code clone detection found that simhash technique is effective to quickly determine similarity matching [138]. However, it is required to quickly validate the benefit of the localness property in capturing code context and the simhash technique in context matching. Thus, this thesis begins with a case study on tracking source code lines due to the simplicity in developing such a technique.

Results from the previous study shows that it is possible to capture the source code context of a line by

considering tokens within the top-4 lines. Furthermore, it is also possible to accelerate the line matching by using the simhash technique. Similar context and the simhash technique are used to develop a method completion technique, called CSCC (Context Sensitive Code Completion). CSCC is context-sensitive in that it uses new sources of information as the context of a target method call. CSCC uses new sources of contextual information together with lightweight source code analysis to better recommend API method calls. Evaluation against state-of-the-art code completion techniques suggests that CSCC has high potential. It is also investigated how the different contextual elements of the target call benefit CSCC.

Existing method call completion techniques only focus on completing method calls but do not focus on completing method parameters. However, a number of previous studies suggested that completing method parameters is also a non-trivial task [106, 107]. Thus, the third study of this thesis focused on the problem of completing method parameters. Investigation of existing code completion systems revealed that only a subset of parameters expression types is supported by existing studies. An experiment was conducted to better understand how developers complete method parameters, to learn the parameter expression types and to discover any patterns of method parameter completion. Results from the study indicated that method parameter completion is also locally specific. Based on the observation from that study, a recommendation technique, called PARC, is developed. The technique collects parameter usage context using the source code localness property that suggests that developers tend to collocate related code fragments. PARC uses previous code examples together with contextual and static type analysis to recommend method parameters. Evaluation of the technique against the only available state-of-the-art tool using a number of subject systems and different Java libraries shows that PARC has high potential.

It was observed during the development of PARC that developers often pass a framework-related object as an argument to an API method call to customize framework behavior. The formal parameter of the method is called an extension point. Such an object can be created by subclassing an existing framework class or interface, or by directly customizing an existing framework object. However, to do this effectively requires developers to have extensive knowledge of the framework's extension points and their interactions. To help developers in this regard, this thesis develops a technique that utilizes partial program analysis and a graph mining technique to detect, group, and rank framework extension examples. The tool extends existing code completion infrastructure to inform developers about customization choices, enabling them to browse through extension points of a framework, and frequent usages of each point in terms of code examples.

1.4 Contributions of the Thesis

The contributions of this thesis are as follows:

- First, a language-independent hybrid line location tracking technique [11, 12]. The technique is developed as part of the case study to capture source code localness property and to determine the effectiveness of the simhash technique. Both have been used in developing code completion systems.

- Second, a context-sensitive method call completion technique, called CSCC [6, 7, 8]. The technique has been evaluated against state-of-art method completion techniques.

- Third, a method parameter completion technique, called PARC, that supports the largest number of parameter expression types [4, 5].

- Fourth, a technique that uses the code completion menu to help developers to learn framework extension examples [9, 10].

1.5 Outline of the Thesis

This chapter (Chapter 1, Introduction) introduces research problems and presents a short description of our research behind addressing those problems. The remaining chapters of this thesis are organized as follows.

- Chapter 2 describes the background topics.

- Chapter 3 provides detail description of the first study (i.e., Study 1) on tracking source code lines to understand capturing code context and to realize the benefits of using the simhash technique. The technique has been published in the Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 2013), a premier international software engineering conference [11]. An implementation of the technique has been published in the tool demo track of the same conference [12].

- Chapter 4 provides a detailed description of the second study (i.e., Study 2) on improving code completion support for API method calls. This work has been published in the Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSME 2014, formerly known as ICSM) [7]. The technique has been implemented as an Eclipse plugin and has been published in the tool demo track of the same conference [6]. An extended version of the paper has also been published in the Journal of Software: Evolution and Process (Volume 28, Issue 7, July 2016) [8].

- Chapter 5 discusses the third study (i.e., Study 3) on how developers use API method parameters including our proposed technique and evaluation results. A paper describing the technique has been published on the 31st IEEE International Conference on Software Maintenance and Evolution (ICSME 2015) [4]. The technique has been implemented as an Eclipse plugin, called PARC. A description of the tool has also been published in the tool demo track of the same conference [5].

- Chapter 6 presents the fourth study (i.e., Study 4) on recommending framework extension examples. A paper describing the technique has been published in the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME 2017) [10]. An implementation of the technique has also been published in the tool demo track of the 32nd IEEE/ACM international conference on Automated Software Engineering (ASE 2017), a leading conference in the area of Software Engineering [9].

- Chapter 7 concludes the thesis by summarizing research contributions and outlining future research directions.

Chapter 2

Background and Taxonomy of Related Work

This chapter introduces the code completion feature, an integral part of modern IDEs. It explains the benefits, requirements and components of code completion systems. The existing code completion systems are divided into a number of categories and the chapter briefly describes each of them. All these build the foundation for this thesis.

2.1 Benefits of Code Completion

To identify the benefits of code completion, a number of research papers, blogs, and tutorials are reviewed. The following section highlights major benefits of code completion.

1. **Avoid remembering every detail:** Due to a large number of APIs, it is difficult for developers to remember every detail, even experienced developers suffer from this problem. Consider the situation where a developer wants to compare two strings in Java ignoring case conditions. The *equalsIgnoreCase* method of the *String* class is designed to accomplish the task. It might be the case that the developer forgets the name of the method to use. However, the method is very popular among various other methods in the *String* class. When she types the dot (.) followed by a *String* variable, the code completion system that sorts method names based on popularity can suggest the target method name in the top few positions and the developer can easily identify the target method.

2. **Help developers to write error-free code:** Code completion systems offer only those completion proposals that are syntactically correct in the current development context. For example, when a developer requests for code completion on a *JFrame* object of the *javax.swing* framework, only those methods that are accessible from the object are offered as completion proposals. This ensures that developers do not write uncompilable code.

3. **Speed up typing:** To help developers to understand the code, descriptive names need to be used for classes, methods or field variables. Typing these names can be cumbersome. However, code completion systems can automatically complete these names based on the selection of the developer and speed up the typing. Many code completion systems go beyond the line level and help developers to automatically complete multiple lines of code [44].

4. **Offers convenient documentation:** When a code completion system offers a list of completion proposals, developers can browse through the proposals and the completion system offers additional information about the currently selected proposal to developers without switching context. This enables developers to learn about completion proposals and the code completion serves the purpose of the documentation. For example, consider the situation where a developer is working on a *MimeMessage* object of the *JavaMail* API. She does not know which method to call and thus use the code completion feature to access the list of methods that can be called on the *MimeMessage* object. As the developers traverse through the list, she learns about various methods of the *MimeMessage* class. The code completion system also shows the documentation of each method in a different window on the fly.

5. **Code reuse:** Developers often reuse code through copy-paste programming practices to avoid implementing everything from scratch. This leads to duplicate code fragments, also known as code clones. Results from code clone research suggest that software systems contain a significant amount of clones and the quantity ranges from 15-25% [120] and can be even up to 50% [112]. However, reusing code through this process requires developers to actively locate code fragments that can be reused. Code completion provides a mechanism to automatically suggest reusable code fragments and thus supports code reuse. For example, Hill and Rideout developed a technique that can provide suggestions to complete a method body based on the initial incomplete implementation of that method [44].

2.2 Requirements of Code Completion Systems

After careful review of the existing literature and existing code completion tools, the following requirements of a code completion system have been identified.

1. **Short response time:** One of the important requirements of code completion systems is the ability to produce completion proposals in real-time. This ensures that developers do not need to wait a considerable time to get suggestions from the completion systems. Although the actual response time differs across different completion systems, most existing research targets sub-second response time. The default activation delay for content assist in the Eclipse IDE is 100 ms. This can be considered an ideal response time because the delay would be less likely noticeable in this situation.

2. **Cue:** Code completion systems should provide additional information, also called cues, to developers where possible to help to decide the applicability of a completion proposal in the current editing location. Cues can be provided in textual or in visual formats. For example, Eclipse offers the method signature and the documentation, as a developer traverses through the list of method completion proposals. This additional information helps developers to pick the correct proposal even when the completion system fails to put the correct proposal in the top few recommendations. All this information should be accessible to users without introducing any context switching.

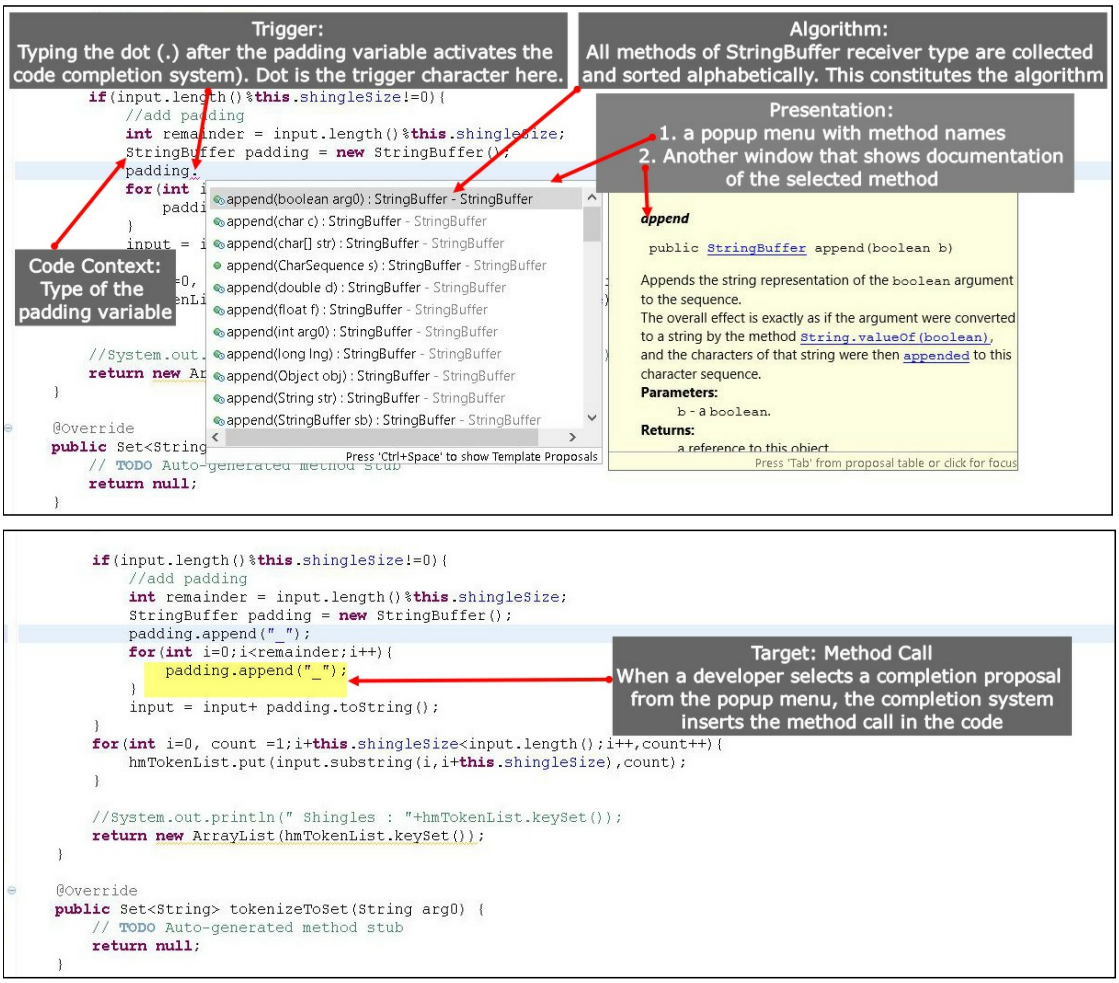


Figure 2.1: Five different components (trigger, algorithm, code context, presentation, and target) of a code completion system

3. **High success rate:** If a code completion system does not recommend useful suggestions in the majority of cases, developers would not be interested in using the tool. Instead, they would consider the tool as a burden to development. Thus, common sense dictates that high success rate is a prerequisite for the survival of a code completion system.

4. **Non-interruptive:** This requirement ensures that the completion systems do not interrupt the natural flow of developers.

5. **Auto-activation:** Code completion systems should be able to automatically capture the code context and recommend completion proposals. This auto-activation feature ensures that the developers do not need to spend time or effort to obtain suggestions. The dot character is typically part of the source code and that is why it is frequently used as the trigger character to activate code completion systems.

2.3 Components of a Code Completion System

To understand what constitutes a code completion system, existing literature is carefully reviewed. Code completion features of various integrated development environments are also investigated. This includes but is not limited to Eclipse,[1] Microsoft Visual Studio,[2] Atom,[3] NetBeans,[4] and PyCharm.[5] To facilitate the discussion, the code completion feature of the Eclipse IDE is referred to in this discussion. Based on the analysis, five different components of a code completion system are identified. Figure 2.1 depicts all five components and the following section briefly describes each of them.

1. **Trigger:** A trigger represents an action that activates a code completion system. For example, when a developer types a dot (.) after a receiver object in the Eclipse IDE, the completion system activates and presents a list of completion proposals (such as method names or field variables). Thus, pressing the dot (.) after a receiver variable acts as a trigger in this case. This is also an example of an implicit trigger because the dot(.) is the part of the code and users do not need to tell the IDE explicitly to activate the code completion system. An example of the explicit trigger is pressing the Ctrl+Space keys to query the code completion system for completion proposals.

2. **Algorithm:** This represents the procedure for computing completion proposals. The Eclipse IDE contains two default code completion systems, ECCAlpha and ECCRelevance. For recommending method calls, both techniques leverage the static type system to collect all methods that can be called on the receiver object. ECCAlpha sorts the completion proposals in alphabetical order and presents them to developers. On the contrary, ECCRelevance uses a positive integer value, called relevance, to sort them. The value is calculated based on the expected type of the expression as well as the types in the code context (such as return types, cast types, variable types etc.). In both cases, the process of computing completion proposals represents the algorithm. Code completion systems often use a data mining or machine learning technique and that is the core of computing completion proposals. In that case, that technique is referred to as the algorithm also, although there can be some additional steps associated to the process. For example, a code completion system can use the association rule mining technique for computing completion proposals and the association rule mining technique is referred to as the algorithm.

3. **Completion Context:** Completion contexts refer to the information required to compute completion proposals, also known as the input. For example, the completion context of the ECCAlpha is the receiver type. However, ECCRelevance uses both the receiver type and relevance value computed from

[1] https://eclipse.org
[2] https://www.visualstudio.com
[3] https://atom.io
[4] https://netbeans.org
[5] https://www.jetbrains.com/pycharm

the expression as completion context. BMN (the Best Matching Neighbors) is a code completion system that collects the receiver type and the list of methods that are called on the receiver object prior the editing location as the completion context [18].

4. **Presentation:** This refers to how completion proposals are presented to developers. The predominant approach of presenting code completion proposals as a list of items through a popup menu. When a developer selects a menu item, further information can be provided through additional popup menus or windows. For example, when a developer selects a method name from a completion popup menu, Eclipse shows the API documentation of the selected method. It is also possible to present additional information that can be linked through the code completion feature.

5. **Target**: This indicates what is recommended by a code completion system. Depending on the objective of code completion systems, it can be method calls [109], method parameters [144] or can be even method bodies [53].

2.4 Techniques

This section summarizes various code completion systems targeting different API elements. It also describes completion systems that work at the lexical levels but can be designed to complete API elements. Their suggestions may contain API elements or they help to understand different aspects of APIs (such as API changes). Towards this goal, existing code completion techniques are divided into the following categories.

2.4.1 Method call completion

Code completion systems in this category focus on completing method calls. A number of code completion systems have been proposed in the literature. This is mostly due to the fact that method calls are more

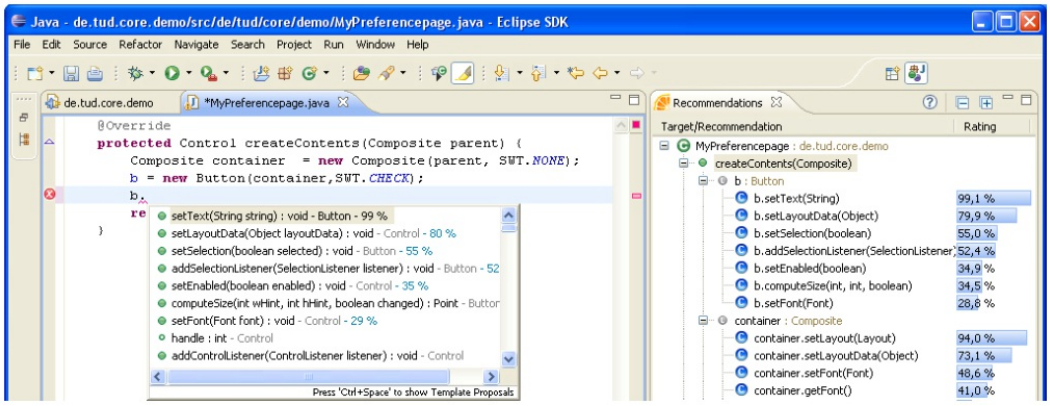


Figure 2.2: An example of integrating the BMN code completion system in the Eclipse IDE. The likelihood of calling each method is also reported to users (collected from Bruch *et al.* [18]).

frequent in source code. Although the target is the same, the context and the algorithm used for recommending completion proposals varied from one technique to another. Bruch *et al.* [18] propose three different code completion techniques to suggest API method calls. The frequency-based code completion technique determines the frequency of method calls in the example code. The larger the frequency of a method call, the higher the position of the method call in the recommended list of completion proposals. The association rule based code completion system applies association rule mining to suggest method calls. Finally, the BMN code completion system uses a k-nearest-neighbor machine learning algorithm to recommend API method calls for a particular receiver object (see Figure 2.2). The technique determines the context of each local variable of the framework type. It determines the declared type of the variable, the names of methods that are called by the variable, and the enclosing method name that contains the local variable.

The above information is encoded in a binary feature vector. BMN generates a set of feature vectors represented as a two-dimensional usage matrix using all the variables extracted from the code base, which acts as the training data. Given a variable v of a framework type, the technique represents the current programming context as a feature vector. It then determines the Hamming distance between the feature vector of the variable v and those variables in the code base that give the lowest distance. The next step is to synthesize recommendations where the technique suggests method calls that are missing on v but appears on the previously selected variables in the code base by considering their frequency. Results from the empirical study reveal that the BMN system outperforms the other two techniques. The BMN system does not consider the order of method calls. However, implicit ordering of API method calls exists while implementing an API usage pattern. Considering the order of method calls and better context information can possibly improve the performance of the technique.

Proksch *et al.* [109] develop a new algorithm, called Pattern Based Bayesian Networks (PBN), that improves the BMN algorithm by using a Bayesian network and using additional context information. BMN considers the type of the receiver variable, receiver call sites and the enclosing method definition. In addition to the above contexts, PBN considers three additional sources of information. These are parameter call sites, class context and definition types. Evaluation results reveal that the technique improves the quality of the prediction by 3% with an increase of the model sizes. PBN can also use a clustering technique to reduce the model size significantly with a slight decrease in the performance.

Hou and Pletcher [47, 48] propose a code completion technique that uses a combination of sorting, filtering and grouping of APIs. They implement the technique in a research prototype called Better Code Completion (BCC) [105]. BCC can sort completion proposals based on the type-hierarchy or frequency count of method calls (see Figure 2.3). The technique supports user-specified filtering of methods calls. This filtering is based on the observation that not all public methods are APIs and certain methods are only designed to be called in a particular context and for particular receiver objects. BCC also enables developers to group related APIs because the alphabetical ordering separates logically-related API methods. However, BCC does not leverage previous code examples. Moreover, BCC requires the filters to be manually specified which can only

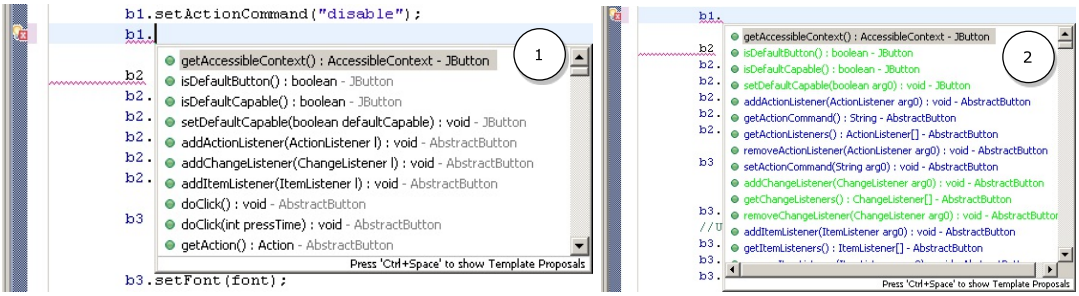


Figure 2.3: Examples of recommending method calls by BCC. The technique can sort completion proposals by popularity (1). The alternative approach (2) is to sort completion proposals based on the position of declaration in the type hierarchy followed by alphabetically within each type (collected from Hou and Pletcher [47]).

be performed by expert users of code libraries and frameworks.

Robbes and Lanza [113] propose eight different algorithms that consider varying sources of information for recommending method calls. These algorithms are evaluated using a data set consisting of the change history of several software systems. To collect the data set, the evolution of a software system is considered as a sequence of change operations rather than a set of versions. Each change operation can be executed to move a software system from one state into another. For object-oriented programming languages, the change operations alter the abstract syntax tree of the program. Examples of change operations can be, creating a node, adding a node as the child of a parent node, changing the property of a node, and removing a node. They find that the type optimist completion algorithm that merges the optimist algorithm with the type information provides the best result.

In a different study, Robbes and Lanza extend the previous work by supporting completion of class names [115]. Here, a code completion system is evaluated each time a class reference is inserted in the code. The program change history can be considered as the temporal context for a method call. While the previous techniques (such as the BCC) do not consider the prefix of a method call as an input, they consider at least two prefix characters for evaluating their algorithms. Their techniques also require a change-based software repository for suggesting completion proposals. The problem is that such repositories are not popular yet and are difficult to obtain.

Heinemann and Hummel [42] propose a technique that considers the identifiers as method call context to recommend API method calls. For each API method call, the technique collects identifiers that appear in the fixed number of non-comment and non-empty lines prior to calling the target method. Identifiers consisting of multiple words are split based on the convention of camel case notation in Java and stemming is performed to reduce inflectional words. Inflection is the name for the extra letter or letters added to nouns, verbs and adjectives in their different grammatical forms. The set of identifiers constitutes the context of the method call and encoded as a binary feature vector. The feature vectors of all API method calls form a two-dimensional matrix, which is similar to the BMN code completion. To recommend a method call, the technique generates a feature vector from the current code context. The nearest neighbors of the current feature vector is determined by comparing the Hamming distance with those feature vectors in the usage

matrix. The corresponding method calls are recommended after sorting them in ascending order of the distance.

In a separate study, they evaluate the impact of several modifications to the original algorithm [41]. For example, they consider a fixed number of identifiers prior to calling an API method as the method call context. The similarity between a query and the examples are determined by the Jaccard similarity of the terms that appear in their context. They also compare the technique with a structure-based approach, called Rascal [82]. Rascal uses a combination of collaborative and content filtering algorithm to recommend method calls. The content-based filtering component only considers the last method call to refine the suggestions made by the collaborative filtering. This limited context information is likely to be contributed to the poor performance of the technique. Both techniques do not consider the type of the receiver variable as the code context, which makes the recommendation more difficult.

APIRec is a method call completion technique that takes into account the fine-grained changes in the source code [91]. The technique is based on the observation that source code changes are repetitive in nature. For example, consider a situation where a developer adds a method call m_1, she also adds another method call m_2. Thus, a change in the source code can trigger another change (such as including a method call). The technique mines the previous versions of source code to determine fine-grained co-occurring changes. When a developer requests a method call completion, the technique determines the fine-grained changes that took place in the same development session prior to requesting the method call. It also determines the tokens that appear within a fixed distance prior to the current editing location. The former is called the change context and the later is known as the code context. These two forms of context are matched with the context of method calls in previous code examples. A combined similarity score is used to sort and recommend method calls.

Although the previous techniques for code completion learn API usages from source code, it may be difficult due to insufficient code examples. The problem has been addressed in a work that develops a statistical approach for learning APIs from DVM bytecode, called SALAD [100]. The technique has been implemented in a tool, called DriodAssist, that supports completion of API method calls [101] (see Figure 2.4). The tool learns API usages from the bytecode of Android mobile applications. It uses the Groum Extractor component to collect graph-based API usage models from the bytecode. The core of the tool is a statistical generative model, called HAPI (Hidden Markov Model for API Usages). HAPI uses a Hidden Markov Model to learn API usage patterns from sequences of method calls. These method call sequences are extracted from Groums first. DriodAssist also checks suspicious calls in a method call sequence and recommends repair to the call marked as suspicious. However, the authors did not compare the technique with various other popular API call completion systems, such as those that are not based on the statistical language models.

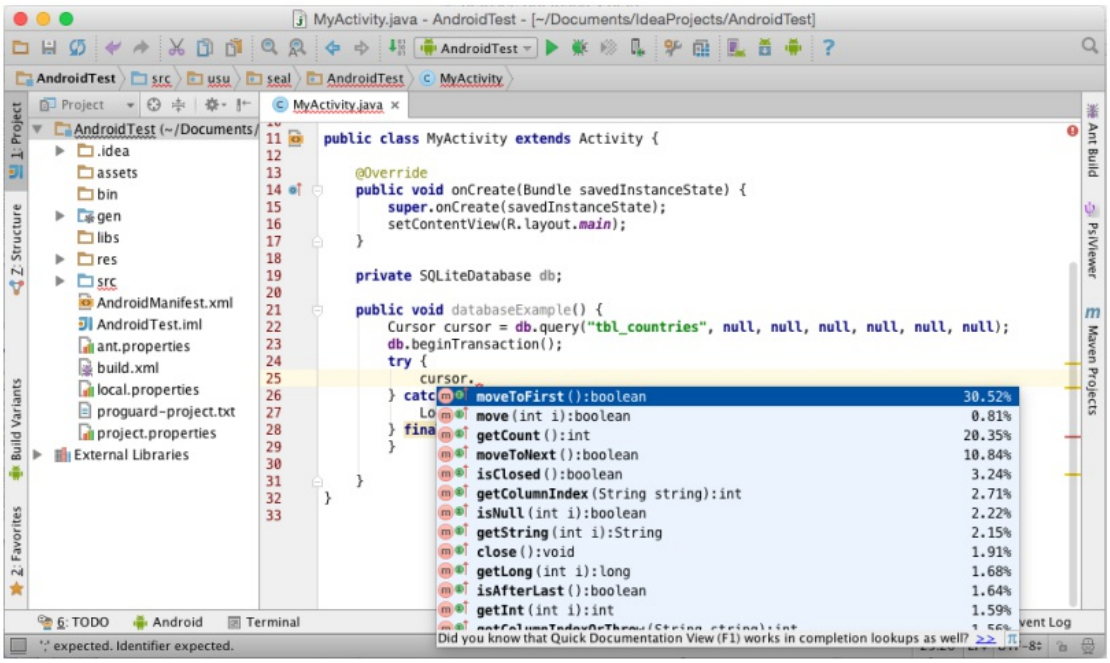


Figure 2.4: An example of recommending method calls by DroidAssist (collected from Nguyen *et al.* [101]).

2.4.2 Method parameter completion

While a number of code completion techniques have been developed to support automatic completion of API method calls [18, 47, 48, 91, 113], they leave the task of completing method parameters on the developers. Here, the term parameter is used to refer to the actual parameter (or the argument) and not the formal parameter. Determining the correct parameter to call a method is a non-trivial task and requires much attention [106, 107].

The only work that targets automatic completion of API method parameters is the study of Zhang *et al.* [144]. They propose a technique, called Precise, that mines existing code bases to generate a parameter

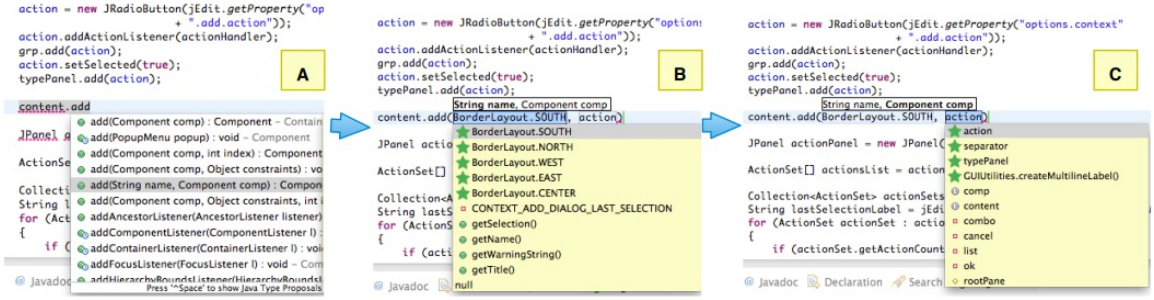


Figure 2.5: An example of recommending method parameters. The code completion system recommends method calls (A). The developer selects a method call and the system automatically recommends the possible completions of the first parameter (B). The developer can cycle through the completion proposals of other parameters also (C).

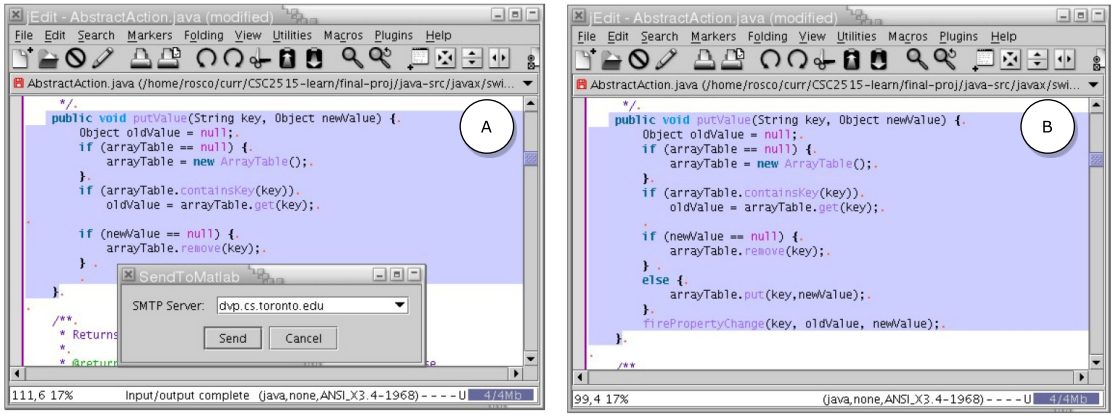


Figure 2.6: An example of completing a method body (collected from Hill and Rideout [44]). The technique sends the partially completed method body as the query to the server (A). The server returns a method that best matches with the query and the technique updates the method body (B).

usage database. A parameter usage instance consists of four pieces of information: (i) the signature of the formal parameter bound to the actual parameter, (ii) the signature of the enclosing method in which the parameter is used, (iii) the list of methods that are called on the variable used in the actual parameter, (iv) methods that are invoked on the base variable of the method invocation using the actual parameter. Given a method invocation and a parameter position, Precise identifies the parameter usage context in the current position and then looks for a match in the parameter usage database using the kNN algorithm. Precise has been evaluated using SWT library method parameters and Eclipse.

Although the technique is interesting, it has several limitations. First, Precise cannot recommend parameters of the following expression types: simple name, boolean, null literal, and class instance creation. Unfortunately, a large number of parameters can have those expression types. Second, JDT[6] (Java Development Tool) can suggest parameter expression types of simple name, boolean and null literal but it does not support parameters of class instance creation type. Although the recommendations suggested by JDT can be combined with those of Precise, it is not clear how these two groups of recommendations can be combined and whether there is a better approach exists. Precise has been evaluated for the source code written in the Java language. Thus, the performance of the technique is not clear across different programming languages (such as the C# language). Visual C# 2010 introduces named and optional arguments.[7] It would be interesting to support auto completion of both named and optional arguments.

2.4.3 Method body completion

A number of techniques have been developed that target automatic completion of method bodies [44, 53, 142]. Although these techniques do not focus on completing API calls, parameters, or expressions involving

[6] http://www.eclipse.org/jdt/

[7] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/named-and-optional-arguments

API elements, they can support extending a framework. For example, such techniques can be used to provide the implementation of API methods that need to be overridden while extending a framework class or implementing a framework interface. Hill and Rideout [44] develop a technique that can automatically complete method bodies using atomic clones that exist in code bases (see Figure 2.6 for an example). Atomic clones are small sized near duplicate code fragments (5–10 lines) that are frequently created by repeatedly providing the same unit of implementation with little or no changes (such as implementing a listener interface in Java). The technique generates a 154-dimensional feature vector for each method in a system. When a developer starts typing a method body, the technique uses the partial method implementation to generate a feature vector. It then applies the KNN algorithm using the Euclidean distance to determine nearest neighbors method bodies. The feature vector representation is limited in the sense that it cannot be used to represent arbitrary blocks of code.

Yamamoto *et al.* [142] also develop a technique to support method completion but the approach is different than the previous technique. Given a collection of source code files, the technique first extracts all the methods, converts them into token sequences and stores them as key-value pairs. For each method, the technique generates keys of variable sizes. Each key consists of a set of tokens that begins the method body and the value corresponds to that key consists of the set of the remaining tokens of the method body. Given a partially written code fragment, the technique converts the code into tokens and uses that as a key to look for values that are potential candidates to complete the remaining parts of the code. Although the technique supports completion of arbitrary code blocks, the technique is limited in that it generates a large number of key-value pairs. If the technique extracts n tokens from a method, it generates *(n-1)* key-value pairs, while in practice only a small set of them can be reused by the developer.

Clones are similar code fragments often created through copy-paste programming practices. Code clone detection techniques, particularly those that focus on detecting clones from a large collection of source code repositories are related to method body completion. This is because they use indexing and similarity detection techniques for clone detection. Given a complete method body (also known as the query) a clone detection technique can be asked to detect clones quickly. The detected clones or method bodies can be identical to the query (also known as exact or type-1 clones) or can contain minor to significant editing changes (also known as near-miss closes). If the differences between near-miss clones are only in identifier names and literal values, they are called type-2 clones. Otherwise, they are called type-3 clones and they differ at the statement level (such as statements can be added, deleted, or modified).

A number of clone detection techniques have been proposed in the literature [15, 26, 31, 57, 58, 63, 68, 69, 70, 76, 81, 118, 120]. However, many of these clone detection techniques are not able to produce real-time cloning because they are either not incremental or not scalable to hundreds of millions of lines of code [50]. Göde *et al.* propose an incremental clone detection technique based on the suffix tree [38]. SHINOBI is a clone detection tool that uses suffix array index to detect clones quickly [59]. ClemanX is another incremental clone detection tool that uses characteristics vector computed from abstract syntax tree of code fragments

for detecting clones [97, 98]. Locality sensitive hashing techniques have also been found effective for quick detection of code clones. DECKARD uses characteristic vectors to approximate structural information of code fragments [55]. It then uses locality sensitive hashing (LSH) to cluster similar vectors or code clones. LSH hashes similar code fragments in the same bucket. The random projective method of LSH is called simhash [24]. The technique has been found effective for detecting near-duplicate web pages [80]. Uddin *et al.* evaluate the effectiveness of using simhash for near-miss clone detection [138]. Each simhash value is represented as a fixed length binary value. The similarity between a pair of simhash values is calculated using the Hamming distance. The Hamming distance between two equal length binary values is the number of positions at which the corresponding binary numbers are different. The higher the similarity between two code fragments, the smaller would be the Hamming distance between their simhash values. Their proposed technique uses two-level indexing to speed up the clone detection process. The first level indexing is based on the line length of code fragments and the second level indexing is based on the number of 1-bits in the simhash value. SourcererCC is a clone detection tool that uses an inverted index structure to quickly locate potential clones of code fragments [124, 125]. This is followed by a filtering step that considers the order of tokens in code fragments to remove false matches. Svajlenko *et al.* develop a technique that detects clones using Jaccard similarity metric [132]. To avoid comparing code fragments that are likely to be not clones of each other, the technique uses the sub-block filtering approach. To eliminate the burden of excessive memory requirement, the technique leverages partitioned partial indexes where code fragments are split into a number of partitions and a partial index is constructed for each partition. During clone detection time, only one partition is kept in memory at a time and clone fragments in that partition are used to query the index to detect potential clones. None of these works focus on completing method bodies.

Real-time search of code clones is also related to automatic completion of method bodies. Keivanloo *et al.* identify a number of important requirements of real-time code clone search [61]. These are scalability, short response time, scalable incremental corpus update, and ability to search all three clone types (type-1, type-2, and type-3). Results from the study reveal that it is possible to perform real-time code clone search through multi-level indexing. Zibran *et al.* develop an IDE integrated code search tool that leverages k-difference suffix tree to support real-time code clone search [146]. Both exact and near-miss clones are supported by the technique. Lee *et al.* develop an instant code clone search tool leveraging a multidimensional indexing structure (i.e., R* tree) that supports automatic detection of top-k clones given a code fragment [73]. To improve the search performance, the technique exploits a dimensionality reduction approach. Method completion techniques can benefit from the real-time clone detection approaches. However, these approaches have not been evaluated for automatic completion of method bodies. Furthermore, while the query in clone detection is either a whole method body or a code block, the query in method completion is only a small fraction of it. This difference prevents the use of code completion techniques for automatic completion of method bodies without any modifications and calls for further research in this direction.

Ishihara *et al.* [53] propose a technique that can recommend past reused code examples that are detected

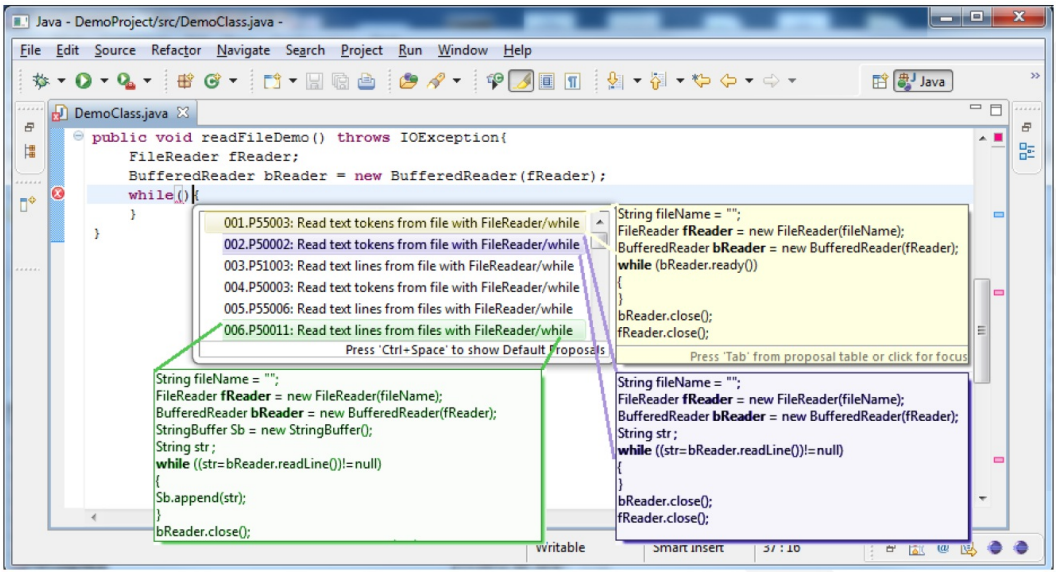


Figure 2.7: GraPacc recommends completion proposals based on API usage patterns (collected from Nguyen *et al.* [92]).

using clone detection techniques. The technique consists of a code analysis procedure and a code suggestion procedure. The code analysis procedure detects clones using a hash-based clone detection technique, determines a score for each code fragment using the component rank technique [51, 52], and extracts keywords in it. The score is defined as the score of the method containing the clone fragment. The component rank technique considers a variation of the page rank algorithm to compute the score. The basic idea behind the score is that methods that are called by many methods are significant and methods that are called by significant methods are also significant. Given a query string, the code suggestion procedure sorts the code fragments related to the query using a combination of component ranking score and word occurrence score. It then suggests code fragments based on the descending order of the score. An incomplete method body can be considered as a query string and code fragments related to it can be suggested to complete the body using the combination of two different scores.

2.4.4 API usage pattern completion

Mining API usage patterns is a great way of learning different usages of APIs and their code examples, and a number of techniques have been proposed in the literature. For example, Acharya *et al.* [1] mine API usage patterns as partial orders, by leveraging static traces of source code. MAPO is an example of API usage pattern mining framework [145]. It applies a clustering algorithm to group related API calls, generates method call sequences for each cluster and then applies a sequential pattern mining technique to discover frequent patterns from those sequences. GrouMiner is a graph-based approach that can mine frequent usage patterns involving multiple objects from source code [99]. Wang *et al.* [140] proposed a technique that uses a combination of closed frequent pattern mining approach and two-step clustering to find succinct and high

coverage API usage patterns. API usage patterns can help code completion systems to better formulate the code context.

GraPacc is a graph-based pattern oriented context sensitive code completion tool [92, 94]. Given a code base, the technique generates API usage patterns along with their frequencies by applying *GrouMiner*, a graph-based pattern mining tool that can determine frequent API usage patterns involving multiple objects. GraPacc uses a graph-based model, called *Groum*, to represent API usage patterns. The nodes in a *Groum* represents method calls, objects/variables and control structures (such as if, for, while). Edges that connect nodes represent the data dependencies. Given a partially completed code fragment involving a target API, called the query, the technique generates graph-based and token based features. It also determines the above two categories of features for mined API usage patterns. The similarity of features between the query and the mined API usage patterns is used to sort the pattern that matches the query. If a developer selects a suggested pattern, the technique completes the query in the code according to the selected pattern (see Figure 2.7). One limitation of the technique is that only the pattern numbers are displayed in the code completion popup menu, which makes it difficult to understand the purpose of the pattern before making the selection. While it is possible to label each pattern manually, automatically finding labels for those mined patterns would be more useful.

2.4.5 Language model-based code completion

Hindle *et al.* [45] find that the source code most developers write is natural in that the code elements are simple, repetitive, and has predictable statistical properties. The repetitiveness of the source code can be captured by statistical language models and the repetitiveness or regularity of the source code is specific to each software project. They also observe that such regularity is common in applications within the same category but less common in applications from different categories. To capture the repetitiveness in the source code, they develop a code completion system using a trigram language that suggests the next token. They compare the technique with the code completion system (ECC) of the Eclipse IDE. Their code completion system performs better than ECC when the target token has 6 or fewer characters. So they merge the two algorithms to develop a new code completion system that recommends suggestions from ECC if it recommends a long suggestion. Otherwise, the new technique combines half of the suggestions of the trigram model with half of the suggestions from ECC.

While the previous works focus on discovering repetitiveness of the source code, they fail to capture the localness property of the source code. Tu *et al.* [137] find that the source code is locally specific and proximally repetitive. The n-gram language models can only capture the global repetitiveness but fail to capture local repetitiveness of source code. For example, some n-grams can only frequently appear in a file rather than in other files in the source code. They develop a cache language model that gives more wights on the locally specific and locally repetitive tokens. CACHECA is an Eclipse plugin that combines the suggestions of the Eclipse code completion system with that of the cache language model [36]. Each technique

produces a list of suggestions separately and CACHECA mixes the suggestions as follows. Suggestions that appear within the top-3 suggestions in both lists are considered first. Next, if either list contains fewer than three suggestions, then the top suggestion from that list is considered. Finally, CACHECA interleaves the remaining suggestions. However, CACHECA does not consider the type information. Including additional context (such as information regarding target token type and type information from the context) can possibly help improving the performance of CACHECA.

SLAMC is an example of the statistical language model for source code [96]. The technique recommends sequences of tokens by bringing semantic information into code tokens through annotating each token with the token role and data type. One of the important contributions of the SLAMC is incorporating the semantic information into code tokens that permit the language model to recommend suggestions even when the lexical information of the tokens vary. Another important contribution is considering the co-occurrences of code tokens because some tokens trigger the appearance of other tokens. For example, the *try* and *catch* keywords appear together in Java and the appearance of the *try* keyword triggers the appearance of the *catch* keyword. However, the technique is not designed for dynamically typed programming languages. It is yet unclear how semantic information can be incorporated for those languages and to what extent semantic information can benefit those languages.

SLANG is a code completion technique that collects sequences of method calls to create a statistical language model [110]. When a developer requests for a code completion, the tool completes the editing location using the highest ranked sequence of method calls computed by the language model. The tool supports completion of multiple statements, handles multiple objects and infers sequences not in the training data. GraLan is a graph-based statistical language model that targets completing API elements [93]. The term *API element* refers to method calls, field accesses and control units (such as for, while, if etc.) used in the API usage examples. The technique mines the source code to generate API usage graphs, called *Groums*. Given an editing location, the technique determines the API usage subgraphs surrounding the current location and use them as context. GraLan then computes the probability of extending each context graph with an additional node. These additional nodes are collected, ranked and recommended for code completion.

2.4.6 Keyword-based code completion

The idea of keyword-based code completion comes from the end user programming research where the goal is to help users to write scripting commands. Little and Miller develop a prototype system that transforms user specified keywords expressing a command to executable code [77] (see Figure 2.8(A)). Their work is based on the assumption that if a user is familiar with the application domain, she can write keywords expressing the command, which they supported using empirical evaluation. They extend the idea to keyword programming where instead of remembering the programming syntax, a developer needs to type a small set of keywords and the system translates those words into a valid expression [78]. Evaluation of the system against developer

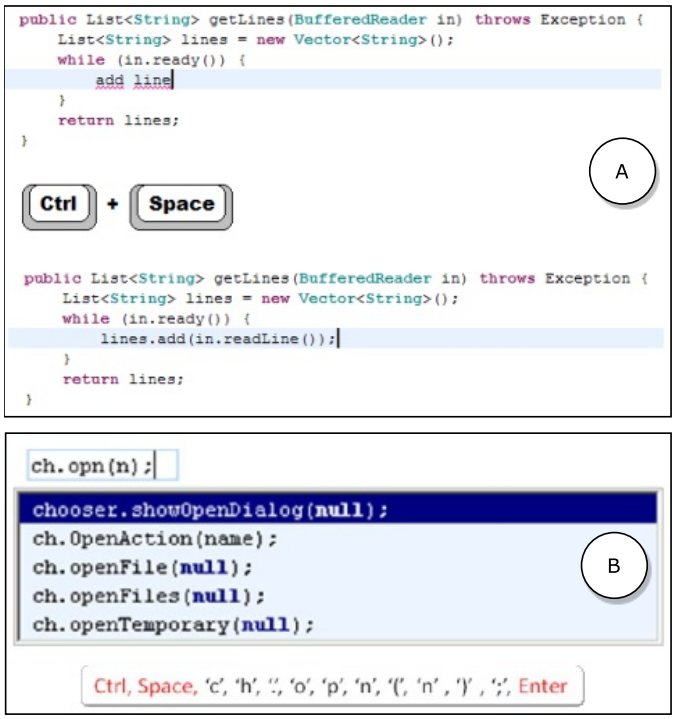


Figure 2.8: Examples of keyword-based code completion. The top figure (A) shows an example of code completion that translates input words into a valid expression (collected from Little and Miller [77]). The bottom figure (B) shows an example of abbreviated code completion (collected from Han *et al.* [40])

generated inputs reveals a number of challenges. For example, developers often type synonyms rather than actual function names. One possible way to improve the technique is to consider online tutorials to determine the mapping between the task description and the function name that is required to complete the task.

Another related work to this category of code completion is completing multiple keywords based on abbreviated input [40] (see Figure 2.8(B)). The technique utilizes a Hidden Markov Model to determine how to expand the abbreviated input using code examples. However, it does not require users to remember the abbreviated inputs; the system supports arbitrary inputs. Abbreviated input is also supported by various editors (such a GNU Emacs[8]). However, these editors only support a predefined set of abbreviated inputs. Each input expands into another text when inserted in the editor and the users need to memorize those inputs.

T2API is a statistical machine translation based tool that allows developers to write the textual description of a task and converts the query to an API usage example [95] develop. The technique leverages Stack Overflow[9] posts to determine the mapping between individual words and API elements that go together. The technique also uses a graph-based language model and determines API usage graphs using collected code examples. The technique processes the query to remove irrelevant words and determines the API

[8] https://www.gnu.org/software/emacs/
[9] https://stackoverflow.com/

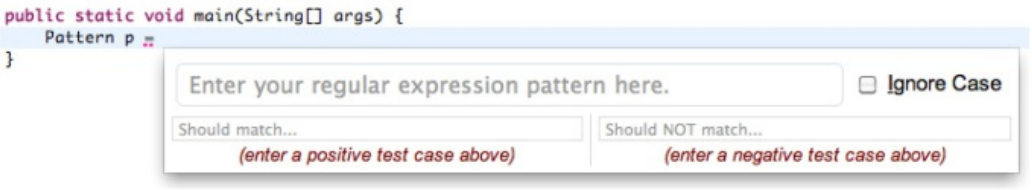


Figure 2.9: Completing a regular expression using the palette (collected from Omar *et al.* [102]).

elements that best match with the given query words. It then suggests the API usage example that covers the most API elements. While the previous works in this section focus on completing an expression or a statement, this technique focuses on completing multiple statements that represent an API usage example.

2.4.7 Other forms of code completion

This section presents code completion systems that differ from the techniques in the above categories in terms of their objectives. They either refine the code completion menu to bring additional sources of information, facilitate discovering of APIs or target completing language constructs other than those discussed above.

Omar *et al.* [102, 103] propose an active code completion system, an architecture that allows developers to integrate specialized code generation interfaces, called palettes, via the code completion command (see Figure 2.9). For example, creating a Color object in Java requires a developer to call its constructor that accepts red, green and blue color values as parameters. Instead of manually specifying the color values, a color palette can assist developers to select the color and generate the corresponding code in the IDE. The technique has been implemented in a tool, called Graphite (Graphical Palette to Help Instantiate Types in the Editor) that provides active code completion support in the Eclipse IDE. To keep the palette development independent from the IDE in use, palette developers use HTML5 technology to create palettes as webpages. Given a URL, the web browser control of IDEs can load the palette by showing the webpage. Graphite provides two different ways to associate palettes with a class. For example, the palette developer can associate the URL of the palette using annotation. However, if the palette developers do not have any access to the class, they can explicitly associate the palette to a class using the preference pane in the Eclipse IDE.

Stylos and Myers discover that developers face difficulties in using APIs where the method they are looking for is not accessible from the object they are currently working with [131]. To better understand the problem consider the task of sending a mail using Java Mail API. Here, a developer has created an object of *MimeMessage* type and then set different properties of the message using various setter methods. Empirical evidence indicates that developers tend to look for the send method in the *MimeMessage* type. However, the method they are looking for is not located in the type they started working. It is located in the *Transport* class and it is a challenging task for developers inexperience to Java Mail APIs to find this information.

Duala-Ekoka and Robillard [30] refers to this issue as an API discoverability problem. They use the term *main type* to refer to the type a developer is currently working with and the term helper type to refer to the

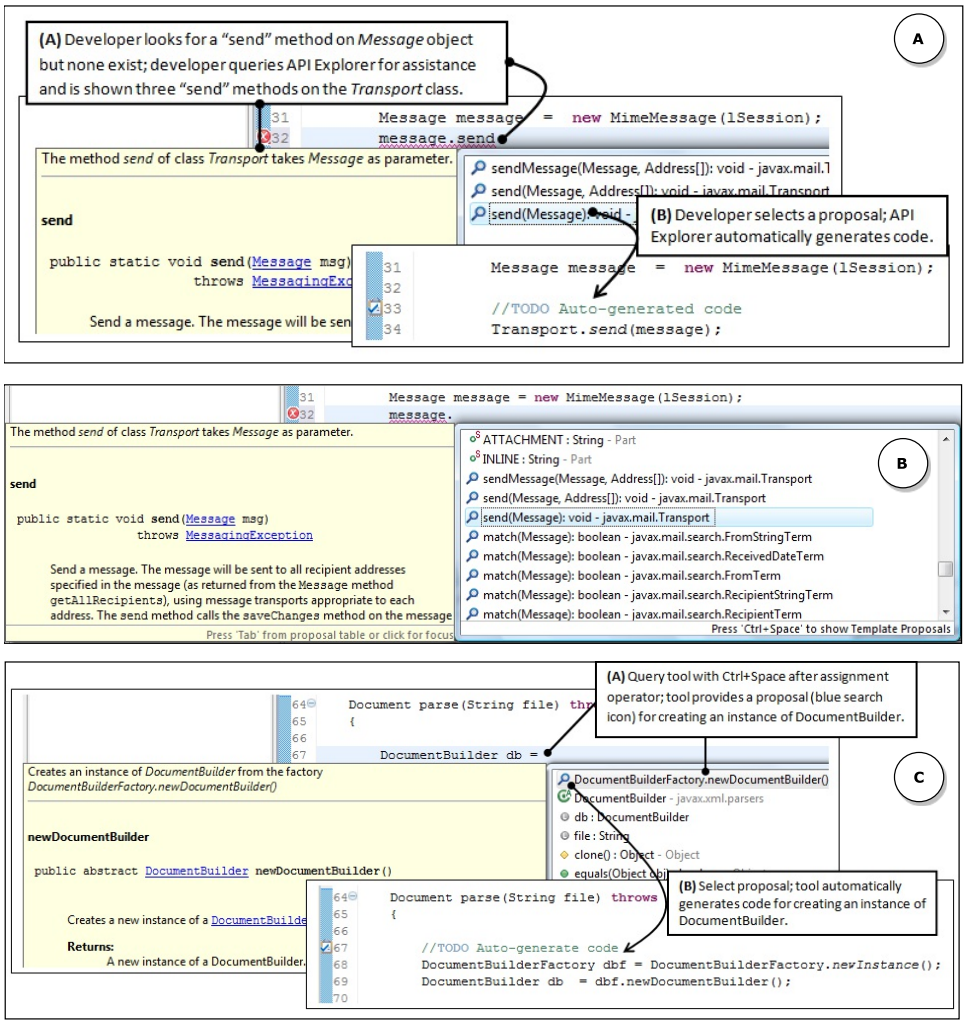


Figure 2.10: Examples of using API Explorer. It can recommend the send method call even if the method call is not accessible from the current receiver object (A). API explorer includes method calls from other related classes (B). It also allows construction of objects (C) (collected from Duala-Ekoka and Robillard [30]).

type that contains the method the developer is looking for. They develop a code completion system that can help discovering helper types and their methods by considering the structural relationship of API elements. This includes the return type relationships, subtypes relations, and the relationships between a method and its parameters. The technique has been implemented as a tool, called API Explorer (see Figure 2.10). The tool supports completing object construction expressions, suggesting methods that are located in helper types instead of the main type and showing the relationships of API elements through code completion menus.

Calcite [87] is an Eclipse plugin that leverages crowdsourcing to support automatic completion of constructors (see Figure 2.11 for examples of using the tool). This is important because an object can be constructed in various ways and not all of them are easy to discover for developers. For example, Ellis *et al.* [32] found that developers require more time to create objects using factories instead of constructors. The additional time is most likely contributed by the effort for discovering classes that are required to create the object.

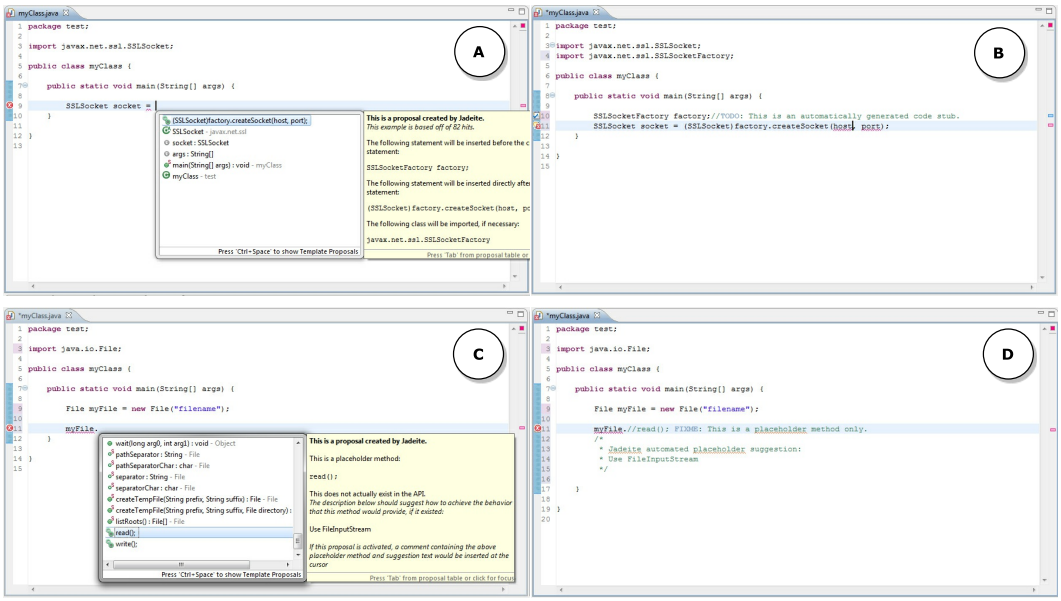


Figure 2.11: Calcite supports construction of objects. Figure 11-A shows the suggested object construction proposals and 11-B shows the output after selecting a completion proposal. The tool also supports recommending placeholder methods. Figure 11-C and 11-D are the output of before and after selecting a placeholder method proposal (collected from Mooty *et al.* [87]).

The tool uses a database that contains the object construction examples of more than 4000 public classes and interfaces in the Java 6 APIs. Instead of searching examples in the code repositories, Calcite uses Yahoo Search Engine[10] to find webpages containing code examples. Those examples are parsed to locate object construction examples and the technique approximates the type of the object being created. Developers can request for a constructor completion on the right-hand side of an assignment expression. Calcite determines the type of the left-hand side expression. It uses this information and the frequency to suggest constructor completions. In addition to the above, Calcite suggests placeholder methods. These are methods that do not exist in the APIs, but represent functionalities expected by developers. Selecting a placeholder method does not insert any code, rather inserts a recommended fix as a TODO comment that can be configured by the developer. The final addition to the code completion menu by Calcite is the placeholder suggestion prompt that when selected allows developers to include a placeholder method to a class including the process of achieving the desired functionality. It should be noted that Calcite shares the database of object construction examples from Jadeite (Java API documentation with Extra Information Tacked-on for Emphasis) [129]. Unlike Calcite, it is a Java-doc like API documentation system that contains information about the placeholder methods, common ways to construct classes, and variable font sizes (for packages, classes, and interfaces).

Lee *et al.* [74] develop a tool that integrates temporal dimensions and navigation to code completion (see Figure 2.12). Typically code completion systems focus only on the current version of the software. However, it may be the case that the API method or the field a developer is looking for has been deleted or replaced

[10] `https://ca.search.yahoo.com/`

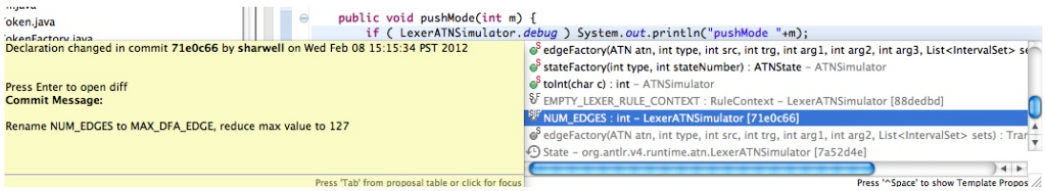


Figure 2.12: The code completion system recommends previously deleted field variables, highlighted in gray color (collected from Lee *et al.* [74]).

by another one. Their technique identifies the program elements in the current version, determines the mapping of them in the past revisions (if any), and shows this information in the code completion popup menu. The technique also shows the methods and fields in the past revisions in the completion menu and allows developers to open them in the IDE. This eliminates the need for context switching.

CookBook is a code completion technique that suggests potential edit recipes [54]. Each edit recipe can be considered as an abstract program transformation script that can suggest related changes. Given examples of changed methods, the technique applies an AST differencing algorithm to determine an edit script for each method. Each edit script consists of a set of line level edits including their edit types (e.g., insert, delete, change). These represent the training data. When a developer starts editing a line in a method, CookBook determines suitable recipes based on the context matching. Each time a developer edits a line, CookBook determines the edit type and the content. It then uses the information to find edit recipes that contain similar edits. CookBook calculates a matching score for each of those recipes by considering a combination of context and edit content matching scores, sorts them in terms of the score, and presents them to developers.

Chapter 3

Capturing code context and using the simhash technique: A case study on tracking source code lines

This chapter discusses an approach to capture source code context. To quickly validate the benefit of such context information and also the simhash technique, this study focuses on the problem of tracking source code lines. This is due to the simplicity in developing line location tracking techniques. Similar context information and the simhash technique are utilized in developing code completion systems in remaining chapters of this thesis. The line location tracking technique described in this chapter and the implementation of the technique have been published in a major software engineering conference [11, 12].

3.1 Introduction

Tracking source code across multiple versions of a program is an essential step for solving a number of problems related with multi-version program analysis. For example, consider the problem of locating bug introducing changes. Existing techniques solve this problem by finding the lines that are affected through bug fixes and then trace back those lines to determine their origin. If a bug has been identified in a software system, tracking lines containing the bug in the subsequent versions can help us determine whether the same problem persists in the next versions and if yes, allows developers to fix the problem at ease. Results obtained from a line tracking technique can be aggregated for fine-grained evolutionary analysis. For example, clone evolution analysis requires tracking clone fragments across multiple versions of a software system. Tracking lines of a clone fragment can help us understand how that fragment evolves over time. Such analysis can also guide us in understanding the nature, effects, and reasons for cloning. To support collaboration in software development, annotation or tagging has been used in source code that can facilitate both navigation and coordination [128], and source location tracking techniques can help manage tags across versions. Software development involves more than just the creation of source code. There are also different kinds of software artifacts that can benefit from mapping lines across versions. Possible applications are, but not limited to, studying when and how requirements are changed or ensuring whether intended changes have been applied to all configuration files.

A number of line tracking techniques can be found in the literature. Reiss [111] listed a number of approaches for tracking lines across versions and found that language-independent approaches often provide

good results. Canfora *et al.* [22] proposed another language-independent line matching technique, called *ldiff*, that uses a combination of information retrieval techniques and Levenshtein distance for mapping lines. Techniques that take into account the syntactic structure of source files can provide change information at a more fine-grain level [35]. An example of a line tracking technique that falls in this group is *SDiff* [127] that can determine changes at method and identifier levels. If fine-grain source code change information is not required, language-independent text-based approaches are suitable for tracking source code lines and can be applied to different kinds of documents besides source code (e.g., test cases or use cases). They also have the potential to be integrated with existing version control systems with little or no modification. However, the performance of these techniques varies depending on the degree of changes applied to the source code. William and Spacco [127] found that techniques that performed well in the Reiss study did not perform well against their benchmark in which files had a higher degree of changes. This motivates us to investigate the reasons behind this discrepancy and to device a robust language-independent solution.

This chapter introduces *LHDiff* (Language-independent Hybrid Diff), a language-independent technique to track the evolution of source code lines across versions of a software system. The technique uses *Unix diff* between two different versions of a file to determine the set of unchanged lines. To track the remaining lines, it uses a combination of context and content similarity. However, to speed up the mapping process, it first leverages *simhash* technique to determine a list of mapping candidates for each deleted line in the old file. Next, the technique computes similarity scores again, but this time on the source code lines instead of the *simhash* values, to select one line from each set of mapping candidates. To validate the effectiveness of our language-independent technique, we compare it against state-of-the-art techniques using different benchmarks where the files are collected from real world applications. We further evaluate our technique with other approaches using different types of changes in a mutation based analysis. The experimental results in both cases suggest that the technique is superior to other language-independent approaches, and even often gives better result than the language-dependent technique *SDiff*.

The remainder of the chapter is organized as follows. Section 3.2 covers previous work related to our study. Section 3.3 describes the hybrid line mapping technique. Section 3.4 presents a quantitative evaluation of line tracking techniques using three different benchmarks. In Section 3.5, we describe results of mutation based analysis. Section 3.4 explains some threats to our study and finally, Section 3.7 concludes the chapter with future research directions.

3.2 Related Work

Tracking source code lines across program versions is crucial to support various maintenance activities and several approaches exist that consider line content, context, abstract syntax tree, edit distance, or a combination of these techniques to solve the problem. In general, existing techniques can be divided into two categories: (1) text-based and (2) syntax tree-based. The first group of techniques is purely textual in nature

and does not require parsing source files. As a language-independent technique the *Unix diff* algorithm has been widely used in many studies, not only to track lines but also for program differencing. *Diff* is based on solving the problem of longest common subsequence and it reports the minimum number of line changes that can convert one file to another. However, it has its limitations. For example, it cannot detect reordered lines. The addition of comments to a line can cause *diff* to report deletion of the old line and addition of a new line. However, such cosmetic changes are irrelevant to a programmer and should be ignored [60].

Diff reports regions of file lines that differ between a pair of files where each region is called a hunk. Zimmermann *et al.* [147] addressed this modification changes using an annotation graph where large modifications are considered as combined addition and deletion of lines, otherwise all lines between hunk pairs are connected with each other in a modification. The technique detects origins conservatively, does not consider the issue of reordered lines and is susceptible to errors.

Canfora *et al.* [22, 23] developed a line differencing technique, called *ldiff*, to track line locations independent of languages. Their technique uses the *Unix diff* algorithm to determine the unchanged lines. After that, set-based and sequence-based metrics are used to complete the mapping of remaining lines. However, they only compared the technique with *Unix diff* algorithm. To the best of our knowledge, Reiss was the first to conduct a study to evaluate the performance of several techniques for tracking source locations [111]. Interestingly, the result reveals that simple techniques like the one mentioned above that do not consider program structure perform better than those that consider syntactic structure of source files (like abstract syntax tree-based techniques). Reiss also recommended the *W_BESTI_LINE* technique to track line locations, which uses a combination of context and content similarity to find the evolution of lines independent of languages. While *LHDiff* is also language-independent, our technique differs from the above approaches in that it uses the *simhash* technique to speed up the mapping process and a set of heuristics to improve the effectiveness of tracking source locations.

Spacco and Williams [127] extended the idea of tracking lines for tracking program statements across multiple revisions of Java code. They developed *SDiff*, an abstract syntax tree based technique that leverages tokenization, *Unix diff* and *Kunhn-Munkres* algorithm [71] to complete tracking line locations. *SDiff* is a great algorithm to determine differences at the line level. However, the technique cannot be applied to arbitrary source code and cannot handle comments. While comments are not executed, their importance cannot be ignored. Tags [128] are usually associated with comments to support asynchronous collaboration and they need to be tracked across versions. Moreover, *SDiff* requires the source code to be parsed without any error. However, in reality this cannot be guaranteed. For example, the source code may be written using an old grammar of a language or developers may issue file differencing commands in the middle of an edit operation. *LHDiff* on the contrary is a language-independent technique, can be applied to arbitrary source code, and can track the evolution of comments/tags across versions.

Line location information can be obtained through techniques that determine fine-grain differences between two versions of a file. However, these techniques require knowledge about language constructs. Among

these approaches, most notable is the ChangeDistiller [35], which considers the abstract syntax tree of a Java source file and leverages a tree differencing algorithm to determine fine-grain changes in the source code. The algorithm is not immune to cosmetic changes (changes that do not affect the behavior of a program like addition of a comment), cannot work on arbitrary text files and is limited to Java files only. Xing and Stroulia [141] presented an algorithm, known as UMLdiff to determine structural changes in object-oriented software. It uses a combination of name similarity and structure similarity measures for recognizing conceptually the same entities. Apiwattanapong [3] presented an algorithm to determine changes between two Java programs. The technique considers program structure and semantics of programming language constructs to determine changes that are difficult to detect with a pure textual differencing technique. While these approaches are language specific and focus on fine-grain change details, *LHDiff* focuses on tracking lines independent of source code languages.

Techniques for tracking program elements across versions are also related to our study. Matching higher level language constructs prior to mapping lines improves mapping quality. A comprehensive survey of various techniques can be found in the work of Kim *et al.* [65]. Godfrey and Zou [39] developed a semi-automatic technique to detect merging and splitting of source code entities during the evolution of a software system. Kim *et al.* [64, 66] used a combination of textual similarity and a location overlapping score to track clone fragments across versions. Duala-Ekoko and Robbillard developed a technique to track the evolution of clones [29]. The technique is also available as an Eclipse-plugin, called Clone Tracker. Here, clones are identified using an abstract clone region descriptor (CRD) that is independent of source code line locations. While the above techniques focus on tracking code fragments, we focus on tracking individual lines.

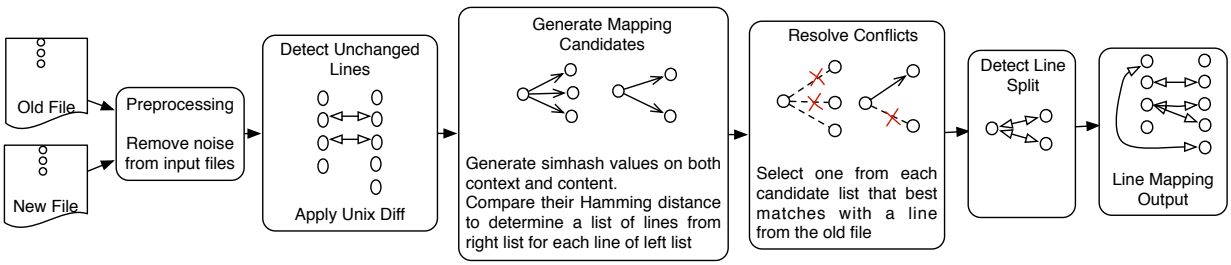


Figure 3.1: Summarizing the line mapping process in LHDiff

3.3 LHDiff: A Language-Independent Hybrid Line Tracking Technique

This section introduces *LHDiff*, our language-independent hybrid line tracking technique. We refer to *LHDiff* as a hybrid technique since we leverage different techniques collected through manual investigation of incorrect mappings of other techniques. Figure 3.1 summarizes the entire mapping process.

3.3.1 Preprocess input files

The algorithm starts with reading lines from two different versions of a file. A large number of changes in source code are only cosmetic in nature and do not change the behaviour of a program. Examples include changes to whitespace/newline characters. They are inserted to change the indentation of a program to improve readability. To ignore such changes, each line of the source file is normalized so that multiple spaces are replaced by only one. We also remove all parentheses and punctuation symbols from the text except curly braces because we obtained best results when considering them as part of the line context.

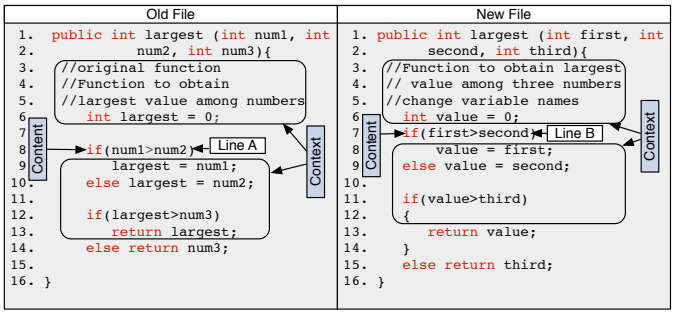
3.3.2 Detect unchanged lines

After preprocessing we apply *Unix diff*, which uses the longest common subsequence algorithm to determine the set of unchanged lines. We use *diff* because previous studies report that it can detect the set of unchanged lines with great accuracy [111]. *Diff* reports the sequences of lines that have been deleted or added between the files. We store the list of deleted and added lines into two different lists. From now on, we refer them as the left and right lists correspondingly.

3.3.3 Generate candidate list

Now, our goal is to determine the mapping of a line from the left list to that of the right list. Similar to *W_BESTI_LINE* (recall that *W_BESTI_LINE* is another language-independent source location tracking technique), we use both context and content of a line to determine the correct mapping. The line itself represents the content and the context is created by concatenating four lines before and after the target line. While the content similarity between a pair of lines is calculated using normalized Levenshtein edit distance which considers the order of characters in them, the content similarity is calculated applying cosine similarity[1] that does not consider the order of tokens/characters. While building context, we ignore blank lines, but keep the curly braces. Such changes allow us to gather sufficient contextual information for a line. We then determine a combined similarity score by considering both content and context similarity. We need to calculate such scores between all possible pairs of deleted and added lines. After that we map only those lines that provide the highest similarity scores and also exceed a predefined threshold value. However, the complete operation would take a long time to finish because of the complexity associated with computing similarity scores, particularly the normalized Levenshtein edit distance. To improve the running time of the algorithm we follow a different strategy. We first apply a form of locality sensitive hashing and calculate the combined similarity score of the hash values instead of the original lines to determine a small set of possible mapping candidates for each line of the left list. Then we use the original line content and combined similarity score to select a line from each set of mapping candidates. Since the hashing technique reduces large data into a much shorter sequence of bits, the overall mapping time is reduced significantly.

[1] `http://www-nlp.stanford.edu/IR-book/`



Context	Simhash	Hamming Distance
Line A	1000100111011010100000000001011	9
Line B	1110000010100101011000001001011	

Figure 3.2: An example of calculating Hamming distance

While a cryptographic hash function tries to avoid generating the same key to prevent collisions, in this form of hashing, files containing similar content are mapped to identical or very similar binary hash keys. The technique is known as *simhash* [24] and it has been found that the technique is practically useful to determine near-duplicate pages in a large collection of documents [80]. The core of the algorithm uses a hash function to generate *simhash* values. Among various non-cryptographic hash functions we use *Jenkin* hash function since it shows better similarity preserving behavior compared to other functions and it was also found to be effective in detecting near-miss code fragments in a different study [138]. We generate a 64 bit *simhash* value for both context and content using the *simhash* algorithm.[2] Instead of working on the original lines, we are now working on the *simhash* values. We now calculate the context and content similarity for each pair of added and deleted lines by calculating the Hamming distance between their corresponding *simhash* values. While the Hamming distance between two strings is the number of substitutions required to convert one string into another, for binary strings (a and b), Hamming distance is calculated by counting the number of 1 bits in $a \oplus b$ (bitwise exclusive OR operation between a and b). The smaller the Hamming distance is, the closer the two strings are (see Figure 3.2). The context and content similarity values are normalized between zero and one. A combined similarity score is calculated for each pair of lines using 0.6 times the content similarity and 0.4 times the context similarity (this is determined after experimenting with different combinations of values). For each line on the left list, we then determine the k-neighbors from the right list that are the most probable mapping candidates of that line based on the combined score. We refer to this set as the matching candidates list. Since we are not comparing raw source code lines for selecting the mapping candidates, this saves significant time. During our study with different values of k we found that $k = 15$ is a good choice.

[2] http://d3s.mff.cuni.cz/∼holub/sw/shash/

3.3.4 Resolve conflicts

We now have a candidate list for each line of the left list (except those that are detected as deleted/unchanged by the algorithm), but we do not know the exact mapping yet. The objective of this step is to select one line from each candidate list to resolve the conflicts. As an example, consider that we are looking for the mapping of line 20, and from the previous step we determine that the candidate list consists of three lines $(37, 46, 51)$. We now use the original lines to generate both context and content, and the algorithm determines the combined similarity score between each possible mapping pairs ($\{20, 37\}, \{20, 46\}$, and $\{20, 51\}$). It selects the one that gives the highest similarity score and also exceeds a predefined threshold value. We now use normalized Levenshtein distance to measure context similarity and cosine similarity to measure content similarity. Both the values are normalized and the combined similarity score is determined using 0.6 times the content similarity and 0.4 times the context similarity. These were the same values used by Reiss. We set the threshold value to 0.45 after experimenting with various other values because at this setting *LHDiff* provides good result.

3.3.5 Detect line splits

The last part of our technique deals with detecting line splitting. We use the term line splitting instead of statement splitting since *LHDiff* is not aware of the boundary of a statement, and only works at the line-level. An example of line splitting is shown in the Figure 3.3 where a single line breaks into multiple small lines. The basic *LHDiff* algorithm tries to map each line from the old file to a line in the new file, but fails to map some lines where the textual similarity differs a lot. To track lines affected by the line splitting, we use the following approach. For each unmapped line of the new file, we repeatedly concatenate the line to the successive lines, one at a time, and determine normalized Levenshtein distance (LD) with another unmapped line in the old file until the similarity value starts to decrease. Then, if the textual similarity between the concatenated lines and the left side line crosses a predefine threshold value, we map them. As an example, we concatenate line 20 with 21 and determine the normalized Levenshtein distance between R_{20+21} and L_{40}. The similarity value is greater than the similarity between R_{20} and L_{40}. Thus, we concatenate the next line and determine the similarity again between $R_{20+21+22}$ and L_{40}. Since the similarity is again greater than the previous step we continue the concatenate operation until the similarity decreases. This happens as soon as we add line 24 ($LD(R_{20+21+22+23+24}, L_{40}) < LD(R_{20+21+22+23}, L_{40})$). This indicates that line 24 cannot be

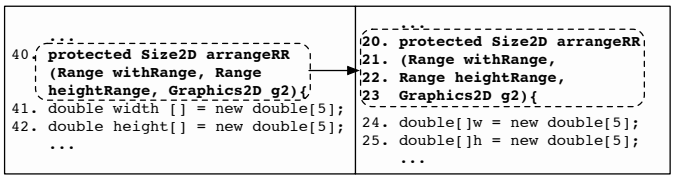


Figure 3.3: An example of a line splitting

a part of the split lines. Since $LD(R_{20+21+22+23}, L_{40})$ exceeds the threshold value and there are four lines in the concatenated part, *LHDiff* maps all these four lines to line 40 on the left side. To avoid false mapping, we set the threshold value to a high value, 0.85, and we also limit the concatenation to a maximum of 8 lines. During our manual analysis, we did not find any example where a line splits more than that. This heuristic approach can only compensate line splitting when such an operation does not change the contents of the line or changes only little. It cannot detect other complex line splitting operations, such as those described in Section 3.5.1.

3.3.6 Evaluation

We evaluate the effectiveness of *LHDiff* using two different methods. First, we consider three different benchmarks, each of them containing line mapping information of versions of files where the files are collected from real world applications. Second, we also compare *LHDiff* with other state-of-the-art techniques using a mutation based analysis where we consider different types of changes. We describe details about these two evaluation methods in the following two sections.

3.4 Evaluation Using Benchmarks

This section describes the benchmarks we used to evaluate source location tracking techniques including results of our evaluation.

3.4.1 Experiment details

We used two different benchmarks available from previous studies to measure the effectiveness of source location tracking techniques. The first one was developed by Reiss, which contains location information of 53 lines of a file (ClideDatabaseManager.java) in 25 revisions and amounts to 1325 test cases [111]. We also evaluated our technique with another benchmark developed by Reiss that contains locations of 14 lines of

Table 3.1: Three forms of incorrect mappings

Label	Meaning
Change	the algorithm finds a mapping of a line but the mapping is not correct
Spurious	the algorithm detects mapping of a line but the line is deleted
Eliminate	the algorithm detects deletion of a line but the line exists in the new file

Table 3.2: Running location tracking techniques against Reiss, Eclipse, and NetBeans benchmarks

Type	Method	Reiss Benchmark					Eclipse Benchmark					NetBeans Benchmark				
			Incorrect Mapping Categories					Incorrect Mapping Categories					Incorrect Mapping Categories			
		Correct%	Change%	Spurious%	Eliminate%	Time [s]	Correct%	Change%	Spurious%	Eliminate%	Time [s]	Correct%	Change%	Spurious%	Eliminate%	Time [s]
Language independent Techniques (Text-based approach)	W_BESTI_LINE	96.7	0	29.6	70.5	4.8	52.6	28.2	56.4	15.5	0.7	61.9	41.9	45.2	12.9	2.9
	LHDiff	**97.0**	**42.5**	**0**	**57.5**	**4**	**82.8**	**22.5**	**20**	**57.5**	**3.3**	**85.5**	**18.6**	**13.6**	**67.8**	**6.8**
	diff	92.5	0	0	100	0.2	41.0	0.7	0	99.3	0.1	48.4	1.4	0	98.6	0.2
	ldiff [-i 0]	96.4	0	66.7	33.3	25.7	62.5	9.2	1.2	89.7	58.2	74.0	16.0	7.6	76.4	6.1
	ldiff [-i 1]	96.4	0	25	75	73.9	59.1	9.5	1.1	89.5	209.2	76.4	14.6	5.2	80.2	103.5
	ldiff [-i 3]	96.2	0	29.4	70.6	118.6	72.0	20	4.7	75.4	419.7	80.6	24.1	8.9	67.1	171.5
	ldiff [-i 5]	96.2	0	29.4	70.6	160.7	72.4	20.3	4.7	75	472.4	81.3	25	9.2	65.8	218.2
	Git	92.5	0	0	100	0.8	45.3	0.8	0	99.2	0.3	53.7	1.6	0.5	97.9	0.2
Language Dependent Technique (Requires access to AST)	SDiff	86.2	0	86.3	13.7	1.9	70.7	11.8	80.9	7.4	1.7	71.8	7.0	86.1	7.0	4.2
	SDiff-probable	87.0	3.0	82.4	14.6	1.7	69.9	11.4	75.7	12.9	1.5	73.5	9.3	82.4	8.3	4.3
	SDiff-possible-probable	87.0	3.0	82.4	14.6	1.7	70.7	11.8	75	13.2	1.6	75.4	12	77	11	4.3
	SDiff-min	85.6	0	82.5	17.5	2.6	74.1	13.3	76.7	10	2	72.5	7.1	85.7	7.1	6.8
	SDiff-min-probable	86.4	2.9	78.6	18.5	2.6	73.3	12.9	74.2	12.9	2.1	74.2	10.5	81.0	8.6	6.8
	SDiff-min-possible-probable	86.4	2.9	78.6	18.5	2.6	74.1	13.3	73.3	13.3	2.1	75.2	12.9	74.3	12.9	6.8
	SDiff-token	85.6	0	82.5	17.4	1.0	73.3	11.3	79.0	9.7	1.0	71.0	6.8	87.3	5.9	2.3
	SDiff-token-probable	85.6	2.7	79.8	17.5	1.0	73.3	11.3	75.8	12.9	1.0	73.7	10.3	82.2	7.5	2.3
	SDiff-token-possible-probable	86.4	2.9	78.7	18.5	1.0	74.1	11.7	75	13.3	1.0	74.5	11.5	76.9	11.5	2.3

the JiveRuntime.java file in 27 different revisions. However, we did not report the result because the changes are simple and there is no significant difference among source location tracking techniques for those changes. In both cases, lines were selected by looking at every tenth line of the source file discarding those containing blank lines. Additional problematic or interesting changes (such as name and comment changes) were also included. Williams and Spacco developed another benchmark (known as Eclipse benchmark) containing the change information of 232 lines [127]. The author of this thesis manually analyzed all changes. This was done to validate changes and to identify interesting change patterns during the evolution of these lines. It should be noted that the manual investigation revealed a few incorrect mappings in the Eclipse benchmark. We used the benchmark in our evaluation after correcting those mappings. That is why readers will notice a slight difference in our results for Eclipse benchmark than what was reported in the original paper [127].

In addition, we also developed a third benchmark using source code from the NetBeans project[3]. NetBeans is an integrated development environment for developing applications using different languages. We randomly selected a number of lines (including those we found challenging) and then determined the new locations of those lines in another version. Since the decision is subjective in nature, to avoid bias we manually determined the correct mapping of these lines separately. Cases where there was a disagreement between the two authors, we removed that line from our study.

Although our technique is language-independent, we considered techniques in both categories for comparison. We instructed *ldiff* to ignore all whitespaces and also to ignore changes whose lines are all blank. For all other settings of *ldiff* we used default values except we changed the number of iterations to four different values. When the value is set to 0, *ldiff* considers only line similarity. For all other positive values it considers hunk similarity. In the case of *SDiff*, we evaluated all different nine configurations used in the original experiment. For details of these configurations we refer interested readers to their original paper [127]. Among various techniques evaluated by Reiss, we report the result of *W_BESTI_LINE*, with the same similarity thresholds and context length suggested by Reiss, because Reiss recommended this technique for tracking lines. We also include results of *diff*, configured to ignore spacing differences and cases of letters during the mapping process.

To calculate a score for each method, we followed the basic scoring mechanism used in previous studies where the score was calculated by determining the number of correct mappings. To document line mapping information, we used the following approach. If a line in the old file is deleted, the new location of that line is encoded with -1, otherwise each line of an old file is associated with a non-negative integer representing the new position of that line. Incorrect classifications can result from three different types of errors and we use different labels to signify them as shown in Table 3.1. All experiments were conducted on a computer running with Ubuntu Linux that has a 2.93 GHz Intel Core i7 processor and 8 GB of memory.

[3] `https://netbeans.org/`

3.4.2 Results

Table 3.2 shows the results of our evaluation. For each benchmark and line tracking technique, we not only provide the percentage of correct mappings but also show results for each incorrect mapping type (these are: change, spurious, and eliminate; see Table 3.1 for their definitions). From the table we can see that *LHDiff* performs better than both *SDiff* and *ldiff*. The default setting of *ldiff* uses cosine similarity for mapping hunks and Levenshtein distance for mapping lines. We considered all possible combinations of metrics and tokenizers. While for the Reiss benchmark *ldiff* correctly maps around 96.5% of lines, the performance drops significantly for the Eclipse benchmark. This is similar to the result provided by Spacco and Williams. On the other hand, although *W_BESTI_LINE* performs similar to *LHDiff* for the Reiss benchmark, accuracy drops to around 53% for the Eclipse benchmark. The latter contains a large number of changes as the files are heavily edited in it. The performance of *W_BESTI_LINE* and *ldiff* varies depending on the number of changes occured to the files. However, both *SDiff* and *LHDiff* are stable in nature. Although *SDiff* shows around 87% accuracy for the Reiss benchmark, according to the authors accuracy can be even up to 96% if we ignore curly braces and non executable statements. For the Eclipse benchmark, *SDiff* produces around 74% accurate result. However, *SDiff* takes into account the structure of source files, requires the files to be syntactically valid, and is only available for Java. *LHDiff*, on the other hand, is purely textual in nature and can be applied to any files (be it a source file or a plain text file). Without considering structural information, *LHDiff* provides around 97% of correct mappings for the Reiss benchmark and for Eclipse, the result is around 83%. Among the incorrect mappings we found for the Eclipse benchmark, the highest amounts (more than 57%) were due to elimination.

For the NetBeans benchmark (see Table 3.2), *LHDiff* shows better results than the other techniques. While *LHDiff* detects around 86% correct mapping, *SDiff* detects only around 75%. Although *ldiff* detects more correct mappings (81%) than *SDiff*, it requires more computation time.

3.4.3 Discussion

In this section, we first explain the reasons behind the poor results of *W_BESTI_LINE* and also *ldiff* on the Eclipse benchmark that originally motivated us to develop another language-independent source location tracking technique. We then compare *LHDiff* with the content tracking technique of *Git* and present another study result where we tried to improve the accuracy of *LHDiff* using tokenization.

Why did the technique recommended by Reiss or *ldiff* not perform well with Eclipse benchmark?

The algorithm (*W_BESTI_LINE*) recommended by Reiss works fine as long as the context and content of a line do not change significantly. The technique fails when both of them go through a large amount of changes. Another possible threat to this approach is the addition of blank lines or lines containing stop list

or punctuation symbols only. If a developer inserts blank lines before and/or after a line, the line becomes isolated and the context differs remarkably. If the line also changes significantly, the algorithm fails to map the old line to the new one. Reiss did not consider this issue. However, we can eliminate this problem by ignoring blank lines which can help locate proper contextual information. Another downside of this technique is that the cost of running the technique is high. For a line in the old version, the technique compares the line with every other lines in the new version and selects the one that provides the best matching. If the objective is to determine new locations of a few lines of the old file, then the technique may be adequate. However, situations where we need to map every line of an old file to the new file, then the approach may be expensive particularly when the size of the file is large. That is why we use the *simhash* technique in *LHDiff* to faster the mapping process. *W_BESTI_LINE* tries to map a subset of lines of an old file to the new file whose mappings are requested by a user. While this approach lessens the running time of the algorithm, it drops the accuracy of the technique, particularly where files have gone through a large number of changes. A line ($l_i^{old} \in f_{old}$) cannot be aggressively mapped to another line ($l_j^{new} \in f_{new}$) without considering the similarity of l_j^{new} to all other lines of the old file. Their might be another line ($l_k^{old} \in f_{old}$) to which the newly mapped line ($l_j^{new} \in f_{new}$) best matches. For these reasons *W_BESTI_LINE* performed poorly for the Eclipse benchmark.

The running time of *ldiff* improves because of applying *Unix diff* at the beginning to determine unchanged lines. A threat to the technique comes from the fact that before mapping lines, *ldiff* tries to map a hunk from the old file to another hunk of the new file. If the hunk similarity exceeds a threshold value then a line mapping process begins. However, there is a possibility that hunk similarity does not exceed the threshold value because of their size difference and only a few lines are common between the hunks. In such a situation *ldiff* reports them as deleted and added lines which contributed to its poor performance for the Eclipse benchmark.

How is *LHDiff* comparable to the content tracking technique of Git?

Some version control systems (like CVS or SVN) change code authorship of a line even if the line goes through formatting changes or moves to a different location. In both cases, the line is reported as a new one. Git overcomes such a limitation by tracking the content of a file across versions. Git *blame -C -M* command can track the movement of unchanged lines. Here, -C finds code copies and -M finds code movement. However, if the lines move even with slight changes, Git assigns code authorship to the new author and marks them as newly created, although ideally these lines come from different locations of the previous version. To compare the content tracking technique of Git with *LHDiff*, we first commit each pair (the original one and its new version) of file versions of a benchmark in a local repository and then apply the *git blame -M -n* to determine the origin of line locations of the new file. Here, *-n* tells Git to show the line numbers in the original commit which is by default turned off. The results are summarized in Table 3.2. In general, the mapping accuracy is similar to diff, except it detects more correct mappings for some cases where lines are moved within files (such as encapsulation, function split and reordering categories).

Table 3.3: Results of LHDiff before and after applying tokenization

Benchmark	LHDiff	Correct%	Change%	Spurious%	Eliminate%
Reiss	Non-tokenize	96.98	42.50	0	57.50
	Tokenize	93.66	0	48.81	51.19
Eclipse	Non-tokenize	82.76	22.50	20	57.50
	Tokenize	82.76	47.50	32.50	20
NetBeans	Non-tokenize	85.50	18.64	13.56	67.80
	Tokenize	82.56	32.39	9.86	57.75

Besides *blame*, Git also offers *pickaxe* to dig deeper into the history. While it can find the commits that change a block of code in a file, it cannot find the list of commits that contain the block of code. In general, Git focuses more on tracking code blocks instead of individual lines and stays out of the advanced diff or line tracking technique, possibly because of eliminating the overhead of running such an algorithm. Thus, the line level content tracking technique of Git is not as powerful as *ldiff* [16] or *LHDiff*.

Can tokenization improve the performance of *LHDiff*?

Source code can go through various cosmetic changes that can affect performance of text-based source location tracking techniques. For example, the order in which parameters appear in the method definition can be changed. In object-oriented programming languages, developers often use the *this* keyword to access object fields or methods which does not change the behavior of the program. The access modifier of a class can change from private to public to make a class visible to all other classes. Handling these changes requires accessing individual source code elements and in this regard tokenization can assist us.

Tokenization is the process of converting a sequence of characters of a source file into a set of tokens, where each token is a string of characters representing a category of symbols. Tokenization does not require parsing source files and can be implemented using regular expressions. We anticipate that instead of working on raw source files, working on the tokens may help us to ignore the effect of source code changes and thus, improve the accuracy of the algorithm. To verify this, we first tokenize source files of our benchmarks according to the lexical rules of the Java programming language. During this process we keep track of the line locations of each token so that we can construct source files with tokens later. Next, we apply a set of transformation rules to normalize token sequences (an example is shown in Figure 3.4). This step is necessary to eliminate differences between source code versions. We then reconstruct source files using transformed token sequences and use *LHDiff* to track source code lines across file versions.

Working on the tokens does not improve the performance of the *LHDiff* algorithm. While for the Eclipse

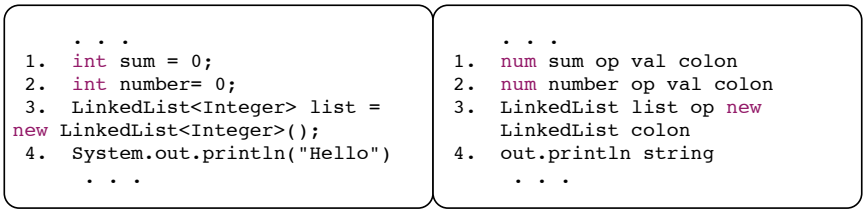


Figure 3.4: Example of tokenization

benchmark tokenization does not change mapping quality, performance drops by 3% for the NetBeans benchmark. When we examined those lines that were not correctly mapped by *LHDiff*, we found that working on the tokens eliminates differences between file versions. We find that for some cases, it helps to correctly map lines that were earlier detected as deleted (the percentage of incorrect mapping for *eliminate* category drops from 67.8% to 57.75% for the NetBeans benchmark). However, for some cases it leads to false mapping too (the percentage of spurious mapping increases from 18.64% to 32.39%). We observe a similar picture for other benchmarks also. Though tokenization does not require parsing of source code, it requires knowledge of the tokens of programming languages. Since our objective is to develop a language-independent solution, *LHDiff* does not include tokenization as a part of the detection process. Moreover, our study results reveal that even tokenization can lead to a poor result.

Although the above results show the performance of *LHDiff* over other methods, it does not explain how source code changes affect line tracking techniques. To find this out, we use a mutation based analysis that considers the editing taxonomy of source code changes as described in Section 3.5.1.

3.5 Evaluation Using Mutation-based Analysis

While previous studies evaluate source location tracking techniques against manually verified line mapping data, there is not much discussion about the types of changes appearing in them. This opens the question of how stable/vulnerable the techniques are to different change groups and makes it difficult to compare them in an objective fashion. Instead of a random selection of lines from source code and tracking their positions in subsequent versions, we use a taxonomy of source code changes to synthesize new lines and evaluate the techniques objectively. Toward this goal, this section first presents an editing taxonomy that captures typical actions performed by developers during source code evolution and then uses a mutation based analysis to evaluate line tracking techniques based on the taxonomy.

3.5.1 Editing taxonomy of source code changes

We developed an editing taxonomy by studying a large body of published work [34, 39, 65, 67], clone taxonomy [119], our previous study on code clones [122], and through manual investigation of the previous benchmarks [111, 127]. We further refined our taxonomy by analyzing changes of two open source Java

Table 3.4: Editing Taxonomy of Source Code Changes

No	Name	Example		Description
		Old File	**New File**	
1	Line Splitting/Merging	`else if (list.get(i)%3 == 0)`	`else` `  if (list.get(i)%3 == 0)`	An *else if* statement splits into two lines

		`if (list.get(i)%3 == 0) sum += list.get(i);`	`if (list.get(i)%3 == 0){`
			`  System.out.println("i = " + i);`
			`  sum += list.get(i)`
			`  }`
				A statement in one line splits into multiple lines where each contains one statement and some lines are added in between them.

		`list.add(Integer.parseInt(line));`	`number = Integer.parseInt(line);`
			`System.out.println(number);`
			`list.add(number);`
				Another example of line splitting of the above kind.

2	Function Splitting/Merging	`public int sum(File file){`
		`//Read numbers from the file`
		`//Store them in a list`
		`  . . .`
		`//Now process the list`
		`//Calculate sum of the numbers`
		`  . . .`
		`return sum;`
		`}`
			`public ArryaList readFile(Filefile){`
			`  //Read numbers from the file`
			`  //Store them in a list`
			`  . . .`
			`  return numberList`
			`}`
			`public int sum(File file){`
			`  ArrayList list = readFile(file)`
			`  //Now process the list`
			`  //Calculate sum of the numbers`
			`  . . .`
			`  return sum;`
			`}`
				A developer creates a function *readFile* with some lines from *sum* and replace those lines in *sum* by adding a call to that function in the new version of *sum* or vice versa

3	Wrapping/ Unwrapping	`while((line = br.readLine())! = null)`	`try {`
			` while((line = br.readLine())! = null)`
			`}`
			`catch (IOException e){`
			`  e.printStackTrace()`
				A line in the old file moves to a try-catch block in the new version of the file or vice versa

4	Change in Data Structure	`ArrayList list = new ArrayList();`	`LinkedList list = new LinkedList();`	*ArrayList* data structure is replaced by *LinkedList* or vice versa
5	Renaming	`sum = sum + list.get(i);` `return sum;`	`total = total + list.get(i);` `return total;`	Variable *sum* is renamed to *total* or vice versa

6	Code Reordering	`public ArrayList readFile (File file) {`
		`  . . .`
		`}`
		`public int sum (File file) {`
		`  . . .`
		`}`
			`public int sum (File file) {`
			`  . . .`
			`}`
			`public ArrayList readFile (File file) {`
			`  . . .`
			`}`
				Functions *sum* and *readFile* are reordered or vice versa

systems (NetBeans[4] and iText[5]). We choose Java because of our familiarity with this programming language which helped us to determine correct mapping of lines with less confusion. Discussing the frequency of the changes in software systems is out of the scope of this study.

Table 3.4 shows our editing taxonomy of source code changes. For each change type, we also provide an example that shows the change from the old version of a file to the new version of that file. The edit operations describe in the taxonomy are not mutually exclusive. Instead, they can be applied together to create more complex changes. To save space, simple edits (addition, deletion or modification of lines) are not discussed although they are the building blocks of all kinds of edits.

The first group of change in our taxonomy is line splitting/merging. Use of code formatters can trigger line splitting to improve readability of a source file. Table 3.4 shows three different kinds of line splitting examples. We do not show any example of line merging, since it is the opposite action of line splitting. Function merging/splitting is the second change type in our taxonomy. While Merging is done for service consolidation or code clone elimination, it can be split also. The term Wrapping refers to a change where an old line is moved inside a block of code and we define the opposite action as unwrapping. Wrapping makes a line difficult to trace even when we try with contextual information since the context may differ and can be worse if the line itself changes significantly. In some cases, one form of wrapping can be changed into another. For example, the line attached with an *if* statement can be moved to the *else* part. We found scenarios where data structures used in previous versions are replaced with other kinds. For example, a HashTable can be replaced by a Map or an ArrayList implementation can be replaced by a LinkedList (see Table 3.4(4)). Changes of this kind may or may not preserve the textual similarity of lines and pose a threat to line mapping techniques. Changes in identifier names are common in source code evolution, particularly where copy-paste programming is involved. Location tracking techniques in general perform well when identifiers use descriptive names. Other language constructs such as function, method or class name can be changed. The last group of change type in our taxonomy is code reordering where functions or blocks of code can be reordered.

3.5.2 Experiment details

In the generation phase, we mutate source files. First, we select a source file in version v_i. We then make another copy of that source file. The mutation is carried out either by injecting or changing existing code fragments in the copied file in such a way that considers several possible change scenarios listed in our taxonomy of source code changes. To avoid bias in comparison, we did not consider mapping of comments or curly braces in our analysis since *SDiff* ignores mapping them.

[4] `https://netbeans.org/`

[5] `http://www.sourceforge.net/projects/itext`

Table 3.5: Results of mutation based evaluation

Method	Change in Data Structure	Wrapping	Line Split	Function Split	Renaming	Reordering
W_BESTI_LINE	18.8	86.7	4.3	93.4	17.9	82.7
LHDiff	**56.3**	**90**	**68.1**	**93.4**	**35**	**83.8**
Unix diff	6.3	80	2.1	55.7	0	13.5
Git	6.3	80	2.1	85.3	0	66.2
ldiff [-i 0]	12.3	80	40.1	55.7	42.5	13.5
ldiff [-i 1]	6.3	90	44.7	86.9	35	52.7
ldiff [-i 3]	6.3	90	40.7	95.1	35	78.4
ldiff [-i 5]	6.3	90	40.7	95.1	35	82.4
SDiff	81.3	80	51.1	10	10	82.4
SDiff-probable	93.8	80	51.1	10	10	82.4
SDiff-possible-probable	93.8	80	51.1	15	15	82.4
SDiff-min	87.5	80	51.1	15	15	82.4
SDiff-min-probable	93.8	80	51.1	15	15	82.4
SDiff-min-possible-probable	93.8	80	51.1	15	15	82.4
SDiff-token	87.5	80	51.1	15	15	82.4
SDiff-token-probable	93.8	80	51.1	15	15	82.4
SDiff-token-possible-probable	93.8	80	51.1	15	15	82.4

3.5.3 Results

The results of our mutation based analysis is summarized in Table 3.5. In most of the change groups, *LHDiff* either outperforms other techniques or performs very close to the best technique with an exception in API changes category. In that case, accuracy drops to 56.3%, whereas *SDiff* correctly detects around 93% mappings. When we investigate the reasons, it is found that both context and content changes significantly which leads to such poor result.

W_BESTI_LINE suffers from the problem of detecting boundary of language constructs (such as a statement) and thus is immune to line splitting. *SDiff* operates at the statement level and can detect line splitting, but the accuracy is variable. For example, cases where lines are added in between of the split lines (see Table 3.4(1), second example), *SDiff* cannot determine correct mappings. In our mutation based analysis, *LHDiff* performs not too bad in detecting line splitting. Manual investigation reveals that even in the case of line splitting, they do share some degree of similarity with the original line.

While *W_BESTI_LINE* and *ldiff* work at the line level, they can track movement of lines because of function splitting/merging. *SDiff* is affected by function or method splitting and cannot detect a block of lines that is moved to other functions. Thus, *SDiff* reports them as deleted. For instance, *SDiff* cannot track lines that are moved from function *sum* to *readFile* (see Table 3.4(2)) and reports them as deleted. The reason is that *SDiff* first tries to map functions and if it finds a mapping, it tries to map the lines. For this example, *SDiff* finds a mapping between sum_{old} and sum_{new} and then maps their lines without considering the fact that lines of the *sum* function can be moved to other functions also.

Renaming of variables, functions or classes also affect line tracking techniques and *ldiff*, *LHDiff* or *W_BESTI_LINE* may or may not map the line containing function declaration depending on the amount of changes occurred in the file. However, *SDiff* uses an origin analysis technique [67] to map functions across versions and thus can determine renaming of functions. In case of reordering, both *LHDiff* and *W_BESTI_LINE* show good performance. As expected, *Unix diff* cannot detect any reordering since reordering of lines causes *Unix diff* to report them as addition and deletion of lines. While *SDiff* shows 82.4% accuracy in detecting changes via reordering, *Unix diff* become the last.

3.6 Threats to Validity

There are a number of threats to the validity of this study. In this section we discuss them in brief.

First, the mapping of a line is subjective in nature and we cannot guarantee that there is no incorrect mapping in our benchmarks. However, we tried to minimize the number of false mapping as much as possible. We manually investigated all mappings including those found in the Reiss and Eclipse benchmarks. Cases where it was difficult to correctly map a line or there was a disagreement, the mapping was removed to avoid ambiguity. Second, considering the small sample size of the benchmarks one can argue that the sample may not represent the population and thus the performance observed in our study does not reflect the original scenario. However, finding changes of lines across versions and determining their correctness is a time consuming task. We carefully validated the correctness of existing benchmarks and developed a new benchmark containing different types of changes. We also used mutation based analysis to evaluate our technique with other approaches. Third, lines can be changed in various ways. In this study, we describe various kinds of changes that can affect the evolution of a line. Although we cannot guarantee that benchmarks contain all change types, we tried to minimize the effect of change types on their evaluation through collecting line change data at random. Finally, we penalize line tracking techniques with the same weight for detecting different false mapping types (spurious, change and eliminate). While one can argue about weighting false mapping types, we want to highlight to the fact that this is the same scoring scheme used in previous studies [111] and we followed the same strategy to make the result comparable with others.

3.7 Conclusion

We proposes a novel, language-independent line tracking approach called *LHDiff*. Our experiment shows that lines can effectively be tracked across versions with *LHDiff*. The experiment also shows that we can capture the context of a line by leveraging the locality of source code. We also observe that we can significantly accelerate context matching process using the simhash technique. We not only evaluate *LHDiff* with benchmarks created from real world applications but also use a mutation based analysis to evaluate it with

other line tracking techniques against different types of source code changes.

The results reveal that *LHDiff* is more effective than any other language-independent line tracking technique. We also compared *LHDiff* with *SDiff* which is a state-of-the-art language-dependent technique and found that in most cases *LHDiff* provides better result than *SDiff*. The mutation based analysis also enables us to explain the strength and weaknesses of line tracking techniques and can help a user to decide when to use which technique. Although *LHDiff* incorporates some features from *W_BESTI_LINE*, another simple line tracking technique developed by Reiss, but the potential of that technique is not fully explored in the early work, possibly because that work focused on comparing a large number of techniques and the benchmark used in that experiment contains small changes. *LHDiff* is reasonably fast (Comparable in speed to *SDiff* also), requires a small amount of memory, can easily be incorporated into source code control systems, and can be used with arbitrary text files.

For future study, we plan to identify the degree of structural knowledge required to reduce incorrect mappings. The current implementation of *LHDiff* only uses information available within files and we would like to explore whether information stored within source code control systems can assists us mapping lines. While this work focuses on tracking lines, visualizing this information poses another challenge that we also want to address. The code of *LHDiff*, data files used in this experiment, and complete evaluation results can be found online.[6]

[6] `https://muhammad-asaduzzaman.com/research/`

Chapter 4

A Simple, Efficient, Context Sensitive Approach for Code Completion

The previous chapter described a technique to track source code lines across versions. The study showed that one way to capture the context of a source code line is by considering only those tokens that appear within the top-4 lines. It also showed the benefits of using the simhash technique that can accelerate the line matching process. This chapter uses both in improving the performance of method call completion techniques. The study results, including the implementation of a method completion technique, have been published in a leading software conference [7, 6] and in a journal [8].

4.1 Introduction

Developers rely on frameworks and libraries of APIs to ease application development. While APIs provide ready-made solutions to complex problems, developers need to learn to use them effectively. The problem is that due to the large number of APIs, it is practically impossible to learn and remember them completely. To avoid developers having to remember every detail, modern integrated development environments provide a feature called Code Completion, which displays a sorted list of completion proposals in a popup menu for a developer to navigate and select. In a study on the Eclipse IDE, Murphy *et al.* [90] found that code completion is one of the top ten commands used by developers, indicating that the feature is crucial for today's development. In this chapter, we focus our attention on method and field completion since these are the most frequently used forms of code completion [114] (other forms of code completion include word completion, method parameter completion, and statement completion). In the remainder of this chapter, we use the term Code Completion to refer to method and field completion unless otherwise stated.

Existing code completion techniques can be divided into two broad categories. The first category uses mainly static type information, combined with various heuristics, to determine the target method call, but does not consider previous code examples or the context of a method call. A popular example is the default code completion system available in Eclipse, which utilizes a static type system to recommend method calls. It sorts the completion proposals either alphabetically or by relevance before displaying them to users in a popup menu. Hou and Pletcher [47] developed another technique, called Better Code Completion (BCC), that uses a combination of sorting, filtering and grouping of APIs to improve the performance of the default

```
String line;
StringBuilder sb = new StringBuilder();
br= new BufferedReader(new
          FileReader("file.txt"));
try{
while ((line = br.readLine()) != null){
    sb . append ( l i n e ) ;
    sb . append ( '\n ' ) ;
}
}finally {br . close ();}
```

Figure 4.1: An example of reading a file where readLine is the target of code completion

type-based code completion system of Eclipse. The second category of techniques takes into account previous code examples and usage context matching to recommend target method calls [93, 18, 45]. For example, to make recommendations, the Best Matching Neighbor (BMN) [18] code completion system matches the current code completion context to previous code examples using the k-Nearest Neighbour (kNN) algorithm.

BMN has successfully demonstrated that the performance of method call completion can be improved by utilizing the context of a target API method call. BMN focuses on using a special kind of context for a given call site, i.e., the list of methods that have been invoked on the same receiver variable plus the enclosing method of the call site. however, there are many other possible forms of context to be considered. As an example of other forms of context, consider the code shown in Figure 4.1, where a file is read via the `BufferedReader` object `br` in a `while` loop. In fact, the `readLine` method is commonly called as part of a `while` loop's condition located inside a try-catch block. Within a few lines of distance of the `readLine` method, developers usually create various objects related with that method call. For example, developers typically create a `BufferedReader` object from an instance of `FileReader` and later use that object to call the `readLine` method. Therefore, in addition to the methods that were previously called on the receiver object `br`, keywords (such as `while`, `try`, `new`), other methods (such as `FileReader`, `BufferedReader` constructor name) can be considered as part of the context of `readLine` as well. Adding these extra pieces of information can enrich the context of the targeted call to help recommend methods that are more relevant (`readLine`, in this case).

In this study, we explore the performance implications of these additional forms of context for code completion. To this end, we first propose a context sensitive code completion technique, called CSCC (Context Sensitive Code Completion), that leverages code examples collected from repositories to extract method contexts to support code completion. Given a method call, we capture as its context *any method names, Java keywords, class or interface names* that appear within four lines of code. In this way, we build a database of context-method pairs as potential matching candidates. We use tokenization rather than parsing and advanced analysis to collect the context data, so our technique is simple. When completing code, given the receiver object, we use its type name and context to search for method calls whose contexts match with

that of the receiver object. To scale up the search, we use *simhash* technique [24] to quickly eliminate the majority of non-matching candidates. This allows us to further refine the remaining much smaller set of matching candidates using more computationally expensive textual distance measures. This also makes our technique efficient and scalable. We sort the matching candidates using similarity scores, and recommend the top candidates to complete the method call.

We compare our context-sensitive code completion technique with five other state-of-the-art techniques [18, 48] using eight open source software systems. Results from the evaluation suggest that our proposed technique outperforms all five state-of-the-art type or example-based systems that we have compared. Moreover, to understand how exactly the context of a method call affects code completion, we propose a taxonomy for the contextual elements of a method call and compare how different techniques perform for each category of contextual elements (see Section 4.4.3). This experiment helps us clearly understand the strengths and limitations of CSCC and other existing techniques. We also conduct a set of experiments on CSCC to uncover the effect of context length, impact of different context information, effectiveness in cross-project prediction, performance for different frameworks or libraries, and evaluate building context using the four lines following a method call.

After releasing CSCC as an Eclipse plugin, we received a number of requests to extend support for fields. During our investigation, we found that the context CSCC uses for method call completion can easily capture the context of field access too. Evaluation on field completions using eight different subject systems and with four state-of-the-art code completion techniques revealed that CSCC is able to outperform all four techniques by a substantial margin.

Recently, a number of techniques have been developed that uses statistical language models for predicting the next token or completing API elements [45, 96, 93, 137]. They use different information sources to capture the context. These include token sequences, caching, API usage graphs, or a combination of them. We are interested on how well CSCC performs compared to those techniques. Toward this goal, we compare CSCC with four other statistical language model techniques and present the study results in this chapter.

We make the following contributions:

(1) A technique called CSCC to support code completion using a new kind of context and previous code examples as a knowledge-base,

(2) A quantitative comparison of the proposed technique CSCC with five existing state-of-the-art tools that shows the effectiveness of our proposed technique,

(3) A taxonomy of method call context elements and an experiment that helps to identify strengths and limitations of CSCC and other existing techniques (see Section 4.4.3 for details of the taxonomy),

(4) A set of studies that helps to understand different aspects of the technique (see Section 4.5),

(5) Extending CSCC for field completion and a quantitative comparison with four other state-of-the-art tools, and

(6) A comparison of CSCC with four state-of-the-art statistical language model code completion techniques.

The remainder of the chapter is organized as follows. Section 4.2 briefly describes related work. Section 4.3 describes our proposed technique CSCC. Section 4.4 compares CSCC with various code completion techniques using eight open source software systems for completing API method calls. We conduct a set of studies to uncover different aspects of the technique and present the results in Section 4.5. Section 4.6 describes the extension of the technique to support field completion including evaluation with four other state-of-the-art tools. We compare CSCC with statistical language model based code completion techniques in Section 4.7. Section 4.8 summarizes the threats to validity. Finally, Section 4.9 concludes the chapter.

4.2 Related Work

Bruch *et al.* [18] propose the Best Matching Neighbors (BMN) completion system that uses the k-nearest neighbor algorithm to recommend method calls for a particular receiver object. The most fundamental difference between BMN and CSCC lies in their definition of context. Our definition of a method call context includes any method names, keywords, class or interface names within the four lines prior to a method call, whereas BMN's context is made of the set of methods that have been called on the receiver variable plus the enclosing method. Due to this difference, BMN and CSCC use different techniques to calculate similarities and distances. Lastly, BMN uses frequency of method calls to rank completion proposals, whereas CSCC ranks them based on distance measures.

Hou and Pletcher [48, 47] propose a code completion technique that uses a combination of sorting, filtering and grouping of APIs. They implement the technique in a research prototype called Better Code Completion (BCC). BCC can sort completion proposals based on the type-hierarchy or frequency count of method calls. It can filter non-API public methods. BCC also allows developers to manually specify a set of methods that are logically related, and thus belong to the same group and appear together while displaying completion proposals in a popup menu. However, BCC does not leverage previous code examples. Moreover, BCC requires the filters to be manually specified which can only be performed by expert users of code libraries and frameworks. However, because CSCC considers the usage context of method calls to recommend completion proposals, methods that are not appropriate to call in a particular context would be automatically filtered out. So CSCC would require less effort to use than BCC.

Another important work is the GraLan, a graph-based statistical language model that targets completing API elements [93]. The term API element refers to method calls, field accesses and control units (such as for, while, if etc.) used in the API usage examples. The technique mines the source code to generate API usage graphs, called Groums. Given an editing location, the technique determines the API usage subgraphs surrounding the current location and use them as context. GraLan then computes the probability of extending each context graph with an additional node. These additional nodes are collected, ranked and recommended for code completion. CSCC differs from GraLan in terms of context definition, recommendation

formulation and ranking strategy. For example, GraLan ranks completion proposals based on their probability distributions, but CSCC uses textual distance measures.

Besides GraLan, a number of code completion systems have been proposed that use statistical language models. Hindle *et al.* develop a code suggestion engine that uses the widely adopted N-gram model to recommend the next token [45]. Tu *et al.* develop a language model (known as Cache LM) that uses a cache component to capture the localness of software [137]. Christine *et al.* develop an Eclipse plugin, called CACHECA, that combines the native suggestions made by the Eclipse IDE with that of the cache language model [36]. Nguyen *et al.* propose a statistical semantic language model for source code, called SLAMC [96]. The model incorporates the semantic information of code tokens. It also captures global concerns using a topic model and pairwise association of language elements. The model has been used to develop a code suggestion engine to predict the next token. SLANG collects sequence of method calls to create a statistical language model [110]. When a developer requests for a code completion, the tool completes the editing location using the highest ranked sequence of method calls computed by the language model. While almost all of these techniques work at the lexical level, CSCC and SLANG work at the API level. CSCC differs from SLANG in context formulation. The technique not only collects method calls but also collects keywords and type names.

Nguyen *et al.* [94, 92] use a graph-based algorithm to develop a context sensitive code completion technique, called GraPacc. The technique mines API usage graphs or Groums to capture API usage patterns in open source code bases. This creates an API usage database. During the development phase, the technique extracts context sensitive features and matches them with usage patterns in the database. It then recommends a list of matched patterns to complete the remaining code. Although both GraPacc and CSCC utilize code context to make recommendations, the goals and approaches are different. GraPacc recommends multiple statements at a time, but CSCC completes a single method call. Similar to GraLan, GraPacc also leverages Groums for identifying code context. Since the objective of CSCC is similar to GraLan and we already include that in our study, we did not compare with GraPacc.

Robbes and Lanza [114] propose a set of approaches to support code completion that use program history to recommend completion proposals. They define a benchmark to measure the usefulness of a code completion system and evaluate eight different code completion algorithms. They found that the typed optimist completion technique provides better results than any other techniques, since it merges the benefits of two different techniques. The program change history can be considered the *temporal* context for a method call, whereas ours is the *spatial* context. While their technique requires a change-based software repository to collect program history, our technique can work with any repository.

Research related with recommending source code examples is also related to our study because of their use of context. Among various work on code examples recommendation, the most relevant work to ours is that of Holmes and Murphy [46]. They use the notion of structural context to recommend source code examples. The context in their case contains information about inheritance, method calls and types declared or used

in a method. Although our method call usage context shares some similarity with theirs, the objectives are completely different.

There are also a number of other techniques or tools that make use of previous code examples, but their goals are different than ours. For example, Calcite helps developers to correctly instantiate a class or interface using existing code examples [87]. While Calcite helps instantiate a class, we help developers complete method calls. Precise mines existing code bases to recommend appropriate parameters for method calls [144]. Hill and Rideout [44] focus on automatic completion of a method body by searching similar code fragments or code clones in a code-base. Lee *et al.* [74] introduce a technique that identify changes of program entities (such as method names) during the evolution of a software system. When a developer requests for a completion, the technique presents the program entities along with their changes through code completion. It also helps developers navigate to the past code examples to see the changes. Jacobellis *et al.* [54] leverage code completion to automate the edit operations of source code from user specified custom, reusable template of code changes.

Keyword programming [78] is also related to our study, but it defines a completely different way of user interaction for code completion. Instead of typing a method name, users type some keywords that give hints about the method the user is trying to call. The algorithm then automatically completes the method call or makes appropriate suggestions to complete the remaining part. Han and Miller [40] later introduce abbreviation completion that uses a non-predefined set of inputs to complete the target method call.

Previously discussed techniques make use of floating menus to present completion proposals and none focuses on the improvement of the user interface. Omar *et al.* [103] present an approach that allows library developers to integrate specialized interfaces, called palettes, directly into the editor. The benefit of the approach is that developers do not need to write the code explicitly; rather, the specialized interface allows users to provide required input and generate the appropriate code. They developed a tool, named Graphite that allows Java developers to write a regular expression in a palette and found that the addition of such a specialized interface is helpful to professional developers. Instead of focusing on a specialized interface for code completion, we focus on predicting target method calls as a basis for suggesting completion proposals. However, palettes can complement our technique to complete method parameters, which remains as a future work.

4.3 Proposed Algorithm

In this section, we describe our algorithm for finding method calls to recommend for a target object. Figure 4.2 presents an overview of the process. Our example-based, context-sensitive code completion system works in three steps:

- Collect the usage context of API method calls from code examples and index them by their contexts to support quick access. We model the context of an API method call by method calls, Java keywords

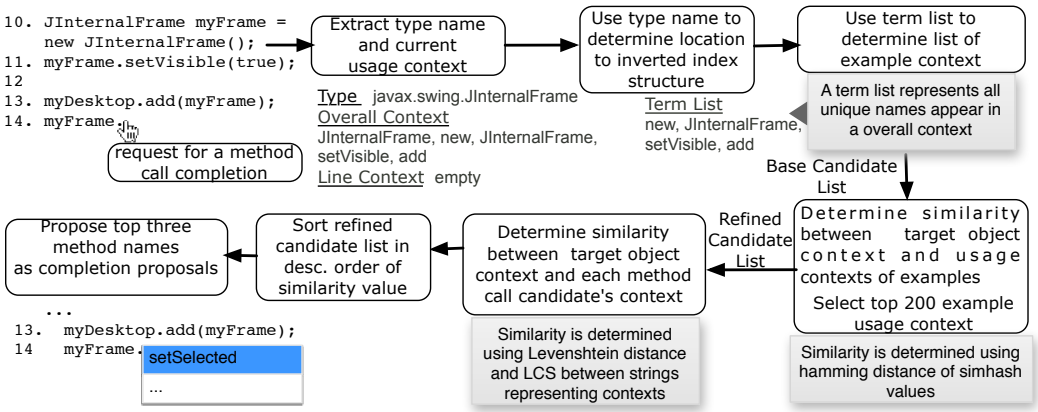


Figure 4.2: An overview of CSCC's recommendation process starting from a code completion request.

and type names that appear within the four lines prior to the receiver object that called the method. We hypothesize that these elements around the target method call can provide a better, fuller context than other approaches [18, 48].

- Search for method calls whose context matches with that of the target object. One approach would be to measure the similarity between the context of the target object and that of each method call in the example code base directly using string edit-distances. However, string edit-distance operations are computationally expensive. To speed up the search, we instead use the Hamming distance over the *simhash* values as similarity measures. We determine a smaller list of method names that are the more likely candidates for code completion, which we refer to as the *candidate list*.

- Synthesize the method calls from the candidate list. For each method name in the candidate list, we use a combination of token-based Longest Common Subsequence (LCS) and Levenshtein distance to determine a similarity value of its context with that of the receiver object. We then sort the method names in descending order of similarity value and recommend the top-3 names to complete the method call.

We describe the three steps in detail as follows.

4.3.1 Collect usage context of method calls

In this step, CSCC mines code examples to find the usage context of API method calls. To capture a method call context, we consider the content of the n lines prior to it, including the line where the target method call appears. In this study, we use $n = 4$ and we validate this decision in Section 4.5.

We collect the following three kinds of information from the four lines of context, which we refer to as the *overall context* of the method call:

(1) Any method names,

(2) Any Java keywords except access specifiers, and

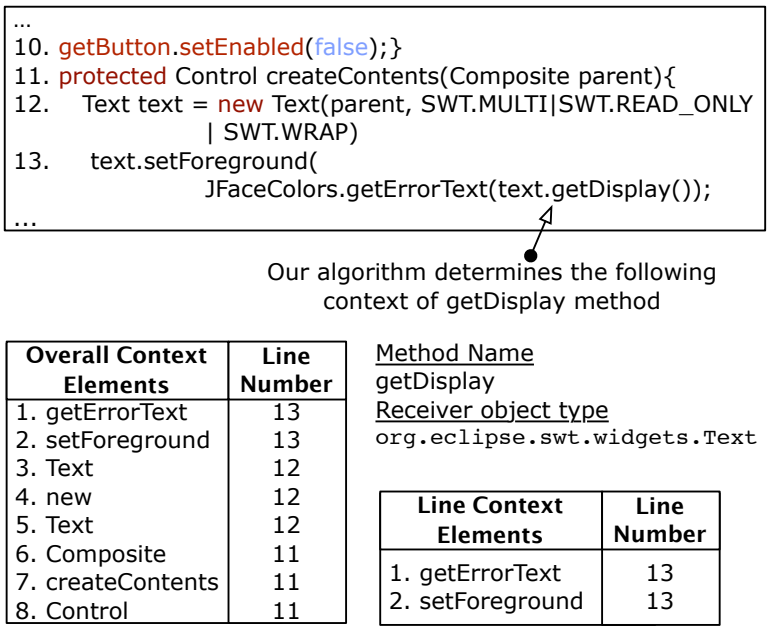


Figure 4.3: Overall context and line context for *getDisplay*.

(3) Any class or interface names.

When extracting the overall context, we ignore blank lines, comment lines, or lines containing only curly braces. We also remove any duplicate tokens from the overall context.

In addition, we separately collect a *line context* for the target method, which includes any method names, keywords (except access specifiers), class or interface names and assignment operators that appear on the same line but before the target method call. When the overall contexts are completely different and fail to match, line contexts act as a secondary criterion for matching.

To further explain the construction of both overall and line context, consider the method call at line 13 as shown in Figure 4.3. The contents of both contexts include tokens and their locations. Note that although line number 10 is within four lines of our target method call *getDisplay*, it is not considered part of the context as it is located outside of the *createContents* method containing the target call *getDisplay*.

We use a two-level indexing scheme to organize the collected usage contexts of method calls (see Figure 4.4 for an example of it). The type name of a receiver object is used to group all method calls that have been invoked on the type. We use an inverted index structure[1] to organize such a group of method calls. More specifically, an inverted index is a data structure that maps each term to its location in a document. We represent each overall context of a method call as a document, and use tokens from the context to index the set of documents where they appear.

[1] http://www-nlp.stanford.edu/IR-book/

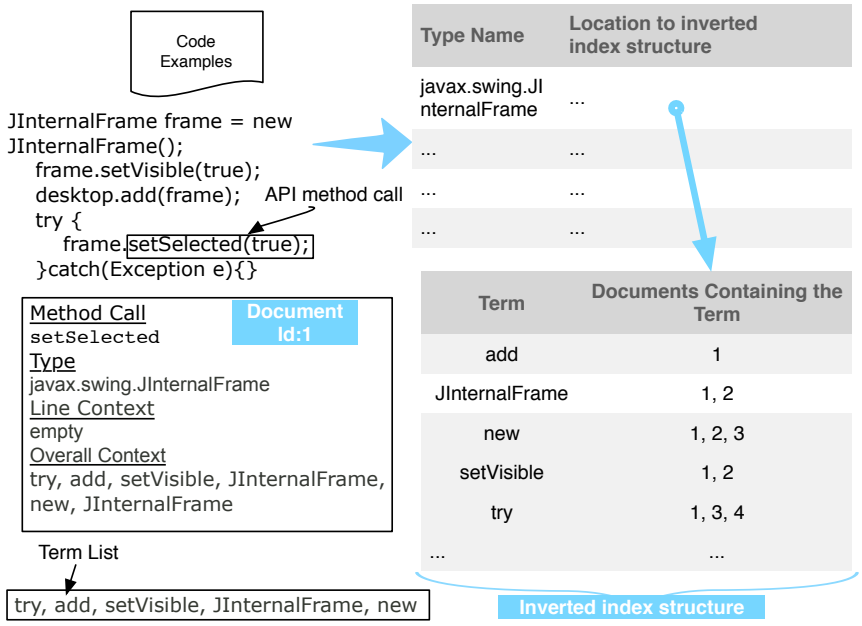


Figure 4.4: Database of method call contexts are grouped by receiver types using an inverted index structure. The figure on the left side shows the collected context information for an API method call. The figure on the right side shows how we use the information to build an inverted index structure.

4.3.2 Determine candidates for code completion

When a user requests a method call completion (for example, in Eclipse typing a dot (.) after an object name initiates such a request), the algorithm first extracts both overall and line contexts for the receiver object. To find candidate methods for code completion, we match the current context to those extracted from the example code base. Specifically, we use the type name of the receiver object as an index to determine the related inverted index structure, which contains all method calls made on the receiver type (Figure 4.4). We then use tokens from the overall context as keys to the inverted index structure to collect all those method calls in the code examples that have the same type as the receiver object. We refer to these matching method calls as the *base candidate list*.

The *base candidate list* often contains thousands of method calls, so we need to reduce them to a small number of most likely candidates in order to recommend. We follow a two-step process to search for the most likely candidates. We first use the *simhash* technique to determine a short list of method names (currently the top 200 that are deemed most similar to the target context) that are more likely to complete the current method call and quickly eliminate the majority of others. To calculate string similarity metrics, we concatenate all the tokens of each context and generate a *simhash* value for the concatenated string (see Figure 4.5 for an example). We use the *simhash* technique to eliminate most of the irrelevant matching candidates because it is both fast and scalable. Second, we use the normalized Longest Common Subsequence (LCS) and Levenshtein distances to measure the fine-grained similarity between the target context and the context of each likely matching candidate to obtain a refined candidate list. Calculating LCS and Levenshtein

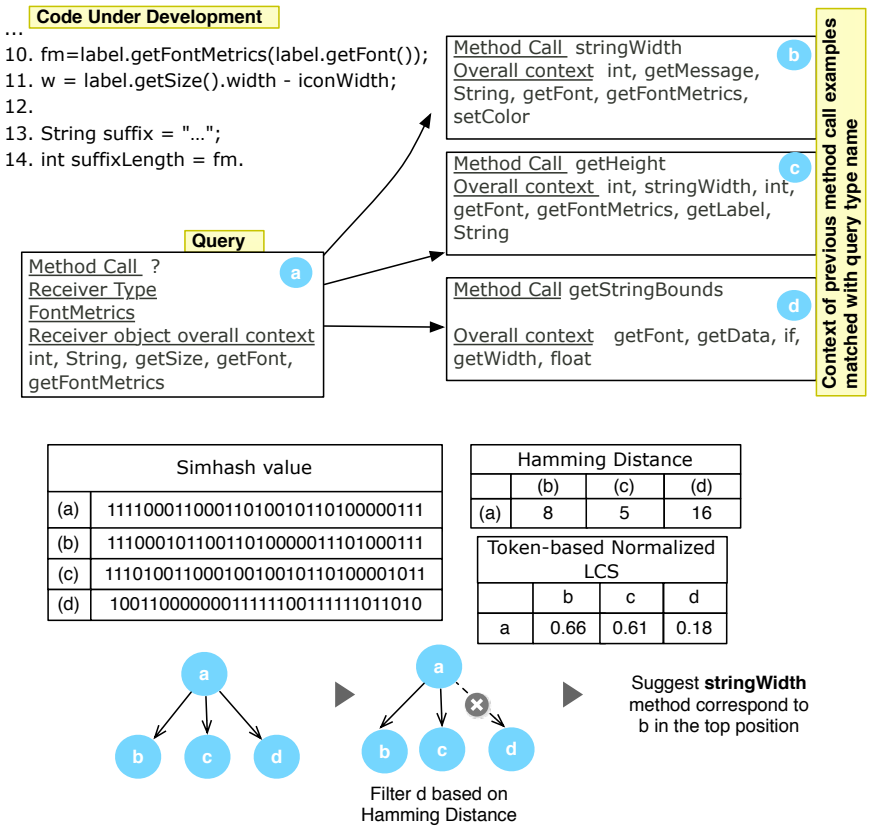


Figure 4.5: Context similarities are measured using the Hamming distance between simhash values.

distances are both time consuming operations. Although they provide fine-grained similarity measures, due to the near real-time constraint needed for practical code completion, we cannot apply them directly on the *base candidate list*.

We use the *simhash* technique [24] to identify the most likely method call candidates. *Simhash* uses a cryptographic hash function to generate binary hash keys, also known as *simhash* values. An important property of simhash is that strings that are similar to each other have either identical or very similar *simhash* values. Therefore, we determine the similarity between each pair of contexts using the Hamming distance of their corresponding *simhash* values. We use the Hamming distance of the overall context to sort the matching candidates unless the Hamming distance of the line context exceeds a predefined threshold value, in which case we use the Hamming distance of the line context as the distance measure. After sorting by similarity, we take the top-k method contexts as the likely matching candidates of the target context. After experimentation with different values of k, we found that $k = 200$ is a good choice to work with and we use that value in our study. We refer to this list as the *refined candidate list*.

To recommend method calls, we further sort the method names in the refined candidate list by combining both overall and line context similarities as follows. We use the normalized Longest Common Subsequence (LCS) distance to measure the similarity of the token sequences from the overall context. We use Levenshtein distance to measure the similarity of the token sequences from the line context. We sort matching candidates

in descending order of their overall context similarity. However, in the case of a tie for the overall context similarity, we use the line context similarity. We ignore all matching candidates whose similarity values drop below a certain threshold. We empirically found that 0.30 is a good choice to work with.

The *simhash* technique has been found effective for detecting similar pages in a large collection of web documents [80] and also has been used successfully in detecting similar code fragments in code clone detection [138]. Although various hash functions are available, we use the *Jenkin* hash function since it has been found effective in a previous study [138]. We generate a 64 bit *simhash* value for both overall and line contexts of the target object. To save computation time, we precompute the *simhash* values. We determine the similarity between each pair of contexts using the Hamming distance of the corresponding *simhash* values.

4.3.3 Recommend top-3 method calls

The objective of this step is to recommend a list of completion proposals (i.e., method names). Since there are many code examples associated with the same method call, the sorted list of method names obtained from the previous step may contain many duplicates. After eliminating duplicates, we present the top-3 method names to the users.

4.4 Evaluation

We evaluate our technique and compare CSCC with five state-of-the-art code completion systems using eight open-source systems (Table 4.1). For a given subject system, we determine all locations where methods from a target API have been called. This set of method calls constitutes our data set. We then apply the ten-fold cross validation technique [18] to measure the performance of each algorithm. This is a popular way of measuring performance of information retrieval systems [20] and has been used previously in many research projects. First, for each system we divide the data set into ten different folds, each containing an equal number of method calls. Next, for each fold, we use code examples from the nine other folds to train the technique for method call completion. The remaining fold is used to test the performance of the technique.

We use recall to measure how many cases a technique produces relevant (correct) recommendations out of the total test cases. Clearly a technique that recommends all possible methods would always achieve a great recall of 1. That is why we consider the precision measure. It is defined as the ratio between the number of times a technique produces relevant recommendations and the number of times that technique produces any recommendations. The higher a relevant recommendation is in the list of recommendations, the better. That is why we collect precision measure at top-1, top-3 and top-10 recommendations. Finally, we use the F-Measure, a widely accepted measure, to correlate precision and recall by computing their harmonic means.

$$Recall = \frac{recommendations\ made \cap relevant}{recommendations\ requested} \tag{4.1}$$

$$Precision = \frac{recommendations\ made \cap relevant}{recommendations\ made} \tag{4.2}$$

$$F\text{-}measure = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \tag{4.3}$$

where *recommendations requested* is the number of method calls in our test data for which we will make a code completion request. *Recommendations made* is the number of times where a code completion system makes a recommendation.

4.4.1 Test systems

We chose to focus on two API's, SWT and Swing/AWT, as the target for our evaluation. These are popular libraries extensively used for developing GUI applications. We selected four systems that used SWT. The largest one is Eclipse 3.5.2,[2] a popular open source IDE. Vuze[3] is a P2P file sharing client using the bittorrent protocol. Subversive[4] provides support to work with Subversion directly from Eclipse. RSSOwl[5] is an RSS newsreader. We also chose four open source software systems for AWT/Swing. NetBeans 7.3.1,[6] the largest, is an IDE; jEdit[7] is a text editor; ArgoUML[8] is a UML modeling tool; and JFreeChart[9] is a Java charting library.

4.4.2 Evaluation results

In this section, we discuss the results of evaluating and comparing CSCC with five other code completion systems (ECCAlpha, ECCRelevance, FCC (Frequency-based Code Completion), BCC, and BMN) using the eight test systems. We have introduced BCC and BMN earlier. ECCAlpha and ECCRelevance are two default Eclipse code completion systems that leverage the static type system. ECCAlpha sorts the completion proposals in alphabetical order, and ECCRelevance uses a positive integer value, called relevance, to sort them. The value is calculated based on the expected type of the expression as well as the types in the code context (such as return types, cast types, variable types etc.). The Frequency-based code completion system (FCC) considers the frequency of method calls in a model to make recommendations. The more frequent a method occurs, the higher its position is in the completion proposals.

Table 4.1 shows the precision, recall, and F-measure values for the six code completion systems. The top four rows are results collected for SWT, and the bottom four rows for AWT/Swing. Overall, CSCC achieves higher precision and recall values than any of the other techniques for both single and top-3 completion proposals.

[2] http://www.eclipse.org/
[3] http://www.vuze.com/
[4] https://www.eclipse.org/subversive/
[5] http://www.rssowl.org/
[6] https://netbeans.org/
[7] http://sourceforge.net/projects/jedit/
[8] http://argouml.tigris.org/
[9] http://sourceforge.net/projects/jfreechart/

Table 4.1: Evaluation results of code completion systems. *Delta* shows the improvement of CSCC over BMN.

Subject Systems		Precision							Recall							F-Measure						
		ECCAlpha	ECCRel	BCC	FCC	BMN	CSCC	Delta (%)	ECCAlpha	ECCRel	BCC	FCC	BMN	CSCC	Delta (%)	ECCAlpha	ECCRel	BCC	FCC	BMN	CSCC	Delta (%)
Eclipse	Top-1	0.006	0.01	0.24	0.31	0.36	0.60	24	1	1	1	1	0.77	0.99	22	0.008	0.020	0.39	0.47	0.49	0.75	26
	Top-3	0.052	0.20	0.52	0.49	0.63	0.80	17	1	1	1	1	0.77	0.99	22	0.10	0.34	0.68	0.66	0.69	0.88	19
	Top-10	0.14	0.34	0.80	0.73	0.69	0.90	21	1	1	1	1	0.77	0.99	22	0.25	0.51	0.89	0.84	0.73	0.94	21
Vuze	Top-1	0.005	0.10	0.23	0.33	0.35	0.56	21	1	1	1	1	0.76	0.98	22	0.009	0.18	0.37	0.50	0.48	0.71	23
	Top-3	0.03	0.16	0.49	0.49	0.59	0.73	14	1	1	1	1	0.76	0.98	22	0.06	0.28	0.66	0.66	0.66	0.84	18
	Top-10	0.20	0.36	0.94	0.71	0.61	0.83	22	1	1	1	1	0.76	0.98	22	0.33	0.53	0.97	0.83	0.68	0.90	22
Subversive	Top-1	0.01	0.03	0.30	0.36	0.58	0.68	10	1	1	1	1	0.38	0.97	60	0.02	0.058	0.46	0.53	0.46	0.80	34
	Top-3	0.02	0.07	0.63	0.62	0.77	0.86	9	1	1	1	1	0.38	0.97	60	0.04	0.13	0.77	0.77	0.51	0.91	40
	Top-10	0.07	0.20	0.96	0.88	0.79	0.91	12	1	1	1	1	0.38	0.97	60	0.13	0.34	0.98	0.94	0.51	0.94	43
Rsowl	Top-1	0.01	0.078	0.25	0.32	0.48	0.65	17	1	1	1	1	0.72	0.98	26	0.20	0.14	0.40	0.48	0.58	0.78	20
	Top-3	0.024	0.16	0.58	0.51	0.74	0.84	10	1	1	1	1	0.72	0.98	26	0.046	0.28	0.73	0.68	0.73	0.90	17
	Top-10	0.077	0.29	0.85	0.74	0.80	0.90	10	1	1	1	1	0.72	0.98	26	0.14	0.45	0.92	0.85	0.76	0.94	18
NetBeans	Top-1	0.12	0.13	0.34	0.29	0.43	0.66	23	1	1	1	1	0.67	0.98	31	0.21	0.23	0.51	0.45	0.52	0.79	27
	Top-3	0.18	0.25	0.62	0.53	0.67	0.86	19	1	1	1	1	0.67	0.98	31	0.31	0.40	0.77	0.69	0.67	0.92	25
	Top-10	0.36	0.48	0.86	0.73	0.70	0.92	22	1	1	1	1	0.67	0.98	31	0.70	0.65	0.92	0.84	0.68	0.95	27
JEdit	Top-1	0.009	0.12	0.41	0.35	0.52	0.62	10	1	1	1	0.98	0.70	0.94	24	0.02	0.21	0.58	0.52	0.60	0.75	15
	Top-3	0.14	0.29	0.62	0.53	0.74	0.79	5	1	1	1	0.98	0.70	0.94	24	0.25	0.45	0.77	0.69	0.72	0.86	14
	Top-10	0.32	0.49	0.83	0.74	0.79	0.85	6	1	1	1	0.98	0.70	0.94	24	0.48	0.66	0.91	0.84	0.74	0.89	15
ArgoUML	Top-1	0.03	0.13	0.40	0.32	0.46	0.58	12	1	1	1	0.99	0.68	0.95	27	0.058	0.23	0.57	0.48	0.55	0.72	17
	Top-3	0.12	0.27	0.65	0.53	0.68	0.74	6	1	1	1	0.99	0.68	0.95	27	0.21	0.43	0.79	0.69	0.68	0.83	15
	Top-10	0.27	0.47	0.83	0.78	0.74	0.81	7	1	1	1	0.99	0.68	0.95	27	0.43	0.64	0.91	0.87	0.71	0.87	16
JFreeChart	Top-1	0.02	0.08	0.35	0.32	0.42	0.63	21	1	1	1	1	0.75	0.98	23	0.040	0.15	0.52	0.48	0.54	0.77	23
	Top-3	0.05	0.19	0.52	0.63	0.76	0.85	9	1	1	1	1	0.75	0.98	23	0.10	0.32	0.68	0.77	0.75	0.91	16
	Top-10	0.27	0.58	0.63	0.92	0.84	0.94	10	1	1	1	1	0.75	0.98	23	0.43	0.73	0.77	0.96	0.79	0.96	17

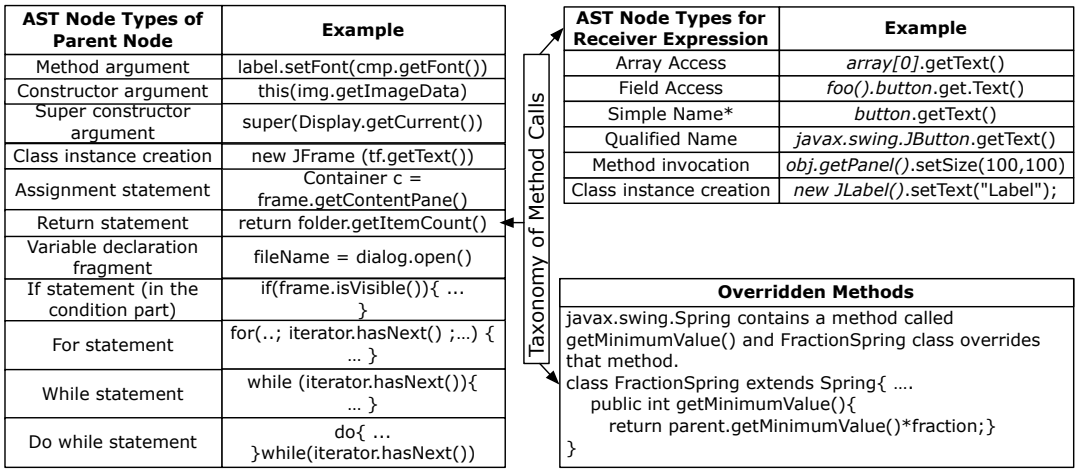


Figure 4.6: Taxonomy of method calls

For the top-3 proposals, it has precision of 73-86% and recall of 97-99%. The recalls for ECCAlpha, ECCRelevance and BCC are all one. Both ECCAlpha and ECCRelevance performed poorly, and BCC outperformed both of them.

Except in a few cases, BCC also outperforms FCC. Furthermore, the performance of FCC is not great, while its recall is close to 100%, its precision is only 49-62% for the top-3 proposals. Interestingly, BMN did not perform well either. Although its precision is better than both BCC and FCC for the single and top-3 suggestions, its recall is poorer in both cases. This is due to the fact that BMN targets local variable method calls but there are many places in source code where methods are called on fields, parameters, chained expressions, or even static types. We performed further experiments to elaborate on this issue in Section 4.4.3.

The results for AWT/Swing shown in the bottom four rows of Table 4.1 are consistent with those of SWT. For example, for the top-3 proposals and for the largest subject system (NetBeans), the F-measure of CSCC is higher by 15% compared to the closest performing technique.

To test whether CSCC performed significantly better than other techniques, we also performed directional Wilcoxon Signed Rank Tests for the top-3 completion proposals between CSCC and other techniques. The null hypothesis is that there is no difference in precision and recall values for the top-3 completion proposals. The test shows that the difference in precision and recall values between CSCC and other techniques are statistically significant at the p value of 0.05.

4.4.3 Evaluation using a taxonomy of method calls

While the evaluation in Section 4.4.2 provides a ranking of the six techniques in terms of their performance, it does not reveal what factors contribute to CSCC's better performance. We hypothesize that it is due to CSCC's ability to capture a fuller context for method calls. To further shed light on this hypothesis, we propose a taxonomy for the characteristics of a method context, and compare the techniques using each

Table 4.2: Categorization of method calls for different AST node types for receiver expressions (for the top-3 proposals)

Expression Types		NetBeans			Eclipse		
		Quantity (%)	Correctly Predicted BMN (%)	Correctly Predicted CSCC (%)	Quantity (%)	Correctly Predicted BMN (%)	Correctly Predicted CSCC (%)
Array Access		0.54	0	54.16	1.65	0	79.73
Class Instance Creation		0.18	0	100	0.11	0	100
Field Access		0.60	0	80.77	0.49	0	77.27
Method Invocation		21.66	0	94	11.88	0	75.42
Simple Name	Type	5.48	79.21	92.13	3.02	96.26	98.26
	Local Variable	32.13	68.26	84.85	41.86	0.64	83.14
	Field	51.94	52.07	79.95	46.92	50	75.46
	Parameter	10.44	52.80	79.35	9.47	46.81	79.50
	Total	74.08	58.84	82.13	85.02	56.86	79.65
Qualified Name		2.94	57.36	82.17	0.85	60.53	71.05

category of method call characteristics within the taxonomy.

Our taxonomy (Figure 4.6) includes three categories of characteristics for the context of a target method call: the AST node types for its receiver expression, the AST node types for its parent node, and the enclosing overridden method that contains the target method call. Although we cannot guarantee that the taxonomy covers every possible aspect of method call completions, it can provide insights into code completion techniques and can also help us to decide where more effort is needed.

We use the following procedure for our evaluation. For each category of method call within each test fold of a subject system, we count how many of them are correctly predicted by a code completion technique. For each system, we then present the final result after adding the numbers for all ten test folds.

AST Node Type for Receiver Expression

We categorize the receiver expressions of the test method calls according to their AST node types. We count the number of test method calls that each code completion technique correctly recommends for each kind of AST node. Table 4.2 shows the results for the top-3 proposals from the two largest test systems, Eclipse and NetBeans.

Table 4.2 suggests that the majority of receiver expressions fall in the simple name category. A simple name can be a variable name (declared as a method parameter, a local variable, or a field) or a type name (static method calls). The original BMN technique considers only local variables and their types to compute

completion proposals. However, Table 4.2 shows that many receiver expressions of method calls are not local variables, and thus for which BMN produces no recommendations. This explains why we did not receive good results for BMN (Table 4.1). The way BMN collects usage context is quite limited. In contrast, CSCC can identify usage context even when the receiver is not a local variable and thus can recommend method names for those cases too.

We performed another experiment where we train and test both BMN and CSCC using only those method calls where the receiver is a local variable. For the top-3 proposals, BMN achieves 68% recall and 77% precision for the Eclipse system, both of which are higher than those of any other techniques except CSCC. The recall and precision for CSCC are 84% and 86%, respectively, indicating that CSCC performs better than BMN even when the receivers are local variables.

AST Node Types of Parent Node

We consider those method call expressions where a particular type of object or value is expected. For example, a framework method call can be located in the condition part of an if statement that expects a boolean value. A method call can be located in the right hand side of an assignment expression. If the left hand side of that assignment expression is of Container type, the right hand side should return an object of type Container or a sub-type of it. The goal is to identify how well techniques that consider type information as context perform in these cases compared to others. To make the result comparable with the BMN code completion system, we consider those method calls where the receiver is a local variable.

Table 4.3 shows the results of top-3 proposals for the Eclipse system. We can see that considering the expected type as contextual information can help improve code completion techniques. That is why the accuracy of BCC becomes close to that of CSCC, which achieves the highest accuracy for all but one category of AST node. Other code completion systems, such as BMN, FCC and default code completion systems of Eclipse, did not perform well in this experiment.

Overridden Methods

The objective is to verify whether methods called from within overridden methods impose any challenge to the evaluated code completion techniques. Our informal observation is that it may be difficult to identify usage context for method calls within overridden methods. When we manually analyze some of the code examples, we notice that method calls within overridden methods may contain very limited contextual information. For example, a number of methods in Java Swing applications result from implementing the ActionListener interface and those methods contain only a few methods called on the receiver objects. This can affect the performance of those techniques that leverage receiver method calls for making recommendations. Similar to the previous experiment we test only those method calls where the receiver is a local variable. CSCC again performs better than any other techniques in this experiment. Table 4.4 summarizes the results of the study. While CSCC correctly recommends more than 84% method calls for the top-3 recommendations for

Table 4.3: Correctly predicted method calls where the parent expression expects a particular type of value (top-3 proposals for the Eclipse system)

AST Node Types of Parent Node	Total Cases	ECCAlpha	ECCRel	BCC	FCC	BMN	CSCC
Method Argument	696	6	441	527	378	336	553
Assignment	429	3	282	338	144	184	305
If Statement	526	36	47	225	217	214	373
While Statement	8	0	0	4	0	2	4
Return	99	1	56	62	38	32	65
Variable Declaration Fragment	1388	10	738	1018	498	515	1084
For Statement	5	0	3	3	0	0	4
Class Instance Creation	126	3	76	95	46	54	98
Prefix Expression	425	30	75	396	282	228	346
Total	3702	89 2.4%	1718 46.4%	2668 72%	1603 44.5%	1565 42.27%	2832 76.5%

Table 4.4: Percentage of correctly predicted method calls that are called in the overridden methods (for the top-3 proposals)

Subject Systems	Percentage of correctly predicted method calls by Code Completion Systems (%)					
	ECCAlpha	ECCRel	BCC	FCC	BMN	CSCC
Eclipse	4.98	15.19	55.56	55.12	57.04	84.28
NetBeans	12.00	33.60	52.20	52.34	56.25	85.24

both subject systems, none of the other technique achieves more than 58% accuracy.

We were interested to see whether we can take advantage of this special case. There are two possible ways we can exploit the overridden method. First, we can include the method name in the context. Second, we can use the name to index training examples. To recommend a method call inside an overridden method we can then access candidates by using receiver type name and overridden method name.

To check the first option, we explicitly add the overridden method name to both overall and line contexts. It should be noted that CSCC may include the overridden method name as part of the overall context, but only in the case where the overridden method name is located within four lines of distance. In those cases, the overall context may contain the overridden method name twice. This even gives more weight on the

Table 4.5: Comparing performance (percentage of correctly predicted method calls) of CSCC at two different settings. The default setting does not explicitly include method name to both contexts, but the second setting does. Here, we only test those method calls that are located in overridden methods.

Recommendations	Subject Systems			
	Eclipse		NetBeans	
	Default	Include overridden method name (%)	Default	Include overridden method name (%)
Top-3	80.93	83.68	81.36	82.61

Table 4.6: Percentage of correctly predicted method calls that are called in the overridden methods (for the top-3 proposals)

Recommendations	Subject Systems			
	Eclipse		NetBeans	
	Without Index %	With Index %	Without Index %	With Index %
Top-3	83	73.4	83.10	70.9

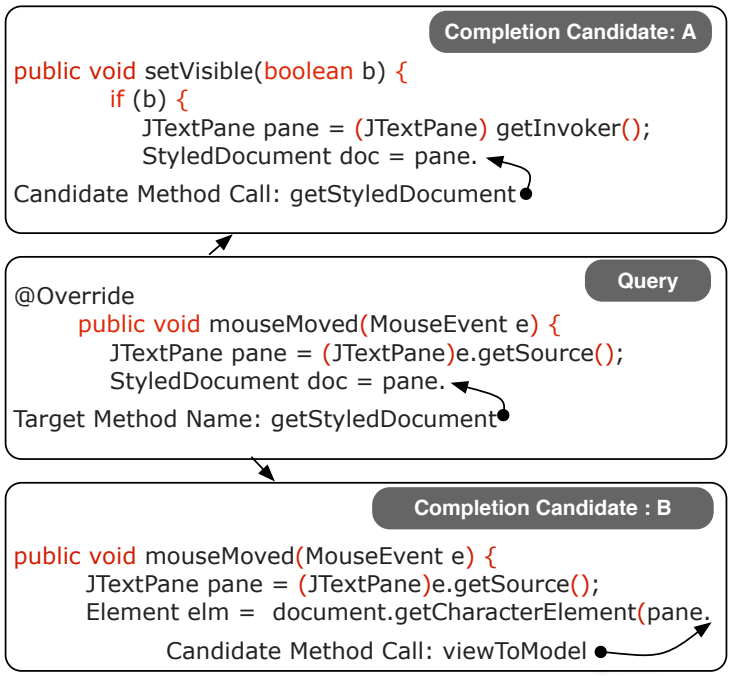


Figure 4.7: Indexing method calls by enclosing method name can lead to the wrong recommendation

overridden method name. For this experiment, we use Eclipse and NetBeans as subject systems and we test the accuracy of correctly predicting method calls within overridden methods. Table 4.5 shows the percentage of correctly predicted method calls that are located in overridden methods. For the Eclipse system, adding overridden method name to both contexts results in an increase of 2.52%. For the NetBeans system, we also obtain a relative improvement of 1.75%. The results from the study thus suggest that adding overridden method names to the context can help better model method call usage context for those that are located within overridden methods.

To check the second option, we again use Eclipse and NetBeans as subject systems. We indexed the code examples by receiver type and enclosing method name and we test the accuracy of the correctly predicted method calls for those test cases where the method calls appeared within an overridden method. Table 4.6 shows the results of the study. The results suggest that such an indexing scheme will not be very effective. Although we can use the indexing scheme to correctly recommend method calls, there are a number of cases where similar training examples may not be located inside overridden methods with same name. Figure 4.7 shows an example of indexing method calls by the enclosing method name that can lead to wrong recom-

Table 4.7: Runtime performance of CSCC generating a database of 40,863 method calls (column 2) and performing 4,540 code completions (column 3) for the Eclipse system.

Setup Used	Database Generation Time	Code Completion Time
CSCC (with inverted index)	8,066 ms	8,998 ms (Avg.1.94 ms)
Without inverted index	8,000 ms	12,888 ms (Avg.2.77 ms)

mendation. Here, the target method call is *getStyledDocument*. The top and the bottom examples represent two completion candidates. Although the enclosing method name of the completion candidate B matches with that of the query code (code under development, shown in the middle of the figure), the context does not match. Instead the correct match should be the completion candidate A whose enclosing method name does not match with that of the query code.

4.4.4 Comparison with Code Recommenders

Code Recommenders[10] is a more advanced Eclipse plugin that evolved from BMN. It utilizes a model trained using a set of code examples to provide intelligent code completion. When a developer invokes code completion in the IDE, Code Recommenders collect the current usage context and then looks in its model for possible matches to complete the code. Because the model is proprietary, we cannot train it with new code examples. Furthermore, because we did not have access to its internal API to obtain completion proposals automatically, we could only perform a manual comparison instead. Our comparison indicates that CSCC performed better than Code Recommenders. Our comparison is limited in scale due to its manual nature. Extensive evaluation may be possible in future if Code Recommenders becomes more open.

From the code examples in a book on the Java Swing framework,[11] we randomly selected 309 Swing/AWT method calls as our test cases. We enabled the Code Recommenders intelligent call completion in an Eclipse IDE. Then for each test case, we manually opened the corresponding file in the IDE, removed any code after the target object and tried to complete the method call by typing a dot (.) after the object name. We recorded the list of completion proposals suggested by Code Recommenders and determined the rank of the target method name in that list. To obtain the performance result for CSCC, we trained CSCC with the remaining examples and tested CSCC against the 309 selected method calls. CSCC achieves better result than Code Recommenders. The precison, recall and F-measure for Code Recommenders are 62%, 76%, and 68%, and for CSCC 92%, 82% and 87% respectively.

4.4.5 Runtime performance

To be useful, code completion must be done at near real-time in order to not interrupt a developer's flow of coding. Thus, to study the time required to suggest completion proposals, we measured the runtime of the

[10] `http://www.eclipse.org/recommenders/`

[11] `http://examples.oreilly.com/jswing2/code/`

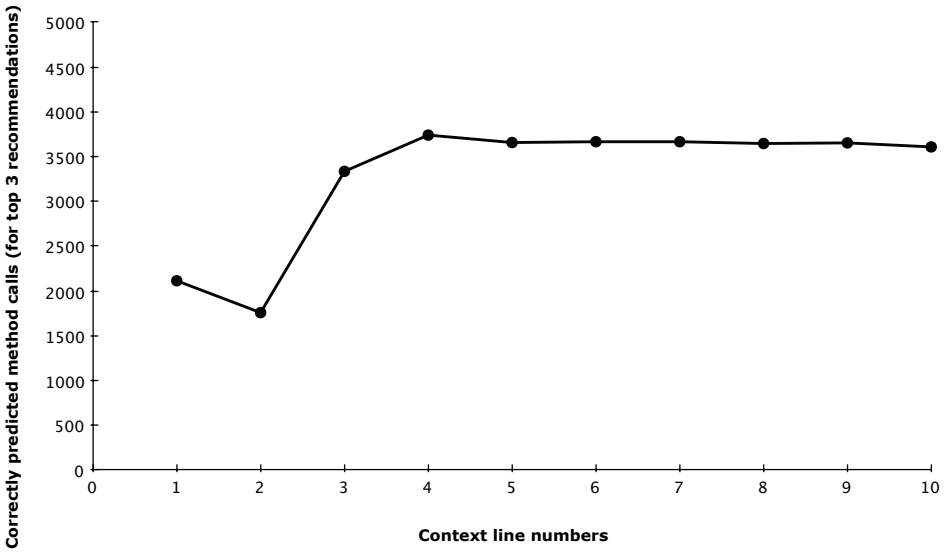


Figure 4.8: The number of correct predictions at different context size

first and last two steps of CSCC. The first step is responsible for building a candidate method call database and the last two steps are about recommending completion proposals. All experiments were performed on a computer running Ubuntu Linux with a 3.40 GHz Intel Core i7 processor and 10 GB of memory.

As shown in Table 4.7, we provide runtime data for the Eclipse subject system where the model is built using 40,863 method calls (column two) and the running time is the time required to test all 4,540 queries (column three). As expected, the first step takes the most time but the database needs to be built only once. On average, it takes 1.94 ms (milliseconds) to compute the completion proposals for each method call, which is negligible.

To understand the benefits of using the inverted index structure, we also developed a variant of our algorithm without using the inverted index structure and measured the runtime again. The result is summarized in the second row of Table 4.7. While the model generation time reduces slightly, the code completion running time increases considerably. Using the inverted index structure not only reduces the runtime of the algorithm, but also improves the result slightly by eliminating many irrelevant mapping candidates.

4.5 Discussion

4.5.1 Why does CSCC consider four lines as context?

For an API method call, we use four lines prior to the method call to determine the overall context. The number four is determined experimentally as follows. For this experiment, Eclipse is used as a subject system. We collect all SWT method calls and randomly select 10% for testing. The remaining 90% of calls are used to train CSCC. Next, we run the algorithm 10 times by varying context line numbers from one to ten. The higher the context line number, the larger the context size, which results in an increase in computation time.

We need to keep the number of context lines as low as possible without impacting performance. Figure 4.8 shows the accuracy of CSCC at various context line numbers. From Figure 4.8, we can see that at the beginning, the number of correctly predicted method calls drops when we increase the context size from one to two lines. However, we observe a sharp increase from that point for increasing the context size. When the context size increases to more than four lines there is no significant change in the number of correct predictions. Therefore, we set the context size to four lines.

4.5.2 Impact of different context information

We evaluate the impact of CSCC's various context information on the performance of method call prediction. We run an experiment on NetBeans, our largest Swing/AWT system.

Table 4.8 shows the percentage of correctly predicted method calls for different combinations of context information. In the first row, we model the overall context considering those methods that were previously called on the receiver object but without the enclosing method name. Next, we consider a variation of the previous model that takes into account enclosing method name. For the above two models, we neither consider any line context nor put any limit on context size. But all the models in the following five rows use overall context to recommend completion proposals. The third row corresponds to a model that only considers those method names that were previously called within four lines of distance on the same receiver object of the target method call. The fourth row represents a model that in addition to the above information, also considers the line context. The fifth row corresponds to another model that in addition to the previous information, also considers any other method names located within four lines of distance. The model in the sixth row takes into account any type names (class or interface names) appearing as part of class instance creation plus the previous information. Finally, the last row implements the complete CSCC, which also includes any Java keywords except access specifiers.

According to the results, CSCC in the last row achieves the highest accuracy. It is also clear from the table that the performance of CSCC is increasing with the addition of additional context information. Since adding the enclosing method name does not improve performance significantly (compare the first two rows),

Table 4.8: Sensitivity of performance for different context information

Model	**Correctly predicted method calls(%)**			
	Top-1	**Top-3**	**Top-5**	**Top-10**
Rec. method calls	46.5	69.5	77.5	82.5
Rec. method calls + enclosing method	46.6	68.7	76.9	81.8
Rec. method calls (within four lines)	33	60.20	76	84.80
Rec. method calls (within four lines)+line context	34.8	61.62	76.26	84.89
Previous factors + Other method calls	58	78	83	87
Previous factors + Type name	62	82	86	89
Previous factors + keyword (CSCC)	64	84	88	90

Table 4.9: Cross-project prediction results. P, R, and F refer to precision, recall, and F-measure, respectively.

Subject Systems			FCC (%)	BMN (%)	CSCC (%)
NetBeans	Top-1	P	39	46	69
		R	100	90	98
		F	56	61	**81**
	Top-3	P	64	77	84
		R	100	90	99
		F	78	83	**91**
JEdit	Top-1	P	57	70	66
		R	100	84	98
		F	73	76	**79**
	Top-3	P	69	85	83
		R	100	84	98
		F	82	84	**90**
ArgoUML	Top-1	P	48	48	54
		R	100	85	100
		F	65	61	**70**
	Top-3	P	65	68	77
		R	100	85	100
		F	79	76	**87**
JFreeChart	Top-1	P	43	44	68
		R	100	89	100
		F	60	59	**81**
	Top-3	P	74	85	88
		R	100	89	100
		F	85	87	**94**

we did not include enclosing method name in CSCC. Among various additional information we considered, method names and the type names (appears in the class instance expression) contributed the most. Although the addition of the line context improves the overall performance by only around 1%, during our manual investigation we found that in those small number of cases, overall context differs considerably, so line context complements the overall context in this case. Surprisingly, adding keyword names did not improve the performance significantly. We analyzed some cases manually and found that while they are effective, their effect diminishes in the matching process because of the presence of a large number of methods, and class/interface names in the context.

4.5.3 Effectiveness of the technique in cross-project prediction

We performed another experiment to evaluate the effectiveness of CSCC in cross project prediction. For this experiment, we considered Swing/AWT library method calls for four subject systems. This includes NetBeans, JEdit, ArgoUML and JFreeChart. We followed the approach described by Nguyen *et al.* [96]. We divide the data set of each system into ten different folds. For each system, we determine the precision and recall values as follows. For each fold of a system, we trained with nine other folds of the same system plus code examples from all other systems and then perform testing on the remaining fold. We then calculate the average of precision and recall values. To make the results comparable with BMN, we only provide prediction results for local variable method calls. Table 5.6 shows the results of our method call prediction.

CSCC once again performs better than other techniques. There are two important lessons to be learned from the result. First, both precision and recall of CSCC either slightly increase or are consistent with those of Table 4.1, indicating that CSCC can recommend correct completion proposals even when the training

model contains examples from different systems. As long as we have relevant usage contexts, CSCC can find them and can recommend completion proposals. Second, despite the considerable increase in the size of the training model, we did not notice significant improvement in performance. This seems to be consistent with Hindle *et al.*'s finding that the degree of regularity across project is similar to that in a single project [45].

4.5.4 Performance of CSCC for other frameworks or libraries

We already evaluated the performance of CSCC for API method call completion using two libraries, SWT and Swing/AWT. Both of them are used for developing graphical user interfaces. Thus, a threat to the validity of the study is that the performance of CSCC may be different for a different framework or library. We were interested to see whether the performance of CSCC is stable across different frameworks or libraries. For answering the question, we conduct another study using java.io and java.util method calls using two of our largest subject systems. The objective and usage pattern of these library method calls are different than those used for developing graphical user interfaces. The first one provides various API method calls to support system input and output, serialization and the file system. The second one contains various utility classes to facilitate working with date, time, collections, and events.

Table 4.10 shows comparison results of CSCC with five other state-of-the-art tools for the java.io API method calls. Results from the study suggest that both ECCAlpha and ECCRelevance perform poorly in the study. While the recall of BMN is lower than FCC, it achieves higher precision than FCC for the top recommendation and for the Eclipse system. In all other cases, FCC performs better than BMN for both subject systems. Interestingly, BCC performs better than both FCC and BMN. While the precision ranges from 47-58% for the top recommendation, the value increases to 70-81% for the top-3 recommendations. CSCC mostly achieves better result than any other techniques. For example, for the top position and for the Eclipse system CSCC achieves 15% more precision value (11% more F-measure value) than its closest competitor. For the NetBeans system, CSCC achieves at least 14% higher F-measure value than any other technique, 9% for the top-3 recommendations.

Table 4.11 shows comparison results for the java.util API method calls. Both ECCAlpha, ECCRelevance and BCC performs poorer than the other completion systems. Similar to the previous result, CSCC performs better than other code completion systems. For the top recommendation and for the Eclipse system, CSCC achieves a minimum of 25% more precision value (15% for the top-3 recommendations) than any other techniques. The technique also achieves at least 25% more recall value (10% for the top-3 recommendations). Although for the NetBeans system the difference is not that much, CSCC still performs the best.

4.5.5 Using bottom lines for building context

In the previous experiments, we assume a top-down programming approach and we ignore the presence of any code after the tested method call. Although our approach is consistent with the previous study [18] we cannot guarantee that the code was developed in that way. It may be the case that the developer copied the

Table 4.10: Evaluation results of code completion systems for the java.io method calls

Measurement	Code Completion Systems	Subject Systems					
		Eclipse			NetBeans		
		Top-1	Top-3	Top-10	Top-1	Top-3	Top-10
Precision	ECCAlpha	0.038	0.10	0.19	0.036	0.10	0.24
	ECCRelevance	0.13	0.27	0.29	0.14	0.30	0.57
	BCC	0.58	0.81	0.96	0.47	0.70	0.94
	FCC	0.47	0.72	0.91	0.38	0.63	0.89
	BMN	0.53	0.75	0.77	0.34	0.79	0.80
	CSCC	0.73	0.89	0.95	0.65	0.84	0.97
Recall	ECCAlpha	1	1	1	1	1	1
	ECCRelevance	1	1	1	1	1	1
	BCC	1	1	1	1	1	1
	FCC	1	1	1	1	1	1
	BMN	0.89	0.89	0.89	0.73	0.73	0.73
	CSCC	1	1	1	0.99	0.99	0.99
F-Measure	ECCAlpha	0.073	0.18	0.32	0.07	0.18	0.39
	ECCRelevance	0.23	0.43	0.32	0.25	0.46	0.73
	BCC	0.73	0.90	0.98	0.64	0.82	0.97
	FCC	0.64	0.84	0.95	0.55	0.77	0.94
	BMN	0.66	0.81	0.83	0.46	0.76	0.76
	CSCC	0.84	0.94	0.97	0.78	0.91	0.98

Table 4.11: Evaluation results of code completion systems for the java.util API method calls

Measurement	Code Completion Systems	Subject Systems					
		Eclipse			NetBeans		
		Top-1	Top-3	Top-10	Top-1	Top-3	Top-10
Precision	ECCAlpha	0.18	0.21	0.58	0.087	0.11	0.27
	ECCRelevance	0.34	0.43	0.76	0.17	0.21	0.32
	BCC	0.38	0.61	0.71	0.27	0.28	0.35
	FCC	0.36	0.74	0.97	0.73	0.86	0.95
	BMN	0.50	0.77	0.79	0.82	0.91	0.91
	CSCC	0.75	0.92	0.97	0.87	0.94	0.96
Recall	ECCAlpha	1	1	1	1	1	1
	ECCRelevance	1	1	1	1	1	1
	BCC	1	1	1	1	1	1
	FCC	1	1	1	0.98	0.98	0.98
	BMN	0.91	0.91	0.91	0.88	0.88	0.88
	CSCC	0.99	0.99	0.99	0.98	0.98	0.98
F-Measure	ECCAlpha	0.30	0.35	0.73	0.16	0.20	0.43
	ECCRelevance	0.51	0.60	0.86	0.29	0.35	0.48
	BCC	0.55	0.76	0.83	0.43	0.44	0.52
	FCC	0.53	0.85	0.98	0.84	0.92	0.96
	BMN	0.65	0.83	0.85	0.85	0.89	0.89
	CSCC	0.85	0.95	0.98	0.92	0.96	0.97

entire piece of code from a different place and performed edit operations. It may also be the case that, during the code review process, a developer replaced a line or changed a method call on the line with a different one. In such cases, we may leverage the bottom lines of code for building context. Due to the lack of complete edit history, we were unable to find those method calls only. However, an alternative approach can be to go for the best situation, that is use the bottom lines of code already developed and test whether considering those lines with the top lines improve the performance of our code completion systems.

For this experiment, we develop a new version of CSCC that uses both top and bottom four lines including the line in which the method call appears to generate the usage context. For evaluation, we use two of our largest subject systems, Eclipse and NetBeans. For Eclipse, we collect the API method calls for the SWT library and for NetBeans we collect AWT and Swing API method calls. We then use the ten-fold cross validation technique to compare the performance of new version of CSCC with the old one.

Table 4.12 compares the correctly predicted method calls for two different forms of CSCC. Here, the term

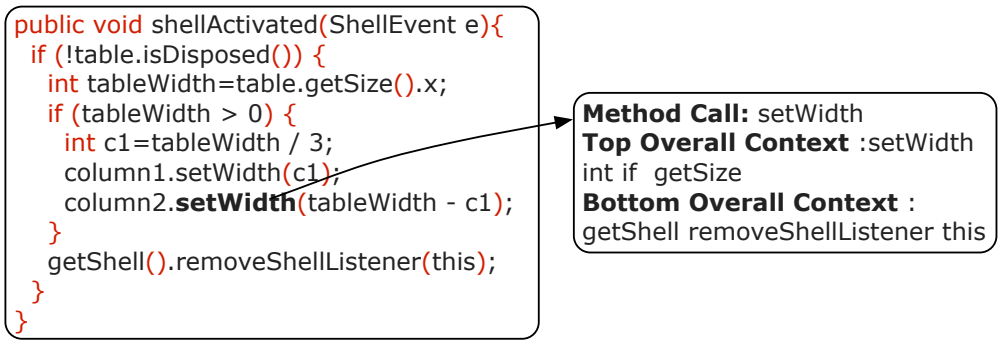


Figure 4.9: An example of top and bottom overall context. When we use both top and bottom context, we concatenated the terms appearing in the bottom context to the terms appearing in the top context

Table 4.12: The number of correctly predicted method calls using two different forms of usage context.

Recommendations	CSCC			
	Eclipse		NetBeans	
	Top Context %	Both Context %	Top Context %	Both Context %
Top-1	62	63	64	70
Top-3	80	81	85	86
Top-10	90	90	91	91

Top Context refers to the original version of the technique. The term Both Context refers to the modified version of the technique that considers both top and bottom four lines to construct usage context of method calls. Results from the study suggest that considering the bottom four lines does not improve results for top-3 or top-10 recommendations. However, we observe improvement for the top recommendation. Although we find only 1% improvement of accuracy for the Eclipse system, the number increases to 6% for the NetBeans system. This suggest that using bottom four lines for building context can be useful.

4.6 Extending CSCC for Field Completion

After releasing CSCC as an Eclipse plugin we received considerable feedback from users. Many of them asked to extend the automatic code completion support for fields also. During our investigation, we found that field completions are not trivial because of the possibility of a large number of choices. The objects we instantiate from different libraries and frameworks often contain a large number of field variables. Before calling methods on those objects, we either assign values to those fields or we may access them to check preconditions. Many of these fields are also constants. The problem is that there are a large number of them and it is difficult for a developer to remember each. Although classes define or inherit a large number of field variables from other classes, objects instantiated from those classes only use a few of them in practice.

Fortunately, many of these field variables are meant to be used in distinct contexts. For example, when we add a swing component to a container, we need to state a field constant indicating the location of the container where the component needs to be added. BorderLayout is a popular layout manager that uses static

A

```
public void displayKeyInfo(KeyEvent event,
            string status) {
   int keyId = event.getID();
   String typedKeyString;
   if (keyId == KeyEvent.Key_Typed){
      char c = event.getKeyChar();
      typedKeyString = "character: " + c;
   }
   else{
       int keyCode = event.getKeyCode();
       typedKeyString = "key code: " + keyCode +","
                      + KeyEvent.getKeyText(keyCode)
   }
        //collect key location
        //display information about key event
}
```

B

```
public void drop(DropTargetEvent event) {
       String action = null;
       switch (event.detail) {
          case DND.DROP_MOVE: action = "moved";
                              break;
          case DND.DROP_COPY: action = "copied";
                              break;
          case DND.DROP_LINK: action = "linked";
                              break;
          default: action = "unspecified";
                   break;
       }
       text.append("\n" + action);
}
```

Figure 4.10: Examples of field accesses (highlighted in bold)

field variables to represent various locations of a container. It supports both absolute and relative positioning constants. However, mixing them can result in an unpredictable result. Automatic code completion support can help in this regard by suggesting the correct positioning constants. As another example, the java.awt.event class contains 78 field variables (67 of them are used as field constants). However, depending on the context of using an event object, only a few of them are meant to be accessed or used. For instance, while writing a piece of code for mouse handling we need to access an event object to detect the mouse button pressed by a user. There are only five field constants of the event object that can be used to determine different states of a mouse and other field constants are simply irrelevant in this context. Automatic code completion support can help in this regard by recommending relevant field constants in top positions.

Figure 4.10 shows an example of a field access (see the top figure). The objective of the method is to display information about the key that generates the event. Although the key event object has a getChar() method, we can only rely on that if the event is a key typed event. Thus, the event id is checked using an if condition to determine whether the generated event is a key typed event. An intelligent code completion system should suggest the correct completion proposal in the top places. However, the default code completion systems of Eclipse did not perform well for this case. ECCRelevance suggests the target code completion proposal in the tenth position and ECCAlpha performs the worst, suggesting the target proposal in the 44th position. Developers typically call the *getId()* method to collect the key id before using the field access. CSCC can collect this information as usage context and can recommend the correct field access in the top position.

4.6.1 Changes made

Extending CSCC to support field completion was easy because of the simplicity of the technique. Recall that the algorithm works in three steps where the first step deals with collecting usage examples for target code completion. We change this step so that CSCC mines code examples to identify and collect usage context of field accesses. The usage context of a field access consists of the same information we use to capture the method call usage context.

We again use a two-level indexing scheme where the type name of the receiver object is used to group all the fields that are used with that type and we use an inverted index structure to organize the group of field accesses. When a developer requests for a field completion, we first determine the candidates for the field completion and then synthesize the results to recommend the top-3 field names. These steps are identical to those we described earlier for API method completion. In summary, CSCC did not require major changes to support automatic field completion.

4.6.2 Evaluation procedure

We evaluate field completion of CSCC with four other state-of-the-art code completion systems. For a given subject system, we determine locations of all field accesses where the receiver type matches with the type name of the target framework or library. We then apply ten-fold cross validation technique to collect evaluation results. Next, we use the precision, recall and F-measure to measure the performance of each algorithm. These are same measures we used earlier for API method completion. The only change is that instead of method calls we now consider field accesses.

The code completion systems we consider in this study besides CSCC are ECCAlpha, ECCRelevance, FCC and BMN. ECCAlpha and ECCRelevance are the two default code completion systems that also support automatic field completions. We include FCC in this study that sorts field accesses based on their frequency in the training data. We exclude BCC from this study because the current implementation of the tool does not support the field completion. We also include BMN in this study to see whether the usage context BMN used to recommend method calls can be used to recommend field accesses.

We consider two APIs (SWT and Swing/AWT) and all eight subject systems we used previously for evaluating method call completion. Since developers request for a field completion in the same way of a method completion (for example, in Eclipse this can be done by typing a dot after a receiver name), we train each technique using both API field and method calls, but we test them for field accesses only. For the four subject systems (Eclipse, Vuze, Subversive and Rsowl), we collect all SWT field and method calls. We collect Swing/AWT field and method calls for the remaining four subject systems.

Table 4.13: Evaluation results of code completion systems for field accesses.

Subject Systems		Precision					Recall					F-Measure				
		ECCAlpha	ECCRel	FCC	BMN	CSCC	ECCAlpha	ECCRel	FCC	BMN	CSCC	ECCAlpha	ECCRel	FCC	BMN	CSCC
Eclipse	Top-1	0.01	0.09	0.24	0.29	0.39	1	1	1	0.95	0.95	0.02	0.16	0.38	0.44	0.55
	Top-3	0.02	0.23	0.52	0.54	0.70	1	1	1	0.95	0.95	0.04	0.36	0.69	0.69	0.81
	Top-10	0.19	0.45	0.74	0.74	0.89	1	1	1	0.95	0.95	0.31	0.62	0.85	0.83	0.92
Vuze	Top-1	0.02	0.09	0.28	0.28	0.42	1	1	1	0.91	0.94	0.03	0.16	0.44	0.42	0.58
	Top-3	0.03	0.22	0.56	0.54	0.73	1	1	1	0.91	0.94	0.05	0.36	0.71	0.68	0.83
	Top-10	0.18	0.42	0.77	0.72	0.92	1	1	1	0.91	0.94	0.30	0.60	0.87	0.81	0.93
Subversuve	Top-1	0.00	0.03	0.31	0.30	0.40	1	1	1	0.89	0.95	0.01	0.05	0.48	0.45	0.57
	Top-3	0.01	0.06	0.58	0.60	0.73	1	1	1	0.89	0.95	0.09	0.11	0.74	0.71	0.83
	Top-10	0.20	0.41	0.82	0.74	0.94	1	1	1	0.89	0.95	0.34	0.58	0.90	0.81	0.94
Rsowl	Top-1	0.01	0.07	0.22	0.14	0.46	1	1	1	0.82	0.94	0.01	0.13	0.37	0.24	0.62
	Top-3	0.02	0.15	0.50	0.28	0.83	1	1	1	0.82	0.94	0.04	0.27	0.67	0.42	0.88
	Top-10	0.07	0.23	0.78	0.58	0.94	1	1	1	0.82	0.94	0.13	0.38	0.88	0.68	0.94
NetBeans	Top-1	0.11	0.30	0.35	0.21	0.43	1	1	1	0.46	0.95	0.19	0.46	0.51	0.30	0.59
	Top-3	0.16	0.50	0.60	0.49	0.71	1	1	1	0.46	0.95	0.28	0.66	0.75	0.48	0.82
	Top-10	0.38	0.79	0.90	0.87	0.90	1	1	1	0.46	0.95	0.55	0.88	0.95	0.61	0.93
JEdit	Top-1	0.04	0.19	0.21	0.26	0.32	1	1	1	0.69	0.78	0.08	0.31	0.35	0.37	0.45
	Top-3	0.07	0.46	0.54	0.57	0.61	1	1	1	0.69	0.78	0.13	0.63	0.70	0.62	0.68
	Top-10	0.35	0.69	0.79	0.72	0.75	1	1	1	0.69	0.78	0.52	0.81	0.88	0.70	0.76
ArgoUML	Top-1	0.01	0.16	0.26	0.25	0.31	1	1	1	0.70	0.87	0.01	0.27	0.41	0.36	0.46
	Top-3	0.03	0.39	0.60	0.56	0.65	1	1	1	0.70	0.87	0.05	0.57	0.74	0.62	0.74
	Top-10	0.21	0.71	0.81	0.71	0.85	1	1	1	0.70	0.87	0.35	0.83	0.89	0.70	0.86
JFreeChart	Top-1	0.32	0.40	0.48	0.47	0.56	1	1	1	0.90	0.96	0.48	0.57	0.65	0.61	0.71
	Top-3	0.39	0.59	0.75	0.71	0.85	1	1	1	0.90	0.96	0.56	0.74	0.85	0.80	0.90
	Top-10	0.57	0.66	0.95	0.90	0.97	1	1	1	0.90	0.96	0.73	0.79	0.97	0.90	0.96

4.6.3 Evaluation results

Table 4.13 shows the precision, recall and F-measure values for five code completion systems using Eclipse and NetBeans as the subject systems. Unsurprisingly, both ECCAlpha and ECCRelevance did not perform well in our field completion study. For example, for the Eclipse system, ECCAlpha performs the worst. ECCRelevance becomes the second last and it achieves better precision, recall and F-measure values than ECCAlpha. Both FCC and BMN perform better than the default code completion systems of Eclipse. BMN achieves higher accuracy for the top position comparing to the FCC. While the precision of FCC is 52% for the top-3 positions, BMN achieves 54%. We also do not observe much difference in recall and F-measure values. CSCC achieves the highest F-measure value than any other code completion systems for the Eclipse system. The technique achieves 12% higher precision value than its closest competitor for the top-3 positions and the recall value is also high. For the remaining three subject systems that uses SWT API, CSCC performs better than any other techniques considered in this study. While it has precision 73-83%, the recall is 94-95%.

In general, for the remaining four systems that use Swing/AWT API, CSCC holds its position. It has a precision of 61-85% and recall 87-96%. While for the top position, CSCC achieves the highest precision value, FCC achieves comparable performance as we increase the number of recommendations. The performance of CSCC is also less significant compared with FCC this time. This is due to the fact that the test cases have less possible completion candidates for systems that use Swing/AWT API compared to those that use SWT. For example, NetBeans has less number of completion candidates per query than the Eclipse system (see Figure 4.11). While BMN achieves precision value similar to that of the previous four systems, the recall value drops, largest for the NetBeans system. After investigation, we found that for the NetBeans system, a considerable number of test case receivers (almost 50%) are other than simple name type and BMN could not make any recommendations for those cases.

4.6.4 Training with method calls and field accesses together

In all the previous experiments, we train example-based code completion systems using only method calls or field accesses. Therefore, given a receiver type, the code completion systems retrieve either method names or field accesses, not both. However, in an integrated development environment (such as in the Eclipse IDE),

Table 4.14: The accuracy (in percentages) of correctly predicted field accesses for two different settings. For setting A, we train code completion systems using both fields and methods. For setting B, we only use fields for training. Both settings use fields for testing.

Subject Systems	**Code Completion Systems**	**Accuracy (%)**					
		Top-1		Top-3		Top-10	
		A	B	A	B	A	B
Eclipse	FCC	23.09	23.10	51.53	51.56	74.77	74.79
	BMN	27.45	27.71	52.74	52.84	72.22	72.47
	CSCC	39.23	39.96	71.24	70.01	89.98	89.64
NetBeans	FCC	33.92	36.57	59.33	69.63	90	92.53
	BMN	10.92	14.25	25.36	28.27	45.31	47.09
	CSCC	43.17	44.58	70	71.12	88.34	89.43

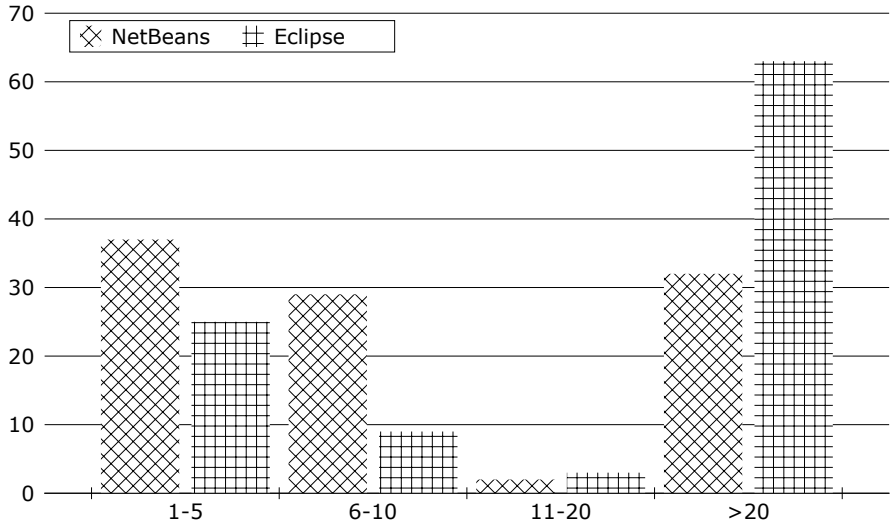


Figure 4.11: Distribution of test cases in different candidate groups. We group the test cases into four categories based on the number of completion candidates. X-axis positions four different groups and the Y-axis value refers to the percentage of test cases in a group.

when a developer requests for a code completion by typing dot (.) after a receiver name, the target can be a method call or a field access. Therefore, it is required to train code completion systems using both method calls and field accesses. However, given a receiver type, the system can retrieve both method calls and field accesses whose receiver type matches with the target receiver type. This larger set of code completion candidates can affect the performance of code completion systems. To determine how this affects their performance, we conduct the experiment of field completion proposals for the largest two subject systems (Eclipse and NetBeans). This time we train each technique using both method calls and field accesses (i.e., setting A), and using only field accesses (i.e., setting B). The test data sets are identical to the previous study on evaluating field completion proposals. We exclude the default code completion systems of Eclipse (ECCAlpha and ECCRel) because they do not require any training.

Table 4.14 shows the accuracy (in percentage) of three example-based code completion systems for two different subject systems. As shown in the table, there is a little or no effect of training code completion systems using only fields for the Eclipse system. When we investigate setting A further, it reveals that, in general, the query completion candidates retrieved for field completion overlap very little with those for method completion. For the NetBeans system, the completion candidates per query retrieved by the receiver type contain higher percentage of methods than the Eclipse system. That is why we notice changes in accuracy between two different settings for the NetBeans system. For example, FCC achieves more than 10% accuracy for the top-3 recommendations when training with only fields. BMN is no exception and the least affected code completion system is the CSCC with only around 1% change in accuracy between two different settings. This has two important implications. First, CSCC can easily be adapted to different forms of recommendations with little changes. This is due to the simplicity in usage context construction,

synthesizing examples and recommendation formulation of the technique. Second, despite the simplicity, the usage context used by the CSCC is generic in nature and can easily capture different usage scenarios. Thus, CSCC can support both method and field completions without making any significant changes.

4.7 Comparison With Statistical Language Model-based Code Completion Techniques

A statistical language model computes either a probability distribution over a sequence of words or the likelihood of occurring a word w_n given a sequence of prior words. These models show great promise in various software engineering tasks. This includes, but is not limited to, predicting source code comments [89], locating syntax errors in source code [21], and identifying coding conventions [2]. These techniques become successful due to the high degree of repetition and repetitiveness that exist in source code. A number of code completion techniques have been recently developed leveraging statistical language models. We are interested in evaluating the effectiveness of those techniques in completing method or field accesses and compare the result with CSCC. Unless otherwise specified we now use the term code completion to refer to both.

4.7.1 Statistical language models

We consider four statistical language model based code completion techniques and CSCC in this study. This includes N-gram, Cache LM, CACHECA, and GraLan [36, 45, 93, 137]. N-gram identifies naturally occurring sequences of tokens in source code. Given a token sequence, N-gram identifies those tokens that tend to follow that token sequence in corpus. Cache LM improves the performance of the traditional N-gram model by capturing locally-repetitive token sequences using a cache component. We add Cache LM in our study to find the effectiveness of the cache component for code completion. CACHECA becomes the third technique in our study and we include it to determine whether combining Eclipse suggestions with Cache LM leads to better result or not.

Unlike the previous three techniques that work at the lexical level, GraLan works at the statistical and data dependency level. We do not include SLANG in this study because GraLan showed better code suggestion accuracy than the technique and we have already included GraLan in this study. We do not consider SLAMC in this study because the tool is not available. We include ECCRel in this study because CACHECA merges results of ECCRel with that of Cache LM. We are interested to identify the performance improvement of CACHECA over ECCRel.

For the N-gram, we use a trigram language model. For the Cache LM, we set the cache scope to the current file or related files (file cache). For cache size and order, default settings are used. We also enable the back-off technique for cache-LM. GraLan requires two parameters. The first parameter (θ) is used to limit the number of API calls that are used to discover context subgraphs. The second parameter (δ) is used to limit the number of context graphs. We set both values to 8 as in previous work [93].

Table 4.15: Comparing CSCC with four statistical language model-based techniques (N-gram, Cache LM, CACHECA, and GraLan). Since CACHECA merges results of ECCRel with those of Cache LM, we also include ECCRel in the comparison

Subject Systems		Code Completion Technqiues					
		N-gram	Cache LM	EccRel	CACHECA	GraLan	CSCC
Eclipse	MRR	0.24	0.39	0.15	0.30	0.60	0.61
	Top-1	0.13	0.29	0.07	0.17	0.44	0.46
	Top-3	0.30	0.45	0.16	0.35	0.71	0.70
	Top-10	0.47	0.59	0.32	0.66	0.84	0.85
Vuze	MRR	0.34	0.44	0.13	0.26	0.59	0.60
	Top-1	0.23	0.35	0.06	0.13	0.34	0.45
	Top-3	0.41	0.51	0.14	0.27	0.59	0.69
	Top-10	0.57	0.65	0.29	0.64	0.80	0.80
SubVersive	MRR	0.36	0.43	0.10	0.34	0.58	0.68
	Top-1	0.25	0.32	0.03	0.18	0.42	0.52
	Top-3	0.44	0.52	0.06	0.44	0.63	0.76
	Top-10	0.57	0.64	0.30	0.69	0.83	0.86
Rsowl	MRR	0.36	0.48	0.20	0.43	0.49	0.59
	Top-1	0.24	0.36	0.12	0.28	0.30	0.42
	Top-3	0.45	0.57	0.23	0.51	0.57	0.64
	Top-10	0.61	0.71	0.33	0.77	0.75	0.72
NetBeans	MRR	0.43	0.52	0.33	0.48	0.70	0.68
	Top-1	0.31	0.40	0.19	0.31	0.50	0.54
	Top-3	0.54	0.62	0.40	0.59	0.76	0.75
	Top-10	0.66	0.74	0.61	0.82	0.88	0.86
JEdit	MRR	0.25	0.42	0.23	0.38	0.60	0.58
	Top-1	0.15	0.33	0.11	0.21	0.37	0.34
	Top-3	0.31	0.49	0.29	0.51	0.63	0.53
	Top-10	0.48	0.61	0.41	0.71	0.74	0.61
ArgoUML	MRR	0.30	0.45	0.31	0.47	0.54	0.59
	Top-1	0.20	0.35	0.15	0.32	0.32	0.41
	Top-3	0.37	0.53	0.37	0.57	0.54	0.56
	Top-10	0.48	0.64	0.66	0.78	0.69	0.65
JFreeChart	MRR	0.38	0.63	0.29	0.53	0.57	0.64
	Top-1	0.26	0.54	0.18	0.38	0.35	0.46
	Top-3	0.46	0.69	0.29	0.62	0.69	0.74
	Top-10	0.65	0.82	0.61	0.93	0.89	0.88

4.7.2 Evaluation procedure

We use the ten-fold cross validation to measure the performance of each technique. To train these techniques, folds are created based on source files. This means that all framework method calls that appear in a source file are either used for training or for testing. We create each fold in such a way that they contain equal number of framework method calls, although the number of source files can be different. We use the same eight subject systems we used in the previous experiment. We collect all SWT method calls and field accesses for the first four subject systems (Eclipse, Vuze, Subversive, and Rsowl) and for the remaining four subject systems (NetBeans, JFreeChart, JEdit, ArgoUML) we collect all Swing/AWT method calls and field accesses.

To make the result comparable with Cache LM, we use the Mean Reciprocal Rank (MRR) and top-k accuracy in this study. For each framework method calls/field accesses in the test data, each technique produces a ranked list of suggestions. The reciprocal rank is calculated by taking the multiplicative inverse of a rank. Mean reciprocal rank is the average of reciprocal ranks for all n framework method calls in the test data. This can be defined as follows:

$$MRR = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{rank_i} \tag{4.4}$$

where $rank_i$ denotes the rank of the ith test method call or field access. If a suggestion list does not contain the correct answer, we use a reciprocal rank of 0. We also measure the top-k suggestion accuracy for

each technique, which is the ratio of the number of cases a technique produces the correct recommendation within the top-k recommendations over the total number of test cases. We report results for top-1, top-3 and top-10 positions.

4.7.3 Evaluation results

Table 4.15 shows the results of our study. We compare CSCC with the four statistical language models (N-gram, CacheLM, CACHECA, and GraLan) in terms of MRR, Top-1, Top-3, and Top-10 accuracies, by performing the directional Wilcoxon Signed Rank statistical test. The tests reveal that in most cases CSCC either outperforms or performs equally well as GraLan, which is the best performing among the statistical language models we examined. This is due to the fact that the technique leverages API usage graphs for recommending method calls or field accesses. For the first four systems, the accuracy of GraLan ranges from 57-71% for the top-3 recommendations. The value ranges from 54-76% for the bottom four systems.

As expected, N-gram obtains the lowest accuracy and MRR value among all statistical language model techniques. However, N-gram performs better than EccRel, indicating that prior code context leads to better result than using only static type information. Cache LM augments the N-gram model with a cache component that results in performance improvement. While the MRR of N-gram ranges from 24% to 43%, for the CacheLM the value ranges from 39% to 63%. The number of correct recommendations for the top position is also much higher than the N-gram technique. However, Cache LM may fail to recommend those method calls or field accesses in top positions that are not locally repetitive. CACHECA combines the recommendations of cache LM with that of Eclipse. However, we observe that the strategy CACHECA uses to combine recommendations from two different sources affect the MRR value. CACHECA can recommend those cases where CacheLM fails but with a sacrifice of the MRR value. The MRR of CACHECA ranges from 26% to 53%. The accuracy of top-3 recommendations ranges from 27% to 62%.

In general, CSCC performs the best compared to all other techniques. While the accuracy ranges from 46-74%, the MRR ranges from 58-68%. Despite the simplicity, CSCC shows better performance than GraLan. When we investigate the reason we found that CSCC is able to recommend suggestions in more cases than GraLan. For example, there are cases where the prior context is empty for GraLan. Although GraLan cannot recommend in those cases, CSCC can because it either finds context or considers empty context for recommending suggestions. We also observe a few cases where GraLan outperforms CSCC. For example, GraLan performs better than CSCC for all metric values for the JEdit system and three out of the four metric values of the NetBeans system. It would be interesting to investigate in future why GraLan does better in these cases.

Overall, CSCC shows better performance than statistical language model-based techniques. We investigate whether performance improvement of CSCC is statistically significant. We perform directional Wilcoxon Signed Rank Test for the MRR and accuracy at top-1, top-3 and top-10 recommendations. The null hypothesis is that there is no difference in MRR or accuracy values. The tests show that the difference in accuracy

Table 4.16: Comparison of CSCC with Cache LM considering locally repetitive method calls and field accesses

Techniques	Eclipse				NetBeans			
	MRR	Top-1	Top-3	Top10	MRR	Top-1	Top-3	Top-10
CSCC	0.63	0.47	0.75	0.88	0.71	0.56	0.79	0.90
Cache LM	0.63	0.54	0.70	0.77	0.70	0.60	0.81	0.88

at top-1 is statistically significant at the p value of 0.05. However, the result is not statistically significant for MRR or accuracy at top-3 and top-10 recommendations. We conclude that CSCC outperforms GraLan for top-1 recommendation and performs equally well in other cases (MRR and accuracy at top-3 and top-10 recommendations).

4.7.4 How good is CSCC at recommending locally repetitive method calls or field accesses?

Previous experiments show that a cache component is helpful to capture the locally repetitive method calls or field accesses that are otherwise difficult to detect by the N-gram model. This raises the following question: is CSCC capable of capturing those locally repetitive calls? We conduct a study to answer the question. We use the same ten-fold cross-validation technique, but this time we collect those field accesses or method calls for testing that appear more than once in a file. We exclude those that appear only once in a file. We also exclude the first occurrence of those calls from testing that appear multiple times in a file because a cache language model may fail to recommend them because of a cache miss. We conduct the experiment using two of our largest subject systems, Eclipse and NetBeans. For the Eclipse system, we collect SWT field accesses and method calls. For the NetBeans system, we collect Swing/AWT library field accesses and method calls. Table 4.16 summarizes results of our study. For both systems, CSCC obtains similar or slightly better MRR metric values than Cache LM. While for the top position Cache LM achieves better result than CSCC (7% higher than CSCC for the Eclipse system and 4% higher for the NetBeans system), performance improves with the increase of number of recommendations. For example, CSCC obtains 5% higher accuracy than Cache LM for the top-3 positions. These indicate that CSCC is able to recommend locally repetitive field or method calls with good accuracy.

4.8 Threats to Validity

In this section, we briefly describe several threats to the validity of this study.

First, we considered only two APIs during the evaluation of field completion. One can argue that the result may be different for a different framework or library. While it can be beneficial to test with additional libraries for other reasons, given that CSCC does not directly rely on these libraries, we believe that this is highly unlikely and that the results we obtain in this study should largely carry over to additional libraries. Moreover, we tested CSCC for method call completion for two different libraries other than those two used

in the evaluation of field completion. Results from that study also suggest that the performance of CSCC is not affected by the kind of library.

Second, we re-implement the BMN system since both the data and implementation of the technique are not available. We also re-implement GraLan due to lack of its implementation. However, instead of working from the scratch we reuse the implementation of Groum for identifying API usage graphs. Although we cannot guarantee that our replication of these techniques does not contain any errors, we have spent a considerable amount of time implementing and testing the technique to minimize the possibility of introducing errors.

Third, in this study we only consider top four lines to determine the context of a method call because we assume a developer is typing code in a top-down manner, which is consistent with previous studies [18]. However, it is also possible that a developer can edit existing code, in which case we can use both top and bottom lines of a method call to create context. Although we conduct an experiment to see the benefit of generating context using both top and bottom four lines, we can not guarantee that the code was developed in the same way as we tested it due to the lack of a change based software repository.

Fourth, we compare CSCC with GraLan to determine the effectiveness of using a graph based statistical language model in method call completion. The API code suggestion engine of GraLan allows users to adjust two parameter values. One can argue that the accuracy of GraLan can be improved by adjusting those parameter values. We would like to point out that we use the same settings used in the GraLan study [93]. It might be possible to further fine-tune GraLan implementation by changing parameter values as a future work.

Fifth, to compare GraLan with CSCC we use the mean reciprocal rank (MRR) and the top-k accuracy metrics. We chose these metrics because they were used by previous code completion studies [96, 137], thus providing a common way of comparison. As future work, we would also be able to compare CSCC with GraLan by considering training time, recommending time, total memory, or external space usage.

Finally, statistical language model-based code completion techniques (i.e., N-gram, Cache LM and CACHECA) were originally developed for recommending a variety of tokens whereas CSCC is designed to recommend method calls and field accesses. Thus, statistical language model-based code completion techniques are typically evaluated considering all token kinds. Although it could be possible to adjust those techniques for recommending specifically method calls and field accesses, for example, by training the models with only method calls or field accesses, we did not do that. This is because we were interested in identifying the effectiveness of statistical language models in recommending method calls.

4.9 Conclusion

In this chapter, we present a simple, efficient, scalable, context sensitive code completion technique, called CSCC. CSCC mines previous code examples to recommend completion proposals. CSCC is simple because it is based on tokenization, instead of parsing or other more advanced analysis. It is efficient and scalable

due to its ways of measuring context similarity: using *simhash* first as a coarse-grained but efficient filter, and LCS/Levenshtein distance second as a refined, more accurate similarity metrics. We have compared CSCC with other state-of-the-art code completion systems using two popular libraries. CSCC performed better than state-of-the-art static type or context-sensitive, example-based systems considered in our study. We then propose a taxonomy of method calls to identify the effect of different context elements on code completion techniques.

CSCC utilizes a simple form of usage context. It collects any method names, any keywords except access specifiers, any class or interface names that occurs within the top four lines including the line in which a method call appears prior calling a method. Although we initially developed and tested CSCC for method call completion, we later found that the usage context information is not limited to method calls only. We conduct a study to see whether CSCC can support the field completion using the same usage context information we collect for method calls. The results from the study suggest that CSCC can easily support field completions with little changes.

Finally, we compare CSCC with four statistical language models. CSCC not only outperforms all three techniques that work at the lexical level, but also in most cases performs better or equally well with GraLan, the state-of-the-art graph-based language model that leverages API usage graphs to recommend API elements. We also implement the technique as an Eclipse plugin which is also available online for public use. The code of CSCC, data files used in this experiment, and the plugin can be found online.[12]

[12] `http://asaduzzamanparvez.wordpress.com/cscc/`

Chapter 5

Exploring API Method Parameter Recommendations

The previous chapter described a method call completion technique, called CSCC. The technique was compared with existing state-of-the-art code completion techniques. However, none of the techniques focus on completing method parameters, which is also a non-trivial task. The term 'parameter' refers to the actual parameter (or argument) and not the formal parameter. This chapter explores how developers complete method parameters. Based on this observation, a technique is developed that supports the largest number of parameter expression types. The proposed parameter completion system is compared with existing state-of-the-art techniques. The study results and the implementation of the technique have been published in a major software engineering conference [4, 5].

5.1 Introduction

Developers use framework and library APIs to reuse code during software development. This not only speeds up development but also saves time and resources. However, studies have shown that it is difficult to learn APIs due to various factors, such as inadequate examples and documentation [116, 117]. It is also difficult to remember APIs due to their sheer volume. To alleviate these problems, modern integrated development environments (IDEs) contain a code completion feature. It has been found that code completion is one of the top ten commands used by developers [90]. It speeds up the process of writing code by reducing typos or other programming errors, and frees the developer from remembering every detail. For example, as a developer types a method call, a code completion system typically uses autocomplete popups to recommend a set of method names, and the developer can then select the appropriate method call from the list. Although a number of techniques have been developed, most focus on the problem of suggesting method calls and leave the task of completing method parameters to the developers. However, determining the correct parameters to complete a method call is a non-trivial task and requires more attention [144].

Incorrect use of API method parameters can lead to software bugs [106, 107] or can cause runtime exceptions. For example, the *valueOf* method in the Java String class returns the string representation of the parameter. The *valueOf* method has been overloaded to accept different forms of parameters and the overloaded method is chosen according to the static type of the parameter. One of the overloaded methods takes an Object as a parameter and the other takes a char array (char[]). The first method contains a check

for the null value but the other one does not. The call String.valueOf(null) will execute the second overloaded method and throw a null pointer exception since type char[] is more specific than Object according to the Java language specification. However, casting the parameter to Object selects the first overloaded method. Since this method has a check for null value, the method will not throw an exception.[1] Although developers can identify and solve the problem by searching online and reading documentation, the effort is considerable. A parameter recommendation technique can help avoid this extra effort by suggesting the required casting operation.

There has been very little research on the problem of automatic parameter completion; however, Zhang *et al.* [144] have developed a parameter completion tool: *Precise*. Precise mines previous code examples to collect parameter usage patterns. Given a request for parameter completion, the tool uses the k-Nearest Neighbor (k-NN) algorithm to recommend proposals by matching the similarity of the current context with the past parameter usage examples. Despite the contribution of Precise, we see a gap between their work and what we can do regarding parameter completion. In this study, we follow on from their work to further explore the problem and to improve support for parameter completion.

Unlike the study by Zhang *et al.* [144], we start with a manual investigation to understand how developers complete method parameters. During our study, we found that parameter usages are *locally specific*. For example, before using an array access expression as a method parameter, developers typically instantiate the array close to that parameter position. When a method invocation is used as a method parameter, other method calls and language constructs that are related to that method call typically are located close together. We conduct an exploratory study to understand parameter usage and leverage the findings to support parameter completion. In particular, we answer the following research questions:

RQ1: Is source code locally specific to parameters?

By studying the local context of parameter usage, we hope to gain insights into building a more robust parameter completion technique.

RQ2: How can we capture the localness property to recommend method parameters?

We propose an example based code completion technique that collects parameter usage context from past examples. The technique only considers tokens close to the parameter position. The usage context in our case consists of any tokens (except identifiers, literals, braces and access specifiers) within the top four lines prior to the method call. When a developer requests for a parameter completion, the technique tries to match the current usage context with that of the collected examples. It then performs static type analysis to adapt the best matched example parameter in the current development context.

RQ3: Does the technique compare well with Precise and JDT?

Precise is the only state-of-the-art technique for recommending API method parameters. We thus compare our technique with Precise using two large subject systems. The results from the study suggest that our technique has strong potential and it can support more parameter expression types than Precise.

[1] `http://stackoverflow.com/questions/14124328`

We also investigate parameter recommendation support in Eclipse JDT. A large number of method parameters are in the simple name expression category, but Precise cannot recommend them. To compare with JDT, we focus our attention to parameters of the simple name category. We ignore others because JDT has very limited or no support for them. We show that the usage patterns of simple name parameters are also locally specific and their locality can be captured in a different way. Evaluation using parameters from two different API libraries and with two different subject systems shows the effectiveness of the new technique. The modified technique has been integrated with our parameter recommendation system.

Our contributions include:

1. A study that empirically validates that source code is locally specific to method parameters.
2. A technique that leverages source code localness property, static types and previous code examples to recommend method parameters.
3. An evaluation of our proposed technique with existing state-of-the-art tools using different subject systems and with different API libraries.

The remainder of this chapter is organized as follows. Section 5.2 describes related work. Section 5.3 describes important concepts related to the study. We verify whether source code is locally specific to method parameters in Section 5.4. We introduce our proposed technique in Section 5.5 and summarize the evaluation results in Section 5.6. We discuss some important issues about our work in Section 5.7. Section 5.8 discusses threats to the validity of our work. Finally, Section 5.9 concludes the chapter.

5.2 Related Work

The most relevant work to our study is that of Zhang *et al.* [144]. They propose a technique, called Precise, that mines existing code bases to generate a parameter usage database. A parameter usage instance consists of four pieces of information: (i) the signature of the formal parameter bound to the actual parameter, (ii) the signature of the enclosing method in which the parameter is used, (iii) the list of methods that are called on the variable used in the actual parameter, (iv) methods that are invoked on the base variable of the method invocation using the actual parameter. Given a method invocation and a parameter position, Precise identifies the parameter usage context in the current position and then looks for a match in the parameter usage database using the k-NN algorithm.

Precise has been evaluated using SWT library method parameters and Eclipse. Our study differs from theirs in a number of ways. First, Precise cannot recommend parameters of the following expression types: simple name, boolean, null literal, and class instance creation. Our proposed technique can not only recommend parameters of all the above four expression types, but also those that are supported by Precise. Second, we use a different approach to construct the parameter usage context compared to Precise that takes advantage of source code localness to method parameters. Simple tokenization suffices to collect the usage

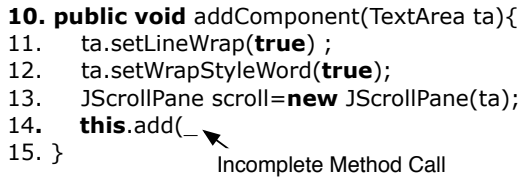


Figure 5.1: An example of a parameter completion query

context. Finally, we also investigate the parameter recommendation support of JDT, which was not explored in the previous study.

There are also a number of other code completion techniques available in the literature. They either use previous code examples or the static type system to recommend completion proposals. However, they all focus on method call completion instead of parameter completion. Bruch et al. [18] propose the Best Matching Neighbors (BMN) completion system that uses the k-NN algorithm to recommend method calls for a particular receiver object. Hou and Pletcher [47, 48] develop a code completion technique that uses a combination of sorting, filtering and grouping of APIs. Robbes and Lanza [113] propose a set of algorithms that use program history to recommend class and method names.

There are also a number of other techniques or tools that use previous code examples, but their goals are different than ours. Nguyen *et al.* [94, 92] use a graph-based algorithm to develop a code completion technique. While their technique focuses on automatic completion of API usage patterns, we focus on completing method parameters. Thung *et al.* [136] develop a technique that given a textual description of a feature request recommends API method names for implementing a feature. The technique leverages records of previous changes made to software systems. Hill and Rideout [44] develop a technique to support automatic completion of a method body by searching similar code fragments or code clones in a code-base. Mooty *et al.* [87] develop an Eclipse plugin, called Calcite, that helps developers to instantiate an object of a class.

5.3 Preliminaries

This section describes some important concepts related to our study. These include the parameter completion query, parameter expression types and the distribution of parameter expression types. We conduct an experiment to understand the frequency of different parameter expression types in software systems.

5.3.1 Parameter completion query

The term query indicates an incomplete method call without any parameters (see Figure 5.1 for an example). The goal is to complete the parameters in order to call the method. For each query, we know the receiver type, the method name and the set of tokens that appears prior to calling the method, but we do not know the actual parameter(s) to complete the method call.

Table 5.1: Examples of parameter expression types (highlighted in bold) and their distribution in three different subject systems

Parameter Expression Types	**Example**	**Distribution of Parameters in Different Expression Types**		
		JEdit	**ArgoUML**	**JHotDraw**
Array Access	button.add(**actions [i]**);	0.19	0.51	1.26
Array Creation	pd.setListData(**new Object[]{new LoadingPlaceholder()}**);	0.14	-	0.06
Boolean Literal	frame.setVisible(**true**);	5.53	7.36	4.26
Cast Expression	progress.setMaximum(**(int)max)**);	0.48	0.68	5.57
Character Literal	list.addChar(**'A'**);	0.43	-	-
Class Instance Creation	frame.setSize(**new Dimension(100,100)**);	11.66	11.44	7
Field Access	g2d.setRenderingHints(**painter.renderingHints**);	0.07	-	0.33
Instanceof Expression	field.setEditable(**e.getItem() instanceof GlobVFSFileFilter**);	0.02	0.06	-
Method Invocation	menubar.add(**Box.createGlue()**);	16.91	16.32	12.06
Simple Name	Dimension dim= new Dimension(100, 100); frame.setSize (**dim**) ;	40.69	39.08	36.77
Qualified Name	Container c = frame.getContentPane() c.add(new JLabel("Place",**BorderLayout.CENTRE**);	8.79	9.96	17.42
Null Literal	JOptionPane.showMessageDialog(**null**, message,JOptionPane.ERROR_MESSAGE);	0.69	1.22	1.69
Number Literal	buffer.setSize(**100**);	8.09	5.85	7.46
Parenthesized Expression	rectangle.setHeight(**(oldHeight*2)**);	0.19	0.45	0.61
String Literal	label.setText(**"Location"**);	3.52	1.92	4.67
This Expression	clipboard.getContents(**this**);	1.84	4.30	0.68
Type Literal	SwingUtilities.getAncestorOfClass(**EditPane.class**,ta);	0.71	0.80	0.48

5.3.2 Parameter expression types

An expression is a syntactic construction that can give us a value. Developers use a number of expression types to complete method parameters. Table 5.1 shows the list of parameter expression types with examples. More about these expressions can be found in the JDT documentation[2].

5.3.3 Distribution of parameter expressions

To determine the frequency of different parameter expression types we consider three different subject systems: JEdit,[3] ArgoUML[4] and JHotDraw.[5] For each system, we collect parameters from the API method calls of Swing and AWT libraries. Table 5.1 shows the distribution of parameters into different expression categories. Despite the difference in subject systems and API libraries, our finding is consistent with that of Zhang *et al.* [144]. The largest number of parameters fall in the simple name category (more than 36% in all three systems). The majority of the remaining parameters fall under the following three expression types: method invocation, qualified name and class instance creation. For JEdit, 8.79% of the parameters are of the qualified expression type and the number reaches 17.42% for JHotDraw. The method invocation expression type comes second for JEdit and ArgoUML (around 16% for both systems) and 12.06% for JHotDraw. The percentage of parameters for the class instance creation expression type ranges from 7% to more than 11%. Then come various literal expression types. In general, array access, cast expression, simple name, qualified name, method invocation, class instance creation, this expression and literal expression types (number, boolean, null and string literals) cover more than 98% of method parameters. Therefore, in this study we focus our attention on these eleven parameter expression types. We ignore others because they are difficult to autocomplete due to the complexity of the parameter expression types.

5.4 RQ1: Is Source Code Locally Specific To Method Parameters?

We say source code is locally specific to method parameters if tokens that appear in close proximity and prior to using method parameters favor them. Then there should exist a skewed probability distribution of those tokens when we group them based on the receiver type, the method name and the parameter position. To empirically determine this we follow the procedure described by Tu *et al.* [137] (we collect SWT API method parameters of the Eclipse system). The idea is to use the entropy measure from information theory to determine the probability distribution of those tokens using the following equation: $entropy = \sum_{i=1}^{k} -p_i \log(p_i)$.

[2] http://help.eclipse.org/juno/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/Expression.html

[3] http://sourceforge.net/projects/jedit/

[4] http://argouml.tigris.org/

[5] http://sourceforge.net/projects/jhotdraw/

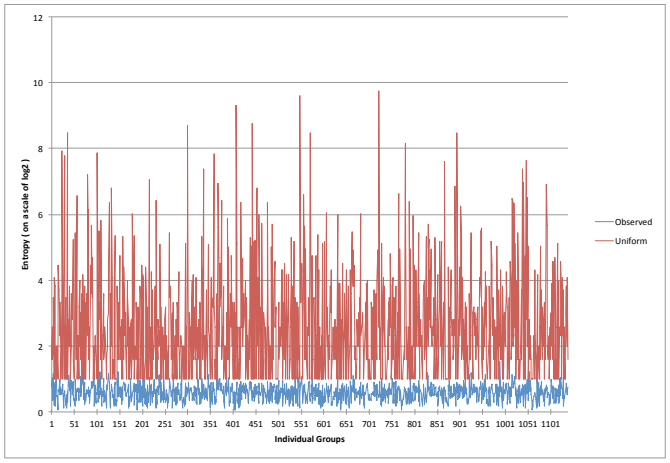


Figure 5.2: Mean entropy distribution for the top ten tokens when we group code examples based on receiver type, method name and parameter position.

Here, p_i is the probability of a token i that appears before a specific method parameter over all examples of that parameter position that is followed by token i. The more the entropy measure is close to zero, the higher the distribution will be skewed. On the contrary, the closer the value is to $\log_2 k$ (k is the number of examples of that parameter position), the more the probability distribution of those tokens will be uniform. For each method parameter, we determine the top-n tokens (in this experiment we set the value of n to 10) before calling the method and then group them based on the receiver type of the method call, the method name and the parameter position. For each group, we determine the entropy of each token using the above equation and then report the mean entropy value. Figure 5.2 shows the results along with the entropy measure for uniform distribution. We can see from the figure that the observed mean entropy values are much smaller than that of uniform distributions. This confirms that we can use locally specific tokens to recommend method parameters. It should be noted that we did not consider the method name in the top ten tokens because of its uniform probability distribution in each group of examples.

5.5 RQ2: How Can We Capture the Localness Property to Recommend Method Parameters?

To answer the above question, this section describes the proposed technique to recommend method parameters. Before discussing details of the technique we summarize challenges in recommending method parameters. First, static type systems cannot help much simply because there can be a large number of variables whose type matches the expected type of the parameter. There can also be a number of methods whose return type matches the expected type of the parameter. Furthermore, the type of an actual parameter can be a subtype of the formal parameter. Second, a parameter of a method can take expressions of different categories (such as simple name, method invocation, class instance creation etc.). Third, some parameters are difficult to

predict because of the high degree of variability or complex structures in its parameter usage examples which makes the recommendation challenging (such as postfix, prefix and infix expression types). However, they are only a few in number and we ignore them in this study.

To support automatic completion of API method parameters, we develop a technique that collects parameter usage instances from previous code examples to form a database, and we call this *PARC* (Parameter Recommender). When a developer requests for a parameter completion, our technique determines the requested parameter usage context and the candidate list. The list is then sorted using the similarity between the context of the requested parameter with that of parameters stored in the database. Finally, the top few candidates are recommended as completion proposals after removing duplicates. Each step is described in detail in the following section.

5.5.1 Building the parameter usage database

To build the parameter usage database, we walk through code examples to look for API method invocations. Then for each method parameter we collect the following pieces of information:

1. **Method Name**: The name of the method containing the parameter.
2. **Receiver Type**: The fully qualified name of the receiver type of that method.
3. **Parameter Position**: The position of the parameter in the method parameter list.
4. **Feature**: Features capture the localness property of method parameters. A feature set for a method parameter consists of a set of tokens that appears within the k lines prior to its method call and such that the lines are part of the method that contains the method call. The tokens we consider are method names, class names, interface names and keywords because these tokens show the most skewed probability distribution. After considering various values of k, we obtain the best performance of our technique using $k = 4$.
5. **Generic Representation**: The generic string representation of a parameter. The identifiers appeared in parameter examples can have different names, but they refer to the same parameter. The generic representation is used to identify duplicity in the parameter recommendations. To generate the generic representation, we replace any simple name that appeared in the parameter expression with its type qualified name. For example, consider the following method invocation where the parameter is a simple name: panel.add(button). The generic representation would be javax.swing.JButton. Consider another case where the parameter is a method invocation: tab.setText(button.getText()). The receiver type of the parameter method invocation here is javax.swing.JButton and the generic representation would be: javax.swing.JButton.getText.

The above items represent a parameter usage instance as shown in Figure 5.3. We use an indexing scheme where the parameter usage instances are indexed based on the method name, parameter position and receiver

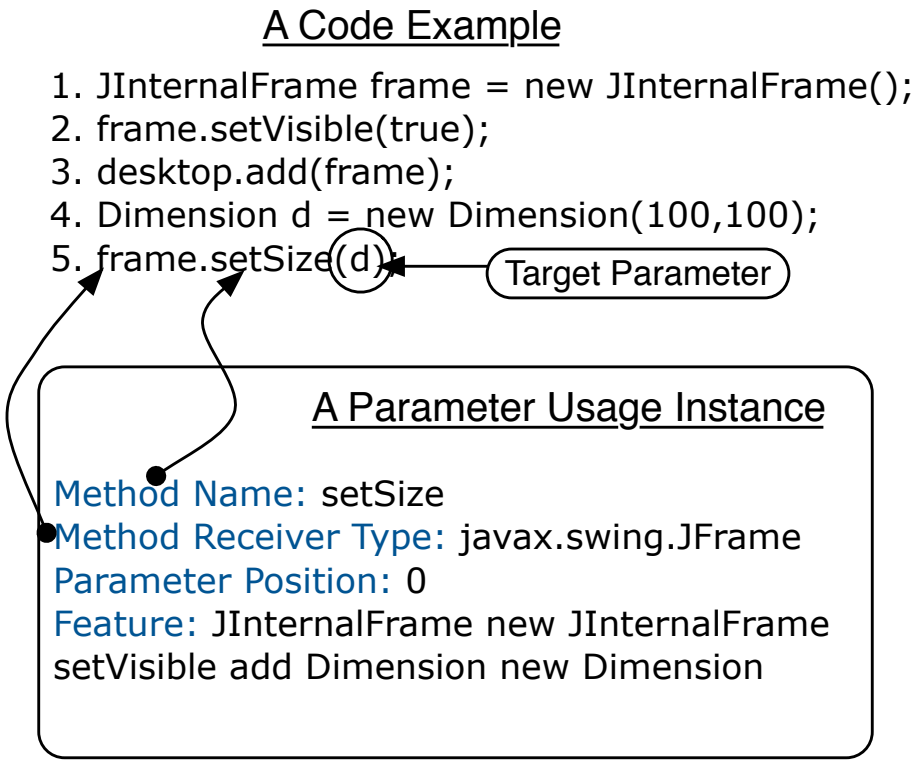


Figure 5.3: An example of a parameter usage instance

type. The usage context of a parameter is stored by concatenating all of the terms where any two consecutive terms are separated by a single space.

5.5.2 Collect features from a query

When a developer requests a method parameter completion (in Eclipse this can be done by pressing ctrl + space after writing the method name), *PARC* collects the same set of information it collected while building the parameter usage database, except for the generic representation of the parameter since we do not yet know the actual parameter. We refer to this information as a *query instance* and the corresponding feature as a *query feature*. We use the method name, receiver type and parameter position value of the query instance as an index to locate all previous parameter usage instances. We call this set the mapping candidates.

5.5.3 Determine feature similarity

For each mapping candidate and the query pair, we apply cosine similarity[6] to determine their feature similarity and then sort the mapping candidates based on the descending order of the similarity value.

[6] http://www-nlp.stanford.edu/IR-book/

If two mapping candidates have the same feature similarity with the query, we use the frequency of the mapping candidate to break the tie. Determining cosine similarity with all mapping candidates can be very time consuming. Therefore, we introduce an intermediate step. We use a form of locality-sensitive hashing to calculate usage context similarity using similarity preserving hash values instead of calculating textual similarity. Since the hashing technique converts a large string into a much smaller bit sequence, the similarity can be calculated very quickly.

We use the *simhash* algorithm [24] to generate 128 bit binary hash values, also known as *simhash* values. The technique has been found effective in detecting duplicate web pages [80]. The more similar two strings are, the smaller would be the Hamming distance between their *simhash* values. The Hamming distance between two binary values is equal to the number of ones in their bitwise exclusive OR operation. The smaller the Hamming distance is, the closer the two *simhash* values are. Therefore, after determining the Hamming distance we sort the parameter candidates in ascending order of distance value and select the top 500 parameter candidates, which we refer to as *likely parameter candidates*. We now determine the cosine similarity of the query context with each likely parameter candidate. We then sort the likely parameter candidates in descending order of similarity value and use their generic representation to remove any duplicates.

5.5.4 Static analysis and recommendation

At this point, we have a sorted list of parameters. Before making any recommendations we need to validate whether the parameter matches with the current context. For this reason, we use the following set of rules to create parameter recommendations:

1. If the parameter is a literal type, we directly use that for the recommendation.

2. If the parameter is a method invocation expression or a qualified name, we need to consider two different cases. If the receiver is empty or a type variable, we use the parameter without any change. However, if the receiver is a simple name, we first find all variables that are type compatible with the receiver variable and located within the scope of the query method call (see Section 5.6.2 for detail about how we generate this list of variables). We then replace the receiver with the topmost variable and insert the method invocation expression in the recommendation list.

3. If the parameter is a simple name, we search the query context to look for variables that are within the same scope as the query method call and type compatible with the formal parameter. This time we insert the top three variables in the list of recommendations. We limit the number to three, because there can be a large number of type compatible variables that reside within the scope of query context and we found the best result using that value.

After generating the recommendations, they are placed on top of the JDT completion proposals to present to the users. The number of recommendations generated by the *PARC* is configurable by users.

5.6 RQ3: Does the Technique Compare Well with Precise and JDT?

To answer the above question, we evaluate *PARC* with existing parameter recommendation techniques (this includes both Precise and Eclipse JDT) using two different subject systems: Eclipse 3.7.2[7] and NetBeans.[8] Both systems are large in size and have a long development history. Since our technique focuses on API method parameter completion, we use two different API libraries. For the Eclipse system, we collect all parameters of SWT library methods. For the NetBeans system, we collect method parameters of Java Swing and AWT libraries. Our selection of libraries is based on the fact that all these libraries are frequently used for developing applications. Thus, automatic parameter completion support for those API methods is more likely to help developers.

We apply the ten-fold cross validation technique to measure the performance of each technique [20]. First, we divide the entire data set into ten different folds. Next for each fold, we use code examples from the nine other folds to train the technique for parameter completion. The remaining fold is used to test the technique. To make the result comparable with Precise, the folds are generated based on classes. This means that all the parameter usage instances occurring in a class are either used together for training or for testing.

We provide parameter usage instances occurring in the training set to the parameter completion techniques for training. During testing, for each API method parameter we ask a code completion technique to generate completion proposals. The actual parameter used in the test set is hidden from the techniques, but the code appearing prior to that parameter is available to them. After generating the completion proposals, we check whether the target parameter appears within the top ten recommendations.

We use precision and recall to measure the performance which are defined as follows:

$$Precision = \frac{recommendations\ made \cap relevant}{recommendations\ made} \tag{5.1}$$

$$Recall = \frac{recommendations\ made \cap relevant}{recommendations\ requested} \tag{5.2}$$

Here, *recommendations made* is the total number of times a parameter completion technique recommends parameters. The term *relevant* refers to the total number of times the actual parameter is present in the top few recommendations. The term *recommendations requested* denotes the number of parameters in our test data.

5.6.1 Evaluation results

This section presents results of our evaluation. We were interested to see how *PARC* performs in predicting all eleven parameter expression types. Table 5.2 shows precision and recall values for two different subject

[7] `http://www.eclipse.org/`
[8] `https://netbeans.org/`

Table 5.2: Evaluation results of parameter recommendation techniques for all eleven parameter expression types *PARC* can detect

Subject System	Recom.	Precision (%)		Recall (%)	
		Precise	PARC	Precise	PARC
Eclipse	Top-1	11.75	47.65	11.07	46.65
	Top-3	15.26	65.05	14.38	63.68
	Top-10	18.45	72.26	17.38	70.73
NetBeans	Top-1	16.67	46.46	13.78	44.86
	Top-3	22.10	66.20	18.27	66.75
	Top-10	25.46	72.06	21,04	69.57

Table 5.3: Evaluation results of parameter recommendation techniques using only those parameter expression types that are supported by Precise

Subject System	Recommen.	Precision (%)		Recall (%)	
		Precise	PARC	Precise	PARC
Eclipse	Top-1	32.77	34.77	30.69	33.04
	Top-3	42.58	46.29	30.88	43.98
	Top-10	51.46	53.49	48.19	48.48
NetBeans	Top-1	25.82	51.69	26.06	49.30
	Top-3	34.23	70.99	34.55	67.71
	Top-10	39.42	78.38	39.79	74.75

systems and for eleven different parameter expression types. For Eclipse, *PARC* has a 47.65% precision for the top position and a 72.06% precision for the top ten positions. The recall value is also very high. For the top ten positions *PARC* achieves more than a 70% recall value. For the NetBeans system *PARC* shows consistent performance. For the top ten positions, precision is 72.06% and recall is nearly 70%. However, Precise did not perform well in this study. This is because Precise cannot detect parameters of the following expression types: simple name, null literal, boolean, and class instance creation. However, a large number of method parameters are of simple name and class instance creation expression types.

We only include Precise in this table to show that it cannot detect a large number of method parameters but they can be easily detected by *PARC*. Since JDT can recommend method parameters of simple name and the highest number of parameters fall in that category, it could easily achieve higher precision and recall value than Precise in this experiment. This can hide the important fact that such parameters are easier to detect and JDT cannot recommend any complex parameters that are detected by Precise. While one can combine JDT with Precise by appending completion proposals of JDT after the recommendations from Precise, there

are challenges in using them together. It may be the case that the actual parameter is a simple name, but Precise recommends other parameter expressions. Appending completion proposals after Precise indicates that we will definitely miss the parameter in the top positions. Thus, we exclude JDT from this experiment and focus on comparing with Precise only.

To determine how *PARC* performs considering only those parameter expression types that are supported by Precise, we conduct the experiment again. This time we consider the following parameter expression types: qualified names, method names, string literals, number literals, this expressions, array access and cast expressions. Table 5.3 shows the evaluation results. For both systems, *PARC* achieves better results than Precise. For the Eclipse system and for the top position, *PARC* obtains 2% better precision value and 2.35% better recall value. While for the top ten positions, *PARC* achieves 53.49% precision value, Precise obtains 51.46%. *PARC* also obtains slightly higher recall value than Precise. Although the performance of *PARC* is slightly better than Precise, these are the most difficult parameters to predict. Moreover, we exclude parameters of class instance creation type from this experiment that can be detected by *PARC*, but not by either JDT or Precise. For the NetBeans system, the relative improvements become more significant. For example, for the top position *PARC* achieves a 25.87% higher precision value and a 33.16% higher recall value compared to Precise. Even for the top ten positions, *PARC* performs significantly better than Precise. However, *PARC* provides more accurate recommendations and shows consistent performance across different subject systems. Table 5.4 shows the accuracy of correctly predicted parameters across different parameter expression types for the top ten recommendations for this experiment.

5.6.2 Exploring parameter recommendations of Eclipse JDT

Despite the fact that a large number of method parameters fall in the simple name category, Precise cannot recommend them. The default code completion system of Eclipse JDT can recommend those parameters. However, Eclipse JDT provides very limited or no support for recommending parameters for other expression types. In this section, we first summarize the parameter recommendation strategy of Eclipse JDT and then conduct an experiment to evaluate the technique with our proposed one for the simple name parameters only.

JDT collects local variables, parameters of the enclosing method, class variables (also known as fields) and inherited variables whose type match with the expected type of the target method parameter. We refer to this set of variables as the candidate set. Depending on the expected types of the method parameters, JDT also adds different literals to the candidate set. For example, if the expected type of the method parameter is boolean, boolean literals *true* and *false* are added. If the expected type is an object of a class, JDT adds *null* literal to the candidate set. Integer literal *0* is added when the expected parameter is a number. It then sorts the elements of the candidate set using the following sequence of rules:

- Local variables have higher priority than class variables and class variables have higher priority than inherited class variables,

Table 5.4: Accuracy of correctly predicted parameters for different expression types for the top ten recommendations

Subject System	Parameter Expression	Test Cases	Accuracy(%)	
			Precise	PARC
Eclipse	Qualified Name	470	42.12	43.82
	Method Invocation	405	47.16	46.67
	String Literal	33	39.39	36.36
	Number Literal	79	65.82	65.82
	This Expression	36	55.56	55.56
	Array Access	24	66.67	66.67
	Cast Expression	1	100	100
NetBeans	Qualified Name	860	26.86	93.02
	Method Invocation	490	29.59	54.28
	String Literal	85	47.05	50.64
	Number Literal	441	74.60	66.67
	This Expression	69	79.71	79.71
	Array Access	5	40.00	20.00
	Cast Expression	11	9.09	9.09

- A longer case insensitive substring matches of the variable name with that of the method formal parameter will prevail,
- Variables that have not been used have a higher priority than those that have already been used. This rule tries to avoid recommending the same variable to multiple method parameters, and
- The more closely a variable is declared to the method parameter position, the more its priority will be. Closeness is calculated using its location in the source code.

We randomly select a number of examples where the method parameter is a simple name and manually investigate them. We notice that developers tend to declare a variable in the same code block the method call is located and then use that as a method parameter. The more closer a type compatible variable is declared, the higher the possibility of using the variable as a method argument. That is why the first and the last rules are more effective than the other two rules. However, there are also a number of exceptions to this parameter usage pattern. We observe cases where developers declare a list of Component variables, initialize them and then add them to a container in the same order they are declared. The situation can be worse when there are a large number of variables that are type compatible with the parameter type. In that case, JDT fails to guess the correct method parameter within the top-3 positions.

We observe two interesting patterns. First, developers typically declare local variables at the beginning

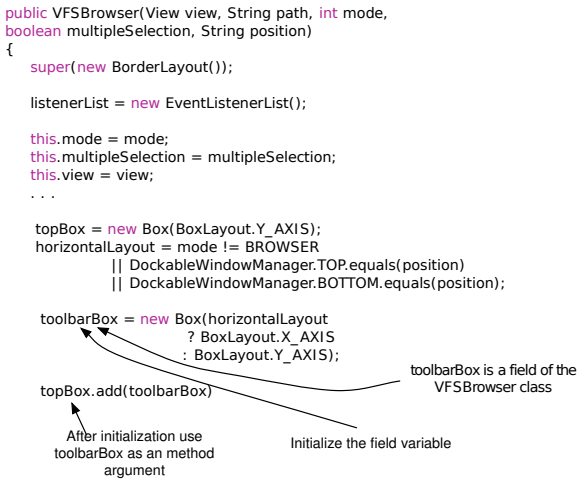


Figure 5.4: When recommending a method parameter, Eclipse JDT puts the local variables first. The field variables are positioned after the local variables and method parameters. Thus, in this case JDT will place the variable toolBarBox in the last position of the completion popup. However, instead of the declaration point, considering the initialization or most recent assignment point prior to calling the target method can help us to place the variable in the top position.

of method bodies and then initialize the variables just before using them as method parameters. Second, developers often initialize or assign a new value to a field variable and in the next statement use the variable as a method parameter (see Figure 5.4). Since, Eclipse JDT places field variables after local variables, it fails to place many field variables in top positions when they are the correct parameter. In both cases, instead of the declaration location, the recent initialization location of a variable can help us better predict the correct method parameter (see Figure 5.5).

We were interested to see whether the above findings can help us to improve parameter completion results. We change the sorting rules as follows (we refer to this as the modified approach): The closer a variable is declared, initialized or assigned new values to the method parameter, the higher the priority of the variable would be. Then there will be the unused parameters of the enclosing method, any unused class variables and finally, any unused inherited field variables. We use the term *unused* to refer to those class variables, inherited field variables and enclosing method parameters that are not initialized or assigned any new value prior to calling the target method in its enclosing method.

To evaluate our proposed sorting mechanism with that of the Eclipse JDT, we develop two different programs. Both programs parse each source file to identify the location of each method argument of simple name category and perform static analysis to generate the candidate set. However, they sort the candidate variables in two different ways. The first program imitates the sorting mechanism of JDT and the second program uses our proposed sorting rules. We identify the location of the target variable in the sorted list of candidates in both cases and record the results. For testing, we again use Eclipse 3.7.2 and NetBeans as subject systems. For the first system we consider parameters of SWT library method calls and for the second system we consider Swing/AWT library method parameters.

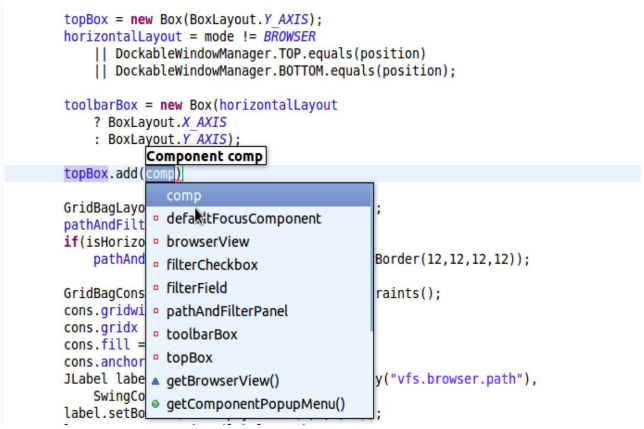


Figure 5.5: An example of parameter recommendations made by the Eclipse JDT for the topBox.add() method. The actual parameter toolbarBox is placed in the sixth position by JDT. When there are more type compatible local variables, JDT would place the toolbarBox parameter in a lower position. Our proposed technique places toolbarBox in the top position.

Table 5.5: The percentage of correctly predicted method parameters for the simple name expression category

Subject System	**Category**	**JDT** (%)	**PARC** (%)	**Relative Improvent**
Eclipse	Local	70	94.67	24.67
	Parameter	90.22	87.45	-2.77
	Field	62.78	66.55	3.97
NetBeans	Local	79.22	87.26	8.04
	Parameter	91.54	88.29	-3.25
	Field	24.33	29.93	5.60

If the usage context of a requested method parameter best matches that of a simple name parameter in code examples, *PARC* collects and sorts the type compatible variables using the modified approach described above. Next, it suggests the top most variable as a method parameter. Thus, it is important to improve the result for the top position. Table 5.5 shows the percentage of correctly predicted simple name parameters for the top position.

In general, our modified approach performs better than JDT for the top position. For example, for the local variable category, the relative improvement is 24.67% for the Eclipse system. We also observe improvement for field variables. Although our proposed change did not result in good results for the parameters, the relative improvement for the local variables is much higher than the relative performance decline for the parameter category. Moreover, the number of field variables and the number of cases developers use a field variable as a method parameter are much higher than the corresponding values for the parameter category. We also observe similar results for NetBeans.

5.7 Discussion

5.7.1 Why did Precise not perform well?

We further investigated why Precise did not perform well with the NetBeans system. One of the reasons that contributed to the poor results of Precise is that in many cases it only partially detects the target parameter. For example, if the expression type of the parameter is a method invocation, Precise detects the name of the method invocation correctly but fails to identify the receiver of that method invocation correctly or vice versa. Consider that Precise finds a match of the query context with that of a method parameter in the training code example, where the expression type of the parameter is a method invocation (i.e., frame.getContentPane()) with simple name as the receiver (the type of the receiver is javax.swing.JFrame). Precise collects all simple names of the same type within the scope of the query (consider there are three simple names: f, myFrame, frame), substitute the receiver with each simple name and recommends all of them (f.getContentPane(), myFrame.getContentPane(), frame.getContentPane()). However, it fails to determine which simple name in the current context is deemed correct for the receiver of the parameter expression. *PARC* on the contrary replaces the receiver with only that simple name that it finds most probable.

5.7.2 Runtime performance

The time required for a recommendation is an important concern of the usability of any code completion systems. We have measured the runtime of *PARC* in recommending completion proposals on a desktop computer equipped with a Core i7 CPU and 10 GB of memory. On average, *PARC* requires 45 milliseconds to recommend completion proposals, which is negligible. This indicates that *PARC* can be easily integrated with the Eclipse JDT code completion system and also leaves the opportunity to include addition type analysis. We are currently working on the implementation of *PARC* as an Eclipse plugin.

5.7.3 Cross-Project prediction

If developers want to use our code completion system at an early stage of their project, it would be difficult to train the system due to lack of code examples. This problem can be solved by using code examples from other projects. We were interested to find whether we can apply *PARC* in a project by training code examples from other projects. This is referred to as cross-project prediction. Cross-project prediction can be difficult due to the differences in project structure, development teams, and programming rules.

To perform cross-project prediction we use two different settings: (A) we train *PARC* using code examples from NetBeans, ArgoUML and JFreeChart for the SWT and Swing libraries. The test cases are collected from the JEdit system. (B) We repeat the same test but this time we allow code examples from JEdit (except those we use for testing) also to train *PARC*. As time passes and a project becomes mature, more code examples will be available. The second setting imitates this scenario. It should noted that we consider

Table 5.6: Cross-project prediction results of *PARC* under two different settings

Recommendatons	Precision (%)		Recall(%)	
	A	**B**	**A**	**B**
Top-1	14.52	35.49	14.01	34.57
Top-3	31.00	52.51	29.90	51.16
Top-5	38.74	59.47	37.38	57.94
Top-10	44.30	69.06	42.75	67.28

in this experiment all eleven parameter expression types *PARC* can detect.

Table 5.6 shows the result of our cross-project prediction. When *PARC* does not have any knowledge about the test project (Setting A), it achieves 44.30% precision and 42.75% recall value for the top ten positions. Reducing the number of recommendations also penalizes the performance considerably. For example, the precision and recall values are around 14% for the top position. When we checked the result, we found that this is due to the differences in the projects. Many of the parameter values are only specific to the JEdit project although many type compatible values exist. For example, in many cases developers use the following method invocation expression as the method parameter: *jEdit.getBooleanProperty(...)*, only specific to the JEdit system and our technique fails to recommend correct parameters in all those cases.

When we include code examples from JEdit for training (Setting B), we observe 100% improvement in the precision and recall values for the top position. For the top ten recommendations, the precision value is 69.06% and the recall value is 67.28%. This also indicates that if the training data contains parameter usage examples similar to the test cases, then *PARC* can detect them. When applying a parameter recommendation system at an early stage of a project, we possibly need to use projects similar to the test project and also add examples from the project as it grows in size.

5.8 Threats to Validity

There are a number of threats to this study. First, one can argue that the results may not generalize for other systems and for different libraries. We want to point to the fact that these are popular libraries and used by various software applications. The subject systems are large in size, have long development history and also used in various other studies [18, 48].

Second, there is no public API available to collect Eclipse JDT parameter completion results. We implement the algorithm used by JDT to guess parameter proposals. Although we cannot guarantee that there is no error in our implementation, we were very careful during the implementation of the algorithm. To avoid any error we have manually tested the results of our implementation with proposals made by JDT. We did not find any differences in the results during our manual inspection.

Third, there are a few parameter expression categories we ignore in this study (such as infix expression, postfix expression etc.). The reason is the variability in those expression categories that make them difficult to predict. Moreover, they represent only a small fraction of total parameters. It should be noted that our technique supports a large number of parameter expression categories compared to Precise.

5.9 Conclusion

Towards the goal of developing an automatic parameter recommendation system, we first conducted a study to learn how developers complete method parameters in practice. This helps us better understand parameter usage patterns. Based on our observation, we developed a technique, called *PARC*, that leverages source code localness property to capture the parameter usage context. The technique models the context by considering the four lines prior to the method invocation containing the parameter. Evaluation with a number of subject systems shows that *PARC* can recommend method parameters with consistently good performance. We also compared our proposed technique with Precise, the only available state-of-the-art parameter recommendation technique, and find satisfactory results. Moreover, *PARC* supports a large number of parameter expression types for recommendation compared to Precise. In addition, we also explored parameter recommendation support of Eclipse JDT and show a way to improve the recommendation of method parameters for the simple name category. The study reveals that the parameter usage context can be modeled with limited information rather than considering various different features that may not be always available. The code, data used in the experiment, and additional information can be found online.[9]

[9] `https://asaduzzamanparvez.wordpress.com/api_parameter_study/`

Chapter 6

Recommending Framework Extension Examples

During the study in the previous chapter, it was observed that developers pass framework-related objects as method parameters to customize the default behavior of software frameworks. This chapter presents a technique that leverages code completion systems to help developers to learn different choices to customize a framework and recommend framework extension examples. The study results described in this chapter and the implementation of the technique have been published in major software engineering conferences [9, 10].

6.1 Introduction

A software framework is a reusable implementation of some generic functionality that saves both development time and effort for a client. However, to meet application specific requirements, a developer often needs to customize aspects of the framework (*extension points*). One common way to do so is to pass a framework related object as an argument in an API call. The argument object essentially encapsulates a specific way to customize the framework. The object may be created by subclassing a framework class, implementing a framework interface, or by customizing the properties of an existing object. In this case, we consider the formal parameter of the API call as an extension point. As an example of an extension point, Figure 6.1(A) shows that the size of a *JFrame* is set to a Dimension object. Some additional, more interesting examples from *JTree* are *TreeModel*, *TreeCellEditor*, *TreeCellRenderer*, and *TreeExpansionListener*. These extension points allow a developer to gain finer control over the behavior of framework classes such as *JFrame* and *JTree*.

Given an extension point, there are often multiple different ways to use it. Consider *TreeCellRenderer* as an example, which controls how a tree node is rendered by passing it as an argument to the *setCellRenderer* method of the *JTree* class. There are three different ways to customize the cell rendering behavior of *JTree*:

1. One can call the *setCellRenderer* method with an object of the existing framework class *DefaultTreeCellRenderer*, which implements the *TreeCellRenderer* interface. One can change the cell rendering behavior by calling a set of methods on the *DefaultTreeCellRenderer* object prior to using that object as an argument (Figure 6.1(B)).

2. Alternatively, one can create a new class to implement the *TreeCellRenderer* interface and override the *getTreeCellRenderComponent* method (Figure 6.1(C)).

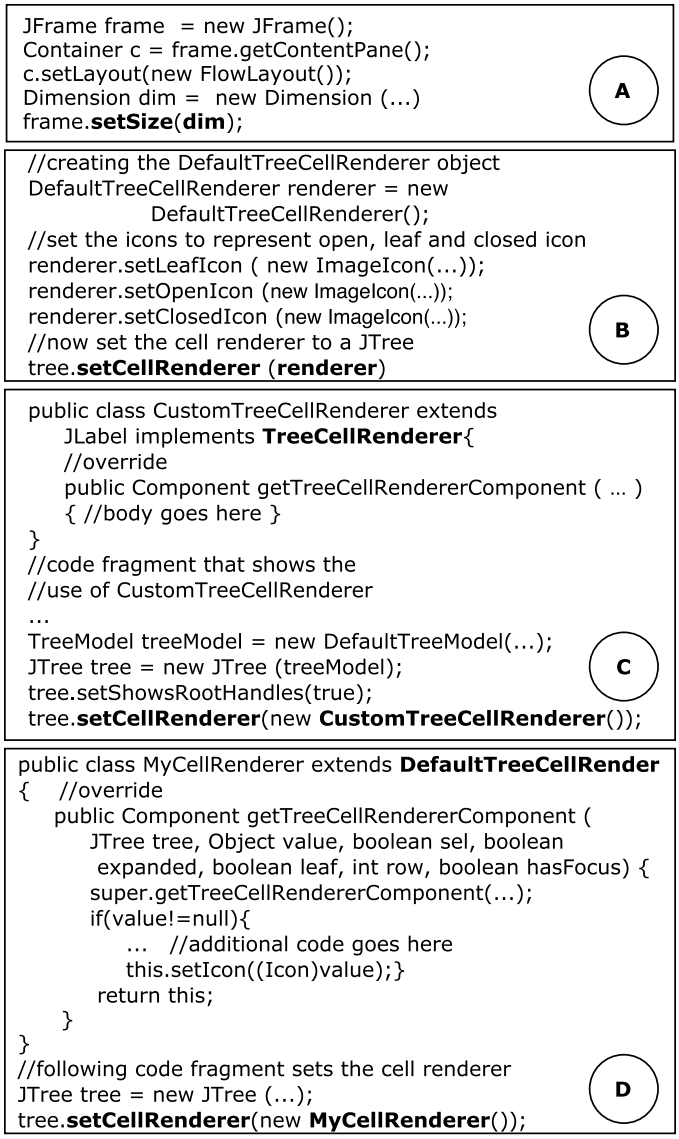


Figure 6.1: Examples of framework extension points (Dimension and TreeCellRenderer) and extension patterns

3. Or one can subclass *DefaultTreeCellRenderer* and pass it as the argument for *setCellRenderer* (Figure 6.1(D)).

By studying existing projects, a developer can discover many examples of how an extension point is used, however this can be quite time consuming. To address this, we propose to mine large code repositories and automatically locate examples of framework extension point use. We also propose a taxonomy of extension patterns to categorize the located usage examples. Furthermore, to utilize these mined examples, we develop a two-step recommendation system. A developer first selects an extension point related to the currently working with framework object. Once selected, the recommender then shows code examples of the relevant extension patterns for perusal.

We evaluate the efficacy of our two-step recommendation system using five different frameworks and

1,267 projects collected from GitHub. Our evaluation of the recommender uses the standard ten-fold cross validation. For the recommendation of top-5 extension patterns, the precision ranges from 78% to 90%, and the recall from 56% to 79%. Our evaluation indicates that our proposed technique is promising in supporting the use of extension points. Our approach is implemented as an Eclipse plugin called *FEMIR* (**F**ramework **E**xtension **Mi**ner and **R**ecommender), which is described in a complementary paper [9]. Thus, we make the following contributions:

- A taxonomy of framework extension patterns,
- A technique that combines syntactic analysis and graph-based mining algorithms to recommend framework extension points and their usage patterns,
- An evaluation of our proposed technique in terms of precision and recall, and
- A set of statistics on framework extension points.

The rest of the chapter is organized as follows. Section 6.2 presents related work. Section 6.3 defines a taxonomy of extension patterns. Section 6.4 presents our approach for mining and recommending extension patterns. Section 6.5 evaluates our approach, including its accuracy, a qualitative study of the quality of the recommended patterns, and statistics that characterize framework extensions. We discuss several questions related to our study in Section 6.6, and threats to validity in Section 6.7. Finally, Section 6.8 concludes the chapter.

6.2 Related Work

The most relevant work to our study is *XFinder* by Dagenais and Ossher [28]. *XFinder* requires developers to create guides as a sequence of steps for extending a framework. Given a code base, a guide, and a framework, *XFinder* locates examples that implement each step of the guide. In contrast, our work finds examples that illustrate different ways of using an extension point; it automatically finds extension points, categorizes their usage patterns, and locates code examples, but without requiring the guide. Moritz *et al.* developed a tool, called *ExPort*, that given a starting query API, presents a list of API methods that are considered similar to the starting API, based on relational topic modeling [88]. Upon selection of an API method, the technique shows code examples that illustrate how the selected method is used. However, the technique is not designed to discover framework extension points and their usage patterns.

Mining sub-classing directives: Bruch *et al.* proposed a technique that mines four sub-classing directives of frameworks [17]. These directives are pieces of documentation that describe the methods of a framework class that need to be overridden, super and framework methods that should be called inside an overriding method, and typical co-overridden methods of a framework class. Our technique captures not only sub-classing directive information but also how they can be used with other parts of the code

Table 6.1: Comparison of this work (FEMIR) with prior research on framework extension points (Y: well-addressed, P: partially addressed)

Supported Queries	**Pattern Extractor** [139]	**FrUit** [19]	**SpotWeb** [134]	**Core** [17]	**FEMIR**
Summary of all extension points for a framework class	P	P	Y	-	Y
Summary of frequent usage patterns of an extension point	-	-	-	-	Y
Examples that illustrate how extension points are used	-	-	-	-	Y
Extension points that are often used together	-	-	-	-	Y
Framework methods designed to be overridden	Y	Y	Y	Y	Y
Framework methods that are often overridden together	-	-	-	Y	Y
Super-implementations that should be called by overriding methods	-	-	-	Y	Y
Framework methods that should be called by overriding methods	-	P	-	Y	Y

(although implicitly through code examples). Thummalapenta and Xie [134, 135] developed a technique, called *SpotWeb*, that determines both frequently (hotspots) and rarely (coldspots) used framework classes and methods by leveraging code search engines. While *SpotWeb* can provide an overview of framework usage and identify extension points and the overriding directives, it does not find extension patterns and examples. Viljamaa proposed a technique using formal concept analysis to identify the typical ways in which framework classes and methods can be extended and overridden, respectively. The technique was implemented as a prototype tool, called *Pattern Extractor*. Michail [85, 86] developed a technique, called *CodeWeb*, that captures reuse relationships that an application must follow when extending a software library (such as which methods to override or call when extending a framework class). Bruch *et al.* [19] developed a technique, called *FrUiT*, that mines Java bytecodes to create framework usage scenarios on five class properties (extends, implements, overrides, calls, and instantiates). Patterns are identified by applying an association rule mining technique on those scenarios. However, none of the above techniques focus on finding framework extension points or different ways of interacting with an extension point.

Mining API usage patterns: Previous works on mining API usage patterns are related to our study because we also apply graph mining technique to locate framework extension patterns. For example, Acharya *et al.* [1] mine API usage patterns as partial orders, by analyzing static traces of source code. Zhong *et al.* [145] proposed an API usage pattern mining framework, called *MAPO*. The technique applies a clustering technique to group related API calls, generates method call sequences for each cluster and then applies a sequential pattern mining technique to discover frequent patterns from those sequences. However, *MAPO* produces many redundant patterns due to subsequences. To solve the problem, Wang *et al.* [140] developed a technique that uses a combination of closed frequent pattern mining approach and two-step clustering to find succinct and high coverage API usage patterns. Nguyen *et al.* [99] proposed *GrouMiner*, a graph-based approach that can mine frequent usage patterns involving multiple objects from source code. The patterns mined by these techniques are typically located in one method, whereas our technique focuses on finding different patterns of using extension points that often span across multiple classes, methods, and files.

Searching code examples: Code search techniques are related in that they also focus on finding code examples. A number of techniques have been proposed in the literature. These include but not limited to *Strathcona* [46], *PARSEWeb* [133], XSnippet [123], and the internet-scale code search engine proposed by Keivanloo *et al.* [62]. However, they are not designed to locate framework extension examples. There are also code search techniques that can find examples from a task description [25, 84, 130]. However, none of these techniques focuses on discovering framework extension points.

There are also a number of other studies that aim to improve framework reuse. These include the work on documenting frameworks using patterns [56] and proven solutions to the frequently appearing challenges in understanding frameworks [33, 49], developing concept implementation templates [43], creating cookbooks containing feature recipes [37, 72], and creating API critics [121]. However, the objectives of these studies are different from ours.

6.3 Taxonomy of Extension Patterns

Extension patterns are common ways of using an extension point. To help manage extension patterns, we categorize them into four broad categories as follows. The taxonomy is inspired by observation on the common ways how an extension object is created and customized in code.

6.3.1 Simple

A *simple* extension pattern passes an argument object of a framework class to an extension point, without any further customization on the object. This pattern does not require extending a class or implementing a framework interface. Figure 6.1(A) shows an example of the simple extension pattern for the *Dimension* extension point, which is the formal parameter of the *setSize* method of the *JFrame* class. Notice that if there exist multiple framework classes that can be used as argument types, multiple *simple* patterns may exist for the same extension point.

6.3.2 Customize

Developers often need to call a set of methods on the argument object of a framework class to customize its behavior. Such an extension pattern belongs to the *customize* category. As an example, Figure 6.1(B) shows such a pattern for the *TreeCellRenderer* extension point, where four methods are called on the *DefaultTreeCellRenderer* object before it is passed to *setCellRenderer*. Notice that if there exist multiple framework classes that can be used as argument types, multiple *customize* patterns may exist for the same extension point.

6.3.3 Extend

In an *extend* pattern, a new class is created to customize an extension point by extending a framework class. Optionally, additional method calls may be made on the argument object. Figure 6.1(D) shows an example of this pattern for the *TreeCellRenderer* extension point. In this example, a new tree cell renderer is created by extending the existing *DefaultTreeCellRenderer* class. Notice that if there exist multiple framework classes that can be used as argument types, multiple *extend* patterns may be found for the same extension point.

6.3.4 Implement

An *implement* extension pattern occurs when the extension point is a framework interface. To customize the extension point, a client class implements the interface. Its object is then used as argument. Optionally, additional method calls may be made on the argument object. As an example, consider the *TreeCellRenderer* interface in Figure 6.1(C). To customize the cell rendering behavior of a *JTree*, a new class implements the

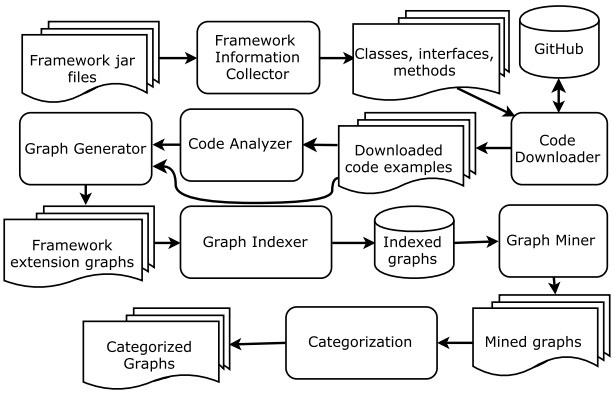


Figure 6.2: Working process of the graph miner of FEMIR

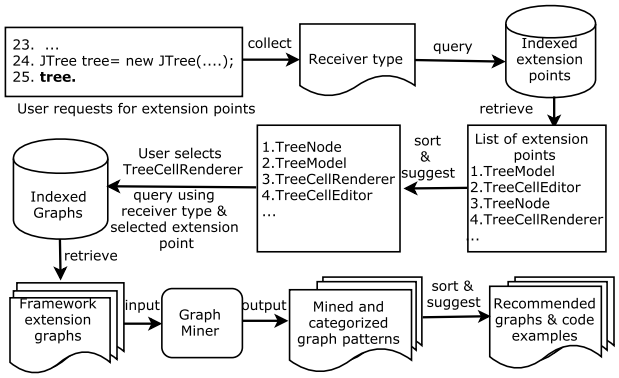


Figure 6.3: Overview of the extension patterns recommender of FEMIR

getTreeCellRendererComponent method of the *TreeCellRenderer* interface. An object of the new class is then passed to the *setCellRenderer* method of the *JTree* class.

6.4 Technical Description

Our approach for mining and recommending examples of framework extensions consists of two components, a *graph miner* and a *recommender*, whose working processes are summarized in Figures 6.2 and 6.3, respectively. The graph miner is responsible for mining and organizing the usage patterns for a framework extension point. Upon a developer's request for help on a selected extension point from a class, the recommender displays a set of code examples to illustrate all of its relevant extension patterns.

6.4.1 Miner

In this section, we briefly describe the seven components shown in Figure 6.2 of our graph miner.

Framework Information Collector

This component accepts a framework jar file as the input and collects information on framework classes, interfaces, and methods. For each class or interface, we collect its name, super classes, implemented interfaces,

and the list of methods. For each method, we collect its name, return type, and types of parameters.

Code Downloader

This component collects open source software projects hosted on GitHub[1] that contain framework usage examples. GitHub hosts a large number of open source Java projects. It also provides APIs to search for code examples. We search repositories using the import statements in Java source files. For example, to identify repositories that use the *Swing* API, we use the following query: *import AND javax AND swing*. After collecting repository information, the technique downloads source code examples.

Code Analyzer

The code analyzer performs static analysis of source code to identify framework extension points and enable the construction of framework extension graphs. It identifies type declarations (classes or interfaces) that are created from framework types, determining their super classes, implemented interfaces and overridden methods. It also identifies those method calls where a receiver is a framework type or a sub-type, and resolves type bindings of both receivers and arguments. The Eclipse JDT parser is used for parsing and we use partial program analysis to resolve type bindings [27].

Framework Extension Graph Generator

At the core of each framework extension graph is a method call that represents a use of a framework extension point. To be counted, the method call must have at least one parameter that is related to a framework type. A framework extension graph thus consists of one or more of the following node types:

- **Receiver type:** A node representing the receiver type of a method call or the class in the case of a constructor call.
- **Method call:** A node representing a method or constructor call.
- **Parameter type:** A node representing the type of a parameter that is either a class or an interface.
- **Argument type:** A node representing the declared type of a method call argument.
- **Other receiver method calls:** All other framework method calls on the receiver variable, including the construction call that creates the variable.
- **Other argument method calls:** All other framework method calls on the argument variable.

If the receiver or the argument of a call is of a client type that extends a framework type, we collect information on the extended classes, implemented interfaces and overridden methods. Thus, a framework extension graph can also contain the following kinds of nodes:

[1] https://github.com/

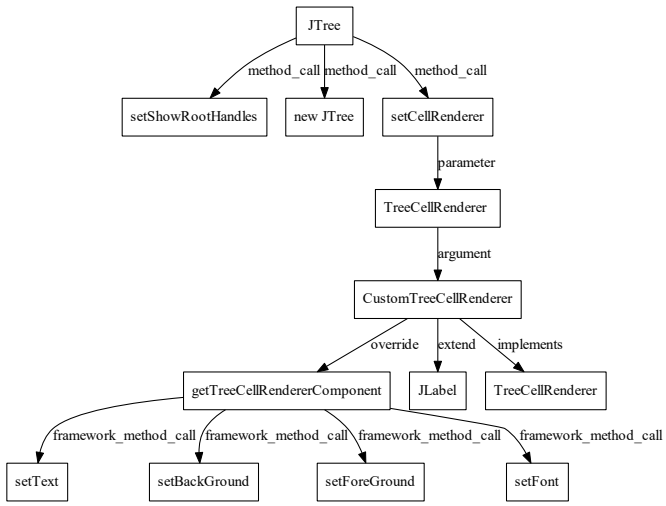


Figure 6.4: Framework extension graph for the example in Fig. 6.1(C))

- **Extended class:** A node representing the parent class in the inheritance hierarchy.
- **Implemented interface:** A node representing an implemented interface.
- **Overriding method:** A node representing a method that overrides a framework method. We collect the method name, return type, parameter types, and the type that declare the method.
- **Super method call:** The super method called by an overriding method.
- **Framework method calls:** A set of framework methods that are called by an overriding method.

The extension graph shown in Figure 6.4 illustrates the various types of nodes introduced above.

Graph Indexer

To enable future retrieval, the extension graphs are indexed by the respective framework-related types of the receiver, the formal parameter, and the argument.

Graph Miner

Given an extension point for a framework class, the goal for the graph miner is to identify the frequent patterns for extending the framework class by generating and counting subgraphs. Given a set of n graphs (also known as base graphs) that belong to the extension point, the miner iteratively and incrementally generates all of their subgraphs as follows. In the first step, it generates all one-node graphs for each base graph. In each subsequent step, the generated subgraphs are counted by comparing their canonical forms (see below). The top-k frequent subgraphs are kept as the starting point for the next step. In the next step, these top-k subgraphs are grown by adding an adjacent node from the base graph. As a result, a new set of subgraphs are generated. The process continues until all nodes of the base graphs are exhausted.

Node Label	Node index	Neighbor Node Index	Out Degree
JTree	101	21, 35, 40	3
setShowRootHandles	21	-	-
new JTree	35	-	-
setCellRenderer	40	67	1
TreeCellRenderer	67	55	1
Client	55	71, 77, 99	3
getTreeCellRendererComponent	34	30, 45, 87, 201	4
JLabel	99	-	-
TreeCellRenderer	77	-	-
setText	201	-	-
setBackGround	87	-	-
setForeGround	45	-	-
setFont	30	-	-

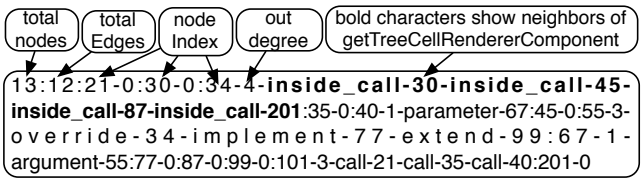


Figure 6.5: Canonical form of the graph shown in Fig. 6.4. Each node is represented by an index, an out degree and a list of neighbor nodes, separated by hyphens. Each neighbor node is represented by an edge label (e.g., inside_call) and its index. Nodes are sorted by index and separated by colons.

The miner categorizes the graph patterns generated above into the extension pattern categories as defined in Section 6.3. For example, consider the extension patterns for the *TreeCellRenderer* extension point. If a candidate pattern contains an argument node that is created by extending *DefaultTreeCellRenderer*, we assign it to the *extend* category. In contrast, if the argument is an object of the framework class *DefaultTreeCellrenderer* with a set of additional methods called on it, we assign the graph in the *customize* category. Notice that due to the existence of nodes such as argument and receiver calls, more than one pattern may be generated for each extension pattern category. For each category, we pick the top-n candidate graph patterns with the highest frequencies. We call them *mined graph patterns*.

However, since the mined graph patterns are subgraphs generated from the base graphs, they may not be ideal for representing a pattern category because they may miss some essential nodes in the base graphs. Thus, the miner further improves a mined graph pattern p as follows. It first determines the *support* of those nodes that are present in base graphs that contain p but not in p itself. The support of such a node is defined as the ratio of the number of base graphs that contain the node over all the base graphs. If the support for a node exceeds a predefined threshold value δ, it is added to the graph pattern. During our experiment, we find that 0.30 is a good value to work with.

To represent and compare graphs, the graph miner uses an approximated canonical form. To answer the question, we need to determine whether two graphs are isomorphic to each other or not. Although there are practical algorithms for testing graph isomorphism, they are computationally expensive [143]. As an alternative, we use their canonical forms [13], because the canonical forms of two isomorphic graphs are identical. Unfortunately, determining the canonical form of a graph is also computationally expensive [99].

Therefore, we approximate the canonical form by using graph invariants, which are structural properties of a graph. Specifically, we create the canonical form of a graph by concatenating the following graph invariants into a string:

1. The total number of nodes in the graph.
2. The total number of edges in the graph.
3. The list of nodes ordered by index.
4. Each node is represented by an index, out degree, and a list of neighbors ordered by index.

A node index is calculated for each graph node by hashing a string concatenating its out degree, name, and node type. Figure 6.5 shows an example on how the canonical form is calculated for the framework extension graph of Figure 6.4. Lastly, to enable comparison, all client classes derived from a framework type (such as the *CustomTreeCellRenderer* in Figure 6.4) are represented using the same label *client*.

An analysis of the computational complexity follows. To mine m base graphs each containing n nodes, the graph miner would need to generate a total of $O(m * 2^n)$ subgraphs. While the value of m can be very large for a large repository, due to the practice of writing shorter methods, on average, the value of n is not. To control the number of generated subgraphs, we limit the maximum size of the candidate graph patterns to 20. Each step of the graph mining process passes on only the top k frequent candidate patterns to the next step. In our experiment, we choose $k = 500$ because we believe that practically no extension point would have more than 500 interesting patterns.

As shown in Section 6.5.2, our qualitative analysis indicates that the current version of FEMIR is able to mine useful patterns. However, it remains as future work to analyze exactly how the maximum size on mined graph patterns impacts the quality of the extracted framework extension patterns.

6.4.2 Recommender

FEMIR recommends the most likely patterns for each extension point that a developer asks for help. To learn how to extend the functionality of a framework class, the developer requests the help from FEMIR by typing a dot after a variable of the class. FEMIR first shows the developer the list of extension points that are applicable to the class. Once the developer selects an extension point, FEMIR then shows multiple patterns for each of its extension categories.

Specifically, FEMIR first retrieves the subset of extension graphs that belong to the extension point from the set of all framework extension graphs that are built by the miner previously. To do so, it utilizes the names of the framework class and the selected extension point as indices. FEMIR then utilizes the graph miner described above to generate and group the graph patterns by extension pattern categories. FEMIR sorts the patterns belonging to each category based on their usage frequency in the training data. Finally, FEMIR recommends the top-n patterns for each category for the developer's perusal. Once the developer selects a recommmended pattern, FEMIR shows the actual code examples that contain the pattern.

Table 6.2: Summary of framework extension dataset used in this study

Framework	Classes	Methods	Projects	Files	LOC
Swing	466	10,251	263	107,380	9.94 M
JFace	524	7880	300	151,523	11.20 M
JUnit	121	944	242	123,220	10.50 M
JGraphT	201	1493	295	82, 621	9.50 M
JUNG	342	3306	167	76,376	8.80 M

6.5 Evaluation

In this section, we first evaluate the performance of FEMIR in recommending code examples. To further illustrate FEMIR's capability qualitatively, we also present a set of extension patterns that are recommended by FEMIR. Lastly, we present a set of statistics to characterize framework extensions.

We choose five different open source frameworks for our evaluation: *Swing*, *JFace*, *JUnit*, *JUNG*, and *JGraphT*. These frameworks are widely used for application development. They are also used in prior research [134]. Table 6.2 summarizes the five frameworks used in this study, including the numbers of projects and source files analyzed. FEMIR is realized as an Eclipse plug-in. All experiments are conducted on a machine with INTEL Core i7 CPU (2.93 GHz), 16 GB RAM.

6.5.1 Accuracy of FEMIR recommendation

This section presents an experimental study to understand the accuracy of our proposed technique FEMIR. To evaluate FEMIR, we assume a scenario where a developer decides to extend the functionality of a framework class, but does not know how to do that. She requests the help from FEMIR after declaring a variable of that framework class. This is done by typing a dot(.) after the variable name. FEMIR first shows the names of a list of extension points that the developer could use to extend the functionality of the framework class. After she selects an extension point, FEMIR shows the different patterns of using it. Given an extension point, we thus evaluate the effectiveness of the technique in recommending a framework extension graph that matches with the actual usage of the extension point.

Evaluation procedure and metrics

We develop an automatic evaluation system that analyzes the source code files and collects framework extension graphs from the subject systems for each framework. Thus, for each framework variable v, we know the fully qualified type name of the variable (t_v), the extension points used ($e_1, e_2, e_3, ..., e_n$), and the framework extension graphs ($g_1, g_2, g_3, ..., g_n$) that illustrate how those extension points are used in the code.

Each extension is considered as a data point in our evaluation. We apply the ten-fold cross validation technique to measure the performance of FEMIR. Specifically, we divide the data set into ten folds, each

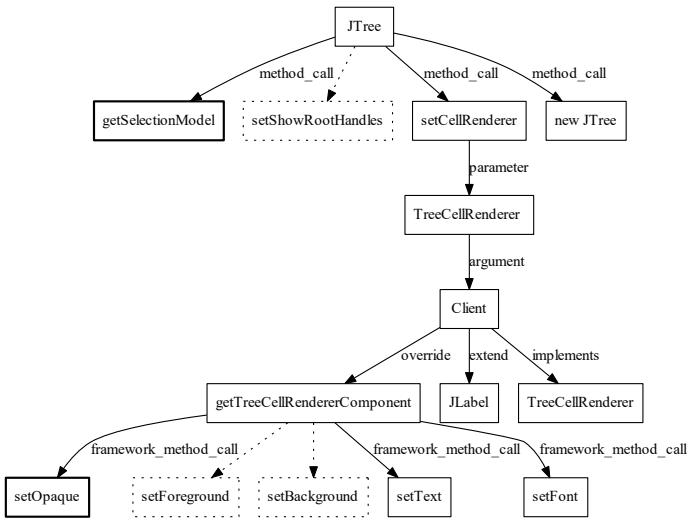


Figure 6.6: One extension pattern recommended for Figure 6.1(C), where missing nodes are shown in bold and incorrect nodes are shown in dotted rectangles.

containing an equal number of extensions. Next, for each fold, we use code examples from the nine other folds to train FEMIR for recommendation. The remaining fold is used to test the performance of the technique. For each test data point, FEMIR recommends the top-n extension graphs for each extension category to show the different ways of using the extension point.

Precision and recall are calculated as follows. Let O denote the original graph under testing and S the graph suggested by FEMIR. Because S can be useful even if it is not identical to O, we do not simply determine whether S and O are identical or not. Instead, we determine the common nodes shared between these two graphs. We use precision, recall and F-measure metrics to measure the performance, which are defined as follows. If a node in O occurs in S, we consider it a correctly recommended node (increasing precision). On the contrary, if a node in O does not occur in S, we consider it a missing node (lowering recall).

$$Precision = \frac{total\ correctly\ recommended\ nodes}{total\ recommended\ nodes} \tag{6.1}$$

$$Recall = \frac{total\ correctly\ recommended\ nodes}{total\ recommendation\ needed\ nodes} \tag{6.2}$$

$$F - measure = 2\ .\ \frac{precision\ .\ recall}{precision + recall} \tag{6.3}$$

To further illustrate the calculation of precision and recall, consider the example depicted in Figure 6.1(c), where a new class implements the *TreeCellRenderer* interface. Suppose this is what a developer eventually wants to create but does not know how initially. So the developer requests FEMIR to help. FEMIR then suggests the list of extension points of *JTree*, such as *TreeCellRenderer*, *TreeNode*, and *TreeModel*. Suppose that the developer selects the *TreeCellRenderer* extension point. In response, FEMIR recommends the top-n graph patterns from each category, for the developer's perusal. To calculate precision and recall, FEMIR compares the test case with the top-n graphs from the same category. It reports the precision and recall from

Table 6.3: Evaluation results of recommending framework extension graphs (RMC: Receiver method call, AMC: Argument method call, E: Class extension, I: Interface implementation, FMC: Framework method call, O: overridden method, Other: Other node types)

Framework	Technique	RMC AMC	E I	O FMC	Other	Precision			Recall			F-Measure		
						Top-1	Top-3	Top-5	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
Swing	FEMIR-Global	30%	6%	14%	50%	0.89	0.87	0.85	0.66	0.76	0.79	0.76	0.81	0.82
	FEMIR-Local					0.89	0.89	0.89	0.71	0.72	0.73	0.79	0.79	0.80
	FEMIR-D					0.20	0.41	0.42	0.31	0.67	0.73	0.24	0.51	0.53
JFace	FEMIR-Global	12%	6%	26%	56%	0.76	0.78	0.78	0.51	0.61	0.64	0.61	0.68	0.70
	FEMIR-Local					0.81	0.82	0.82	0.44	0.48	0.49	0.57	0.60	0.61
	FEMIR-D					0.25	0.34	0.43	0.37	0.56	0.63	0.30	0.42	0.51
JUnit	FEMIR-Global	17%	5%	12%	66%	0.90	0.90	0.90	0.60	0.67	0.69	0.72	0.77	0.78
	FEMIR-Local					0.89	0.92	0.92	0.61	0.68	0.70	0.72	0.78	0.80
	FEMIR-D					0.64	0.70	0.70	0.61	0.70	0.71	0.63	0.70	0.71
JGraphT	FEMIR-Global	31%	7%	5%	57%	0.86	0.85	0.85	0.41	0.53	0.56	0.55	0.65	0.67
	FEMIR-Local					0.86	0.87	0.87	0.42	0.52	0.55	0.57	0.65	0.67
	FEMIR-D					0.72	0.75	0.76	0.40	0.61	0.64	0.52	0.67	0.70
JUNG	FEMIR-Global	35%	6%	5%	54%	0.87	0.86	0.86	0.59	0.67	0.71	0.70	0.76	0.78
	FEMIR-Local					0.88	0.88	0.88	0.61	0.67	0.70	0.71	0.76	0.78
	FEMIR-D					0.56	0.70	0.71	0.51	0.67	0.70	0.56	0.68	0.70

the graph that yields the best F-measure. Figure 6.6 shows the best graph recommended by FEMIR, where missing nodes are highlighted in bold, and incorrect nodes are highlighted using dotted rectangles. Thus, the precision and recall for this recommendation are 10/12 and 10/13, respectively.

We refer to the strategy described above as the *local* strategy because it recommends the top-n patterns from within the same category, which is denoted as FEMIR-Local in Table 6.3. An opposite strategy is to recommend the global top-n patterns regardless of their categories. We denote the global strategy as FEMIR-Global in Table 6.3.

Alternative strategies for maximizing accuracy

Recall that when recommending framework extension graphs (Section 6.4.1), FEMIR first mines the most frequent graph patterns for each extension pattern category. To improve accuracy, it then applies a greedy strategy to add more nodes to each graph pattern that are deemed important but missed during the mining process. To further explore other options, we compare the greedy strategy with an alternative that is called the diversity strategy.

The diversity strategy works as follows. After determining the top-n graph patterns of an extension pattern category, we determine all the base graphs in the training data that contain the patterns. Among these base graphs, we recommend the ones that contain the largest number of different node types. We refer to this strategy as the diversity strategy, and denote it as FEMIR-D in Table 6.3.

Evaluation results

Table 6.3 shows the precision, recall, and F-measure values for recommending target framework extension graphs. Furthermore, columns three to six show the percentages of different kinds of nodes in the test cases. The largest number of nodes are of *other* type. Together *RMC* (receiver method calls) and *AMC* (argument method calls) represent the second largest group of nodes. Although the percentages of E, I, O and FMC nodes are much smaller in number, they would be the most difficult to use because of the inheritance structure and limited usage examples.

In general, FEMIR-Global performs well in all these test cases, with precision ranging from 78% to 90% and recall 56% to 79% for the top-5 recommendations. FEMIR-Local performs close to FEMIR-Global. While the precision of FEMIR-Local is slightly better, ranging from 82% to 92%, the recall ranges between 49% to 73% for the top-5 recommendations, which are slightly lower than FEMIR-Global. More investigation will be needed to explain the difference between FEMIR-Local and FEMIR-Global.

The result of FEMIR-D is not better than FEMIR-Global either. In fact, the precision and recall values are lower than FEMIR-Global in all cases except for *JGraphT*, where we observe that FEMIR-D has slightly better F-measure than FEMIR-Global.

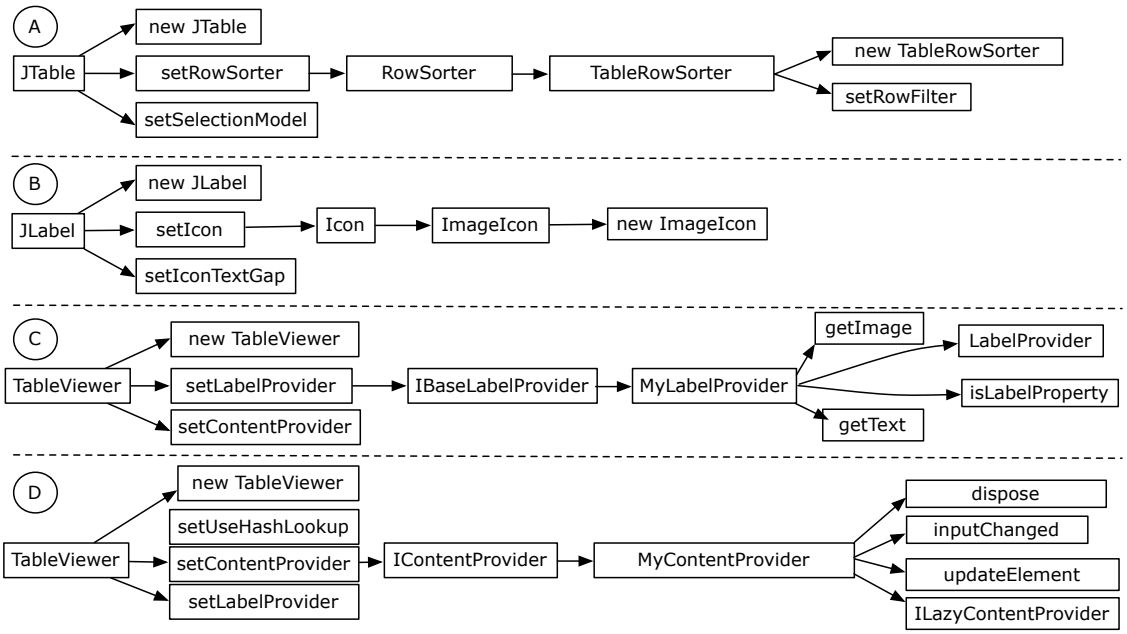


Figure 6.7: Example framework extension patterns mined by FEMIR: A *customize* extension pattern for the *RowSorter* extension point (A), a *simple* pattern for *Icon* (B), and two *extend* extension patterns for *IBaseLabelProvider* (C) and *IContentProvider* (D), respectively.

6.5.2 Examples of framework extension patterns

In this section, we report a qualitative evaluation of the quality of the patterns recommended by FEMIR. We consult tutorials[2,3] (both official and community-based) to find a set of relevant patterns. Our goal is to check whether FEMIR is able to identify these patterns by mining code bases. We focus on Java *Swing* and *JFace* due to our familiarity with them.

TableRowSorter

Tables in Java Swing support sorting and filtering capability. To do so, it is required to pass an instance of *TableRowSorter* to the *setRowSorter()* method. However, to gain more control, one can override *TableRowSorter* or its parent class *DefaultRowSorter*. Figure 6.7(*A*) depicts an example pattern mined by FEMIR for using *TableRowSorter*. FEMIR is able to mine patterns for controlling table sorting in all three different ways. Furthermore, a developer can provide a filter object by calling *setRowFilter()* method on the sorter object to control which rows will be displayed. Our mined pattern is able to collect that information too. All of these indicate that FEMIR is able to show different ways of using an extension point, providing the developer the opportunity to select the one that best matches with her goal.

[2] `https://wiki.eclipse.org/Eclipse_Corner`
[3] `https://docs.oracle.com/javase/tutorial/uiswing/`

Icon

Many swing components can be decorated with an icon, a small fixed size picture. Figure 6.7(B) shows a pattern where an icon is used jointly with a *JLabel*. The pattern also shows that developers frequently call other methods to set the properties of an icon. An alternative to this pattern is to implement the Icon interface to create a custom icon (not shown). FEMIR is able to mine both patterns. Again, this example shows that FEMIR can collect all popular ways to complete a task and allow the developer to select the option that best fits her needs.

LabelProvider

A label provider allows viewers to customize the display of labels. By default, a label provider uses the element's *toString* value to display text and null for image. Figure 6.7(C) shows a pattern of using a custom *LabelProvider* for the *TreeViewer* component. It also shows that creating a label provider typically involves sub-classing the *LabelProvider* class and overriding the following methods: *isLabelProperty*, *getImage*, *getText*, and *dispose*. This matches with information provided in the documentation of the JFace. However, mined patterns often contain more important details than framework documentation. Thus, results mined by FEMIR could complement documentation as a developer aid.

ContentProvider

Eclipse *JFace* viewers support a content provider that establishes the connection between the data and the viewer. Figure 6.7(D) shows a pattern of creating a content provider in order to customize a *TableViewer*. The pattern also shows that when using the content provider, developers also set a label provider by calling the *setLabelProvider* method, which is another extension point for the *TableViewer* (Figure 6.7(C)). Thus, this example illustrates that multiple extension points may be used together and a pattern of using one also helps to discover others.

6.5.3 Distribution of extension patterns by categories

Further to the evaluation, we want to see the distribution of extension points by categories using the five frameworks. As shown in Figure 6.8, the distributions are similar across all five frameworks. The most popular category of framework extension patterns is the *customize* usage pattern, which by definition requires a developer to call a set of methods on the method argument. The *simple* category, which does not require extending a framework type, or using a variable with several call sites, is the second most popular. Lastly, although the *extend* or *implement* categories appear to be minor, they require more efforts to learn and use than the first two categories.

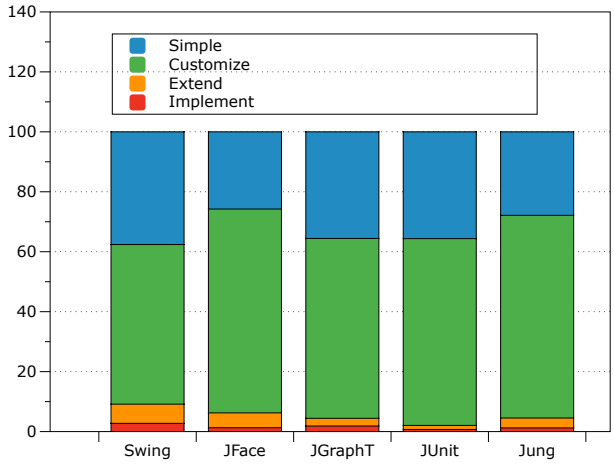


Figure 6.8: Distribution of extension patterns by categories

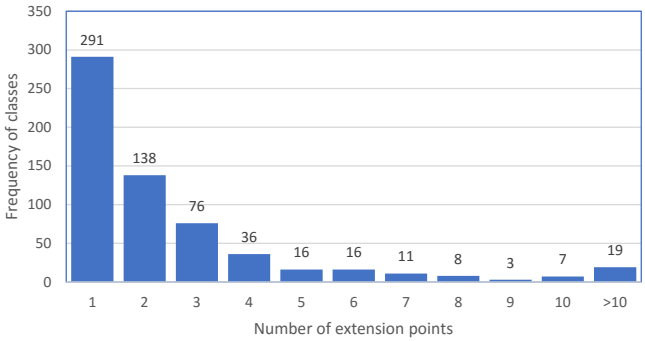


Figure 6.9: Distribution of extension points across classes (Swing framework)

6.5.4 Distribution of extension points by classes

To understand how usable it is to show extension points to developers, Figure 6.9 depicts the distribution of extension points across framework classes for Java Swing. Only 4% of the classes have 10 or more extension points. However, this small fraction of framework classes appears to be more frequently used. As shown in Figure 6.10, more than 54% of all framework extension examples in Java Swing involve such a class.

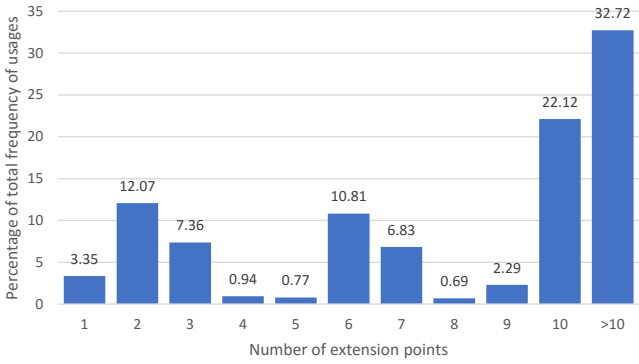


Figure 6.10: Usage frequencies of Swing framework classes with different numbers of extension points

Table 6.4: The percentage of cases where different numbers of framework extension points are used together

Framework	**Different numbers of extension points that are used together**				
	1 (%)	2 (%)	3 (%)	4 (%)	>4 (%)
Swing	83.50	11.00	3.90	1.10	0.50
JFace	83.10	12.60	3.70	0.50	0.10
JUnit	95.70	3.70	0.50	0.10	0
JGraphT	95.33	4.07	0.10	0.40	0.10
JUNG	87.00	8.10	2.50	1.60	0.80

Table 6.5: Evaluation Results of FEMIR for each extension pattern category

Extension	Precision			Recall			F-Measure		
Pattern Category	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
Simple	0.83	0.82	0.80	0.55	0.62	0.63	0.66	0.71	0.71
Customize	0.92	0.89	0.87	0.73	0.76	0.87	0.82	0.82	0.87
Extend	0.91	0.92	0.92	0.77	0.90	0.91	0.83	0.91	0.91
Implement	0.88	0.85	0.85	0.57	0.65	0.65	0.69	0.73	0.74

6.5.5 How often are multiple extension points used together?

A framework class may have multiple extension points. To understand the association between framework extension points, we are interested in knowing how often developers use different extension points together. To answer this question, we collect the data as follows. While parsing the source code, we collect the set of methods that are called on the same receiver object, including the constructor call that creates that object. The framework extension graphs are then grouped together based on the receiver objects. Each group of framework extension graphs provides the set of extension points that are used together. Table 6.4 shows how frequently different kinds of extension points are used together.

When they do use multiple extension points together, they most often use two (from 3.70% to 12.60%). The percentages decrease as the numbers of different extension points that are used together increase. It seems that developers rarely use five or more different extension points together.

6.6 Discussion

This section discusses a set of questions related to our study. These include the effectiveness of FEMIR in detecting extension pattern examples by categories, quality of canonical forms, effect of the threshold and the runtime performance.

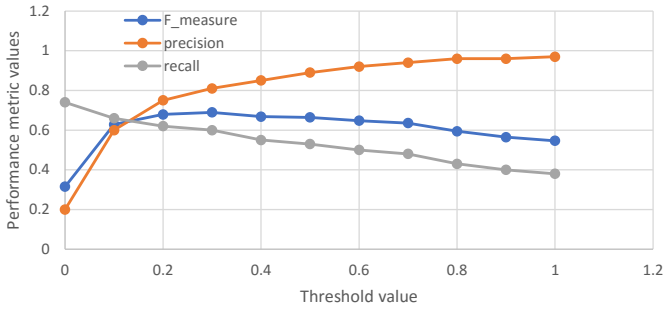


Figure 6.11: Precision, recall, and F-measure at different threshold values

6.6.1 Detecting examples of extension patterns by categories

The goal of this section is to explore how well FEMIR performs in detecting examples in each of the four extension pattern categories. We use the same experiment settings (but use only the Swing framework) and evaluation metrics as in Section 6.5. We first categorize the test cases of Java Swing framework into four different extension pattern categories. For test cases of each category, we run the experiment and determine the precision, recall, and F-measure values. Table 6.5 shows the results of the experiment. The result shows that FEMIR is indeed useful in recommending extension patterns for all four categories. Interestingly, the simple extension pattern category produces the lowest precision (0.80) and recall (0.63), and the extend pattern category produces the highest precision (0.92) and recall (0.91). More investigation is needed to understand what causes the differences between the categories, which remains as future work.

6.6.2 Quality of canonical forms

To test the accuracy of our canonical form representation, we generate a graph database using one hundred subject systems for the Java *Swing* framework. Then, we enumerate through the list of canonical forms in the graph database. We collect the sets of graphs that share the same canonical form. We then apply the graph isomorphism detection algorithm by Mckay [83] to all pairs of graphs in the set. The Mckay algorithm indeed identifies all pairs of graphs as being isomorphic. This confirms that the canonical form works correctly.

6.6.3 Effect of the threshold value

As explained in Section 6.4.A, our proposed technique uses a threshold value to determine the missing nodes that need to be added to the extension graph. To understand the effect of this threshold value, we conduct a study using *JFace* and its subject systems. It is the largest of the five different frameworks we have considered in our study. We use the ten-fold cross validation startegy and determine the precision, recall and F-Measure values by changing the threshold value from 0 to 1, at 0.10 interval. Figure 6.11 shows that the precision increases at the expense of recall. To balance the effect, our goal is to select a value that maximizes the F-measure value. The value increases with the increase of the threshold value and reaches to its highest value

at 0.3 threshold. After that we observe a gradual decrease of the F-measure value. Hence we recommend to use 0.3 in our study.

It should be noted that we assume that the costs of precision and recall errors are the same. If a precision error and a recall error have a different cost (for example, it would be reasonable to assume that a recall error is more costly than a precision error because developers could easily filter out irrelevant nodes), a different threshold will be chosen.

6.6.4 Runtime performance of FEMIR

To measure the runtime performance of the technique, we measure the time required to make recommendations. Based on executing a total of 16,268 queries, the average time required to recommend framework extension graphs per query was 0.92s for Java Swing. The major part of the recommendation time is needed for mining framework extension graphs. However, a significant fraction of the time can be saved if we mine the graphs for all extension points beforehand and index them to be used by the recommender later.

The time required to analyze the source code files and generate framework extension graphs requires significantly long time. For example, for *JFace* alone, it takes around 78 hours on a single node machine for FEMIR to generate framework extension graphs from the source code files. The time is mostly contributed by partial program analysis of the source code. However, this is only a one time operation.

6.7 Threats to Validity

There are three threats to the validity of this study.

First, we use five frameworks to evaluate our proposed technique. One can argue that the findings could be different for other frameworks. The frameworks we consider in our study are popular and a large number of Java systems are actively using them. Given that our proposed technique does not directly depend on a particular framework, we believe that the results we obtain should largely carry over to other frameworks.

Second, in this study, we collect software systems that are publicly hosted in GitHub. It is possible that software systems hosted in a different project hosting site other than GitHub, or close source projects can exhibit different framework extension patterns. To mitigate this effect, we consider a large number of projects in our study.

Third, the implement extension pattern category can also appear in the form of anonymous classes or even lambda expressions in the case of single abstract method interfaces. Our current implementation does not handle such cases.

6.8 Conclusion

In this chapter, we propose an approach to recommending framework extension patterns by mining previously written source code examples. We first define the concept of framework extension points and propose a taxonomy of framework extension patterns. Based on these, we then develop a graph mining approach for recommending the top-n frequent extension patterns. We hypothesize that showing these patterns and code examples will usefully aid developers in learning how to use the extension points. Our evaluation using a large set of applications built on top of five popular frameworks show that the proposed technique can help developers automatically discover framework extension points and their usage patterns. We also collect several statistics to characterize the framework extensions in our code base.

Chapter 7

Conclusion

Code completion has become an integral part of modern integrated development environments. Although code completion does not eliminate the need to learn software frameworks or libraries, it frees developers from remembering every detail and facilitates learning different aspects of APIs. Results from studies described in chapters 4 and 5 indicate that by carefully capturing code context and by integrating different sources of information, the performance of existing code completion systems can be improved. Code completion systems can also help developers to learn complex aspects of software frameworks or libraries that are difficult to notice otherwise. The rest of this chapter is organized as follows. Section 7.1 contains the summary of four studies presented in the last four chapters and Section 7.2 discusses future research directions in code completion.

7.1 Summary

This section summarizes the contributions of this thesis to the state-of-the-art research in code completion.

- **Capturing code context and using the simhash technique: A case study on tracking source code lines:** Capturing source code context is an important step in developing code completion systems. One possible way to do so is by considering tokens that appear within the top-4 lines prior to the target location. It is also possible to use the simhash technique to accelerate the context matching process. This can possibly help recommending completion proposals in real-time. However, it is required to quickly validate the benefits of utilizing the source code context and the simhash technique before using them in developing code completion systems. Thus, this thesis starts with the problem of tracking source code lines as a case study. This is because of the simplicity in developing and evaluating a line location tracking technique. A language independent line location tracking technique, called LHDiff, has been developed. Evaluation results indicate that LHDiff outperforms existing state-of-the-art techniques. This also validates the effectiveness of using new context information that leverages the source code localness property and the simhash technique. An implementation of the technique is also publicly available.

- **Context sensitive method call completion system:** Code completion helps developers use APIs and frees them from remembering every detail. This thesis describes a novel technique called CSCC (Context Sensitive Code Completion) for improving the performance of API method call completion.

CSCC is context sensitive in that it uses new sources of information as the context of a target method call. CSCC indexes method calls in code examples by their context. To recommend completion proposals, CSCC ranks candidate methods by the similarities between their contexts and the context of the target call. Evaluation using a set of subject systems and five popular state-of-the-art techniques suggests that CSCC performs better than existing type or example-based code completion systems. Additional experiments are conducted to find how different contextual elements of the target call benefit CSCC. Next, this thesis investigates the adaptability of the technique to support another form of code completion, i.e., field completion. Evaluation with eight different subject systems suggests that CSCC can easily support field completion with high accuracy. Finally, CSCC is compared with four popular statistical language models that support code completion. Results of statistical tests from the study suggest that CSCC not only outperforms those techniques that are based on token level language models, but also in most cases performs better or equally well with GraLan, the state-of-the-art graph-based language model.

- **Recommending API method parameters:** A number of techniques have been developed that support method call completion. However, there has been little research on the problem of method parameter completion. This thesis presents a study that helps to understand how developers complete method parameters. Based on the study, a technique is developed, called PARC that recommends method parameters. PARC collects parameter usage context using a source code localness property that suggests that developers tend to collocate related code fragments. The technique uses previous code examples together with contextual and static type analysis to recommend method parameters. Evaluation of the technique against the only available state-of-the-art tool using a number of subject systems and different Java libraries shows that PARC has potential. In addition, PARC is also compared with the parameter recommendation support provided by the Eclipse Java Development Tools (JDT).

- **Recommending framework extension examples:** The use of software frameworks enables the delivery of common functionality but with significantly less effort than when developing from scratch. To meet application specific requirements, the behavior of a framework needs to be customized via extension points. A common way of customizing framework behavior is by passing a framework related object as an argument to an API call. Such an object can be created by subclassing an existing framework class or interface, or by directly customizing an existing framework object. However, to do this effectively requires developers to have extensive knowledge of the framework's extension points and their interactions. To aid the developers in this regard, this thesis develops a graph mining approach for extension point management, called FEMIR. Specifically, a taxonomy of extension patterns is proposed to categorize the various ways an extension point has been used in the code examples. The technique mines a large amount of code examples to discover all extension points and patterns for each framework class. Given a framework class that is being used, FEMIR aids developers by following a two-step recommendation process. First, it recommends all the extension points that are available in the class.

Once the developer chooses an extension point, FEMIR then discovers all of its usage patterns and recommends the best code examples for each pattern. Using five frameworks, the performance of the two-step recommendation is evaluated, in terms of precision, recall, and F-measure.

7.2 Future Research Directions

To determine future research directions, I focus on the development of code completion systems that provide better support for discovering APIs and related resources. Based on the analysis of existing code completion literature, the necessity of further research in the following directions is expected.

7.2.1 Analyze code completion systems using IDE usage data

Typically, code recommendation systems, (such as code completion systems) are evaluated by considering the fact that the source code was developed in a top-down manner. However, this is not always the case. Developers frequently copy and paste code fragments and then edit them. It is also found that the currently employed artificial evaluations are far from perfect [108]. They may fail to capture the evolving code context and this has a significant impact on the performance of the code completion systems. To gain further insights about the benefits of code completion systems it is required to collect the sequence of edits that lead to the latest version of the source code. Although there are a small number of such systems available [113], they either focus on collecting the commands developers use in the IDE or are difficult to use. Thus, it is required to develop an infrastructure to collect edits of developers and source code modification information. Such an infrastructure enables to perform both qualitative and quantitative studies to compare the performance of code completion systems considering the flow of source code changes. Furthermore, such a system will help developing and evaluating automatic code commit systems, source code change detection systems, and any other code recommendation systems.

7.2.2 Bringing natural language processing to code completion

Code completion systems offer a list of completion proposals as developers type code. For example, when a developer types a dot(.) after a receiver type, the completion system shows a list of methods or variable names. Thus, the existing code completion systems mainly focus on the types that are directly accessible from the receiver type developer is currently using. However, an alternative to this is to tell developers the operations that can be performed on a receiver type. Many of these operations lead to a state that cannot be directly achievable from the receiver type the developer is working with. For example, if a developer is working on a collection object, she can sort the elements of the collection object. She can also iterate through the collection elements. However, many of these operations may involve with a sequence of method calls and the types of objects that are required to perform these operations may not be known to developers. Thus, these operations need to be discovered through brief natural language text. A few works in this direction have

already been proposed [78][126][40], but they have their own limitations. For example, in case of the keyword matching technique, developers need to type a query in natural language text and the system generates code that answer the query. However, the problem with this kind of approach is that developers need to create their own queries and these may lead to vocabulary mismatch problems [78]. Another problem is that there is no way to inform the developer that the system does not have an answer to her query. Thus, it is required to integrate techniques from other areas (like human computer interaction) to find better user interfaces, perform both empirical and user studies to understand the problem and verify the usefulness of the proposed solution.

7.2.3 Going beyond code completion

The predominant approach in code completion is to offer a list of completion proposals that are textual in nature. When a developer selects a proposal, the corresponding code is inserted in the IDE. This indicates that the interaction between the developer and the completion menu is quite limited. However, the code completion feature of the IDE can be extended by improving the completion interface. This can be done by using the code completion menu to connect developers to other sources of information or to allow developers to refine their input. For example, a completion menu can not only help the developers to complete an HTML table but can also help developers to preview the styles that can be applied to it. Similarly, a completion menu can bring the external API documentation in the IDE. However, only a few works have been done in this direction (such as the temporal code completion [74, 75], active code completion [102], and Dompletion [14]).

7.2.4 Locating placeholder methods

When a developer is working with an object of a class, the method they are looking for may not be located in that class. Instead, the method can be located in a different class. Duala-Ekoka and Robillard [30] improve the discovery of these methods by considering the relationship between a method and its parameters. However, such relationship cannot help to discover all required methods. For example, the method a developer is looking for may be located in a class that is not directly related to the receiver class. Consider that a developer has a graph object and she is looking to display the object. She finds a *JGraphT* library that can help in this problem. She creates an instance of the *JGraph* and now looking for a method to set the layout of the graph. She is expecting a method similar to *setGraphLayout* or *setLayout*. There is no such method in *JGraph*. The following code fragment shows how to do that in the *JGraph*. To set the layout of a *JGraph* object, one needs to call the run method that is located in the *JGraphFacade* class. This is an example of a placeholder method, a method that a developer is looking but the method does not exist in the class of the current receiver variable.

```
JGraph jgraph = new JGraph(new JGraphModelAdapter(g));
JGraphFacade facade =new JGraphFacade(jgraph);
```

```
facade.run(circleLayout, true);
```

The Prospector tool [79] is relevant in this case but is indirectly related. It allows users to provide the type of the source and destination objects and then provides corresponding implementation example. However, the tool cannot solve the problem because the goal is to help developers to discover the placeholder methods and only the source object type is available here. Code completion system that helps developers discovering methods that are likely to be called later but only indirectly related to the object developer is currently working with can save both development time and effort.

7.2.5 Supporting untyped/weakly typed languages

Many of the code completion systems discussed in this section take into account the type information to improve the performance of recommendation. For example, API method call completion systems consider the receiver type to filter irrelevant method calls [18, 109]. While for statically typed programming languages (such as Java) those type information can be obtained easily, it is difficult to collect the information for weakly typed languages (such as JavaScript) due to various implicit type conversions. It is also not clear to what extent the existing algorithms perform for the untyped/weakly typed languages. Thus, it is required to extend and explore code completion support for those languages.

References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API Patterns As Partial Orders from Source Code: From Usage Scenarios to Specifications. In *Proc. of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 25–34, 2007.

[2] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning Natural Coding Conventions. In *Proc. of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 281–293, 2014.

[3] T. Apiwattanapong, A. Orso, and M. J. Harrold. JDiff: A Differencing Technique and Tool for Object-Oriented Programs. *Automated Software Engg.*, 14(1):3–36, March 2007.

[4] M. Asaduzzaman, C. K. Roy, S. Monir, and K. A. Schneider. Exploring API Method Parameter Recommendations. In *Proc. of the IEEE International Conference on Software Maintenance and Evolution*, pages 271–280, 2015.

[5] M. Asaduzzaman, C. K. Roy, and K. A. Schneider. PARC: Recommending API Methods Parameters. In *Proc. of the IEEE International Conference on Software Maintenance and Evolution*, pages 330–332, 2015.

[6] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou. Context-Sensitive Code Completion Tool for Better API Usability. In *Proc. of the 2014 IEEE International Conference on Software Maintenance and Evolution*, pages 621–624, 2014.

[7] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou. CSCC: Simple, Efficient, Context Sensitive Code Completion. In *Proc. of the IEEE International Conference on Software Maintenance and Evolution*, pages 71–80, 2014.

[8] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou. A Simple, Efficient, Context-sensitive Approach for Code Completion. *Journal of Software: Evolution and Process*, 28(7):512–541, 2016.

[9] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou. FEMIR: A Tool for Recommending Framework Extension Examples. In *Proc. of the 32nd International Conference on Automated Software Engineering*, 2017. To appear.

[10] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou. Recommending Framework Extension Examples. In *Proc. of the 33rd International Conference on Software Maintenance and Evolution*, 2017. To appear.

[11] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and M. D. Penta. LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines. In *Proc. of the 2013 IEEE International Conference on Software Maintenance*, pages 230–239, 2013.

[12] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and M. D. Penta. LHDiff: Tracking Source Code Lines to Support Software Maintenance Activities. In *2013 IEEE International Conference on Software Maintenance*, pages 484–487, 2013.

[13] L. Babai and E. M. Luks. Canonical Labeling of Graphs. In *Proc. of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 171–183, 1983.

[14] K. Bajaj, K. Pattabiraman, and A. Mesbah. Dompletion: DOM-Aware JavaScript Code Completion. In *Proceedings of the 29th International Conference on Automated Software Engineering*, pages 43–54, 2014.

[15] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proc. of the International Conference on Software Maintenance*, pages 368–377, 1998.

[16] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The Promises and Perils of Mining Git. In *Proc. of the 6th IEEE International Working Conference on Mining Software Repositories*, pages 1–10, 2009.

[17] M Bruch, M Mezini, and M Monperrus. Mining Subclassing Directives to Improve Framework Reuse. In *Proc. of the 7th IEEE Working Conference on Mining Software Repositories*, pages 141–150, 2010.

[18] M. Bruch, M. Monperrus, and M. Mezini. Learning from Examples to Improve Code Completion Systems. In *Proc. of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 213–222, 2009.

[19] M. Bruch, T. Schäfer, and M. Mezini. FrUiT: IDE Support for Framework Understanding. In *Proc. of the Workshop on Eclipse Technology eXchange*, pages 55–59, 2006.

[20] M. Bruch, T. Schäfer, and M. Mezini. On Evaluating Recommender Systems for API Usages. In *Proc. of the International Workshop on Recommendation Systems for Software Engineering*, pages 16–20, New York, NY, USA, 2008.

[21] J. C. Campbell, A. Hindle, and J. N. Amaral. Syntax Errors Just Aren't Natural: Improving Error Reporting with Language Models. In *Proc. of the 11th Working Conference on Mining Software Repositories*, pages 252–261, 2014.

[22] G. Canfora, L. Cerulo, and M. Di Penta. Tracking Your Changes: A Language-Independent Approach. *IEEE Softw.*, 26(1):50–57, January 2009.

[23] G. Canfora, L. Cerulo, and M. Di Penta. Identifying Changed Source Code Lines from Version Repositories. In *Proc. of the Fourth International Workshop on Mining Software Repositories*, pages 14–, 2007.

[24] M. S. Charikar. Similarity Estimation Techniques from Rounding Algorithms. In *Proc. of the 34th Annual ACM Symposium on Theory of Computing*, pages 380–388, 2002.

[25] S. Chatterjee, S. Juvekar, and K. Sen. SNIFF: A Search Engine for Java Using Free-Form Queries. In *Proc. of the 12th International Conference on Fundamental Approaches to Software Engineering: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, pages 385–400, 2009.

[26] J. R. Cordy, T. R. Dean, and N. Synytskyy. Practical Language-Independent Detection of Near-Miss Clones. In *Proc. of 24th Annual International Conference on Computer Science and Software Engineering*, pages 1–12, 2004.

[27] B. Dagenais and L. Hendren. Enabling Static Analysis for Partial Java Programs. In *Proc. of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, pages 313–328, 2008.

[28] B. Dagenais and H. Ossher. Automatically Locating Framework Extension Examples. In *Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 203–213, 2008.

[29] E. Duala-Ekoko and M. P. Robillard. Tracking Code Clones in Evolving Software. In *Proc. of the 29th International Conference on Software Engineering*, pages 158–167, 2007.

[30] E. Duala-Ekoko and M. P. Robillard. Using Structure-Based Recommendations to Facilitate Discoverability in APIs. In *Proc. of the 25th European Conference on Object-oriented Programming*, pages 79–104, 2011.

[31] S. Ducasse, M. Rieger, and S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proc. of the IEEE International Conference on Software Maintenance*, pages 109–118, 1999.

[32] B. Ellis, J. Stylos, and B. Myers. The Factory Pattern in API Design: A Usability Evaluation. In *Proc. of the 29th International Conference on Software Engineering*, pages 302–312, 2007.

[33] N. Flores and A. Aguiar. Patterns for Understanding Frameworks. In *Proc. of the 15th Conference on Pattern Languages of Programs*, pages 1–11, 2008.

[34] B. Fluri and H. C. Gall. Classifying Change Types for Qualifying Change Couplings. In *Proc. of the 14th IEEE International Conference on Program Comprehension*, pages 35–45, 2006.

[35] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[36] C. Franks, Z. Tu, P. Devanbu, and V. Hellendoorn. CACHECA: A Cache Language Model Based Code Suggestion Tool. In *Proc. of the 37th International Conference on Software Engineering*, pages 705–708, 2015.

[37] G. Froehlich, H. J. Hoover, L. Liu, and P. Sorenson. Hooking into Object-Oriented Application Frameworks. In *Proc. of the 19th International Conference on Software Engineering*, pages 491–501, 1997.

[38] N. Göde and R. Koschke. Incremental Clone Detection. In *Proc. of the 13th European Conference on Software Maintenance and Reengineering*, pages 219–228, 2009.

[39] M. W. Godfrey and L. Zou. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[40] S. Han, D. R. Wallace, and R. C. Miller. Code Completion from Abbreviated Input. In *Proc. of the 24th International Conference on Automated Software Engineering*, pages 332–343, 2009.

[41] L. Heinemann, V. Bauer, M. Herrmannsdoerfer, and B. Hummel. Identifier-Based Context-Dependent API Method Recommendation. In *Proc. of the 16th European Conference on Software Maintenance and Reengineering*, pages 31–40, 2012.

[42] L. Heinemann and B. Hummel. Recommending API Methods Based on Identifier Contexts. In *Proc. of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, pages 1–4, 2011.

[43] A. Heydarnoori, K. Czarnecki, W. Binder, and T. T. Bartolomei. Two Studies of Framework-Usage Templates Extracted from Dynamic Traces. *IEEE Transactions on Software Engineering*, 38(6):1464–1487, 2012.

[44] R. Hill and J. Rideout. Automatic Method Completion. In *Proc. of the 19th International Conference on Automated Software Engineering*, pages 228–235, 2004.

[45] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the Naturalness of Software. In *Proc. of the 34th International Conference on Software Engineering*, pages 837–847, 2012.

[46] R. Holmes and G. C. Murphy. Using Structural Context to Recommend Source Code Examples. In *Proc. of the 27th International Conference on Software Engineering*, pages 117–125, 2005.

[47] D. Hou and D. M. Pletcher. Towards a Better Code Completion System by API Grouping, Filtering, and Popularity-Based Ranking. In *Proc. of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pages 26–30, 2010.

[48] D. Hou and D. M. Pletcher. An Evaluation of the Strategies of Sorting, Filtering, and Grouping API Methods for Code Completion. In *Proc. of the 27th IEEE International Conference on Software Maintenance*, pages 233–242, 2011.

[49] D. Hou, C. R. Rupakheti, and H. J. Hoover. Documenting and Evaluating Scattered Concerns for Framework Usability: A Case Study. In *Proc. of the 15th Asia-Pacific Software Engineering Conference*, pages 213–220, 2008.

[50] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-Based Code Clone Detection: Incremental, Distributed, Scalable. In *Proc. of the IEEE International Conference on Software Maintenance*, pages 1–9, 2010.

[51] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component Rank: Relative Significance Rank for Software Component Search. In *Proc. of the 25th International Conference on Software Engineering*, pages 14–24, 2003.

[52] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto. Ranking significance of software components based on use relations. *IEEE Transactions on Software Engineering*, 31(3):213–225, 2005.

[53] T. Ishihara, K. Hotta, Y. Higo, and S. Kusumoto. Reusing Reused Code. In *Proc. of the 20th Working Conference on Reverse Engineering*, pages 457–461, 2013.

[54] J. Jacobellis, N. Meng, and M. Kim. Cookbook: In Situ Code Completion Using Edit Recipes Learned from Examples. In *Proc. of the 36th International Conference on Software Engineering*, pages 584–587, 2014.

[55] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *Proc. of the 29th International Conference on Software Engineering*, pages 96–105, 2007.

[56] R. E. Johnson. Documenting Frameworks Using Patterns. In *Proc. on Object-Oriented Programming Systems, Languages, and Applications*, pages 63–76, 1992.

[57] E. Juergens, F. Deissenboeck, and B. Hummel. CloneDetective - A workbench for Clone Detection Research. In *Proc. of the 31st International Conference on Software Engineering*, pages 603–606, 2009.

[58] 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*, 28:654–670, 2002.

[59] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida. SHINOBI: A Tool for Automatic Code Clone Detection in the IDE. In *Proc. of the 16th Working Conference on Reverse Engineering*, pages 313–314, 2009.

[60] D. Kawrykow and M. P. Robillard. Non-Essential Changes in Version Histories. In *Proc. of the 33rd International Conference on Software Engineering*, pages 351–360, 2011.

[61] I. Keivanloo, J. Rilling, and P. Charland. Internet-Scale Real-Time Code Clone Search Via Multi-Level Indexing. In *Proc. of the 18th Working Conference on Reverse Engineering*, pages 23–27, 2011.

[62] I. Keivanloo, J. Rilling, and Y. Zou. Spotting Working Code Examples. In *Proc. of the 36th International Conference on Software Engineering*, pages 664–675, 2014.

[63] I. Keivanloo, C. K. Roy, and J. Rilling. SeByte. *Science of Computer Programming*, 95(P4):426–444, 2014.

[64] M. Kim and D. Notkin. Using a Clone Genealogy Extractor for Understanding and Supporting Evolution of Code Clones. In *Proc. of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, 2005.

[65] M. Kim and D. Notkin. Program Element Matching for Multi-Version Program Analyses. In *Proc. of the 2006 International Workshop on Mining Software Repositories*, pages 58–64, 2006.

[66] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An Empirical Study of Code Clone Genealogies. In *Proc. of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 187–196, 2005.

[67] S. Kim, K. Pan, and E. J. Whitehead. When Functions Change Their Names: Automatic Detection of Origin Relationships. In *Proc. of the 12th Working Conference on Reverse Engineering*, pages 143–152, 2005.

[68] R. Koschke. Large-Scale Inter-System Clone Detection Using Suffix Trees. In *Proc. of the 16th European Conference on Software Maintenance and Reengineering*, pages 309–318, 2012.

[69] R. Koschke, R. Falke, and P. Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. In *Proc. of the 13th Working Conference on Reverse Engineering*, pages 253–262, 2006.

[70] J. Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proc. Eighth Working Conference on Reverse Engineering*, pages 301–309, 2001.

[71] H. W. Kuhn. The Hungarian Method for the Assignment Problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[72] R. F. Q. Lafetá, M. A. Maia, and D. Röthlisberger. Framework Instantiation Using Cookbooks Constructed with Static and Dynamic Analysis. In *Proc. of the 2015 IEEE 23rd International Conference on Program Comprehension*, pages 125–128, 2015.

[73] M. Lee, J. Roh, S. Hwang, and S. Kim. Instant Code Clone Search. In *Proc. of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 167–176, 2010.

[74] Y. Y. Lee, S. Harwell, S. Khurshid, and D. Marinov. Temporal Code Completion and Navigation. In *Proc. of the International Conference on Software Engineering*, pages 1181–1184, 2013.

[75] Y. Y. Lee, D. Marinov, and R. E. Johnson. Tempura: Temporal Dimension for IDEs. In *Proc. of the 37th International Conference on Software Engineering*, pages 212–222, 2015.

[76] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.

[77] G. Little and R. C. Miller. Translating Keyword Commands into Executable Code. In *Proc. of the 19th Annual ACM Symposium on User Interface Software and Technology*, pages 135–144, 2006.

[78] G. Little and R. C. Miller. Keyword Programming in Java. In *Proc. of the 22nd International Conference on Automated Software Engineering*, pages 84–93, 2007.

[79] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid Mining: Helping to Navigate the API Jungle. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 48–61, 2005.

[80] G. S. Manku, A. Jain, and A. Das Sarma. Detecting Near-Duplicates for Web Crawling. In *Proc. of the 16th International Conference on World Wide Web*, pages 141–150, 2007.

[81] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *Proc. of the 1996 International Conference on Software Maintenance*, pages 244–253, 1996.

[82] F. McCarey, M. Ó Cinnéide, and N. Kushmerick. Rascal: A Recommender Agent for Agile Reuse. *Artificial Intelligence Review*, 24(3):253–276, 2005.

[83] Brendan D. Mckay and Adolfo Piperno. Practical Graph Isomorphism, II. *J. Symb. Comput.*, 60:94–112, 2014.

[84] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding Relevant Functions and Their Usage. In *Proc. of the 33rd International Conference on Software Engineering*, pages 111–120, 2011.

[85] A. Michail. Data Mining Library Reuse Patterns in User-Selected Applications. In *Proc. of the 14th IEEE International Conference on Automated Software Engineering*, pages 24–33, 1999.

[86] A. Michail. Data Mining Library Reuse Patterns Using Generalized Association Rules. In *Proc. of the 22nd International Conference on Software Engineering*, pages 167–176, 2000.

[87] M. Mooty, A. Faulring, J. Stylos, and B. A. Myers. Calcite: Completing Code Completion for Constructors Using Crowds. In *Proc. of the Symposium on Visual Languages and Human-Centric Computing*, pages 15–22, 2010.

[88] E. Moritz, M. Linares-Vásquez, D. Poshyvanyk, M. Grechanik, C. McMillan, and M. Gethers. ExPort: Detecting and Visualizing API Usages in Large Source Code Repositories. In *Proc. of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 646–651, 2013.

[89] D. Movshovitz-Attias and W. W. Cohen. Natural Language Models for Predicting Programming Comments. In *Proc. of the 51st Annual Meeting of the Association for Computational Linguistics*, pages 35–40, 2013.

[90] G. C. Murphy, M. Kersten, and L. Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[91] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, R. Lily, E. Rademacher, T. N. Nguyen, and D. Dig. API Code Recommendation Using Statistical Learning from Fine-Grained Changes. In *Proc. of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 511–522, 2016.

[92] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. GraPacc: A Graph-Based Pattern-Oriented, Context-Sensitive Code Completion Tool. In *Proc. of the 34th International Conference on Software Engineering*, pages 1407–1410, 2012.

[93] A. T. Nguyen and T. N. Nguyen. Graph-Based Statistical Language Model for Code. In *Proc. of the 37th International Conference on Software Engineering*, pages 858–868, 2015.

[94] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-Based Pattern-Oriented, Context-Sensitive Source Code Completion. In *Proc. of the 34th International Conference on Software Engineering*, pages 69–79, 2012.

[95] T. Nguyen, P. C. Rigby, A. T. Nguyen, M. Karanfil, and T. N. Nguyen. T2API: Synthesizing API Code Usage Templates from English Texts with Statistical Translation. In *Proc. of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1013–1017, 2016.

[96] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A Statistical Semantic Language Model for Source Code. In *Proc. of the 9th Joint Meeting on Foundations of Software Engineering*, pages 532–542, 2013.

[97] T. T. Nguyen, H. A. Nguyen, J. M. Al-Kofahi, N. H. Pham, and T. N. Nguyen. Scalable and Incremental Clone Detection for Evolving Software. In *Proc of the International Conference on Software Maintenance*, pages 491–494, 2009.

[98] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. ClemanX: Incremental Clone Detection Tool for Evolving Software. In *Proc. of the 31st International Conference on Software Engineering - Companion Volume*, pages 437–438, 2009.

[99] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-Based Mining of Multiple Object Usage Patterns. In *Proc. of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2009.

[100] T. T. Nguyen, H. V. Pham, V. Hung, P. M. Vu, and T. T. Nguyen. Learning API Usages from Bytecode: A Statistical Approach. In *Proc. of the 38th International Conference on Software Engineering*, pages 416–427, 2016.

[101] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyennh. Recommending API Usages for Mobile Apps with Hidden Markov Model. In *Proc. of the 30th International Conference on Automated Software Engineering*, pages 795–800, 2015.

[102] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active Code Completion. In *Proc. of the Symposium on Visual Languages and Human-Centric Computing*, pages 261–262, 2011.

[103] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active Code Completion. In *Proc. of the International Conference on Software Engineering*, pages 859–869, 2012.

[104] T. Omori, H. Kuwabara, and K. Maruyama. A Study on Repetitiveness of Code Completion Operations. In *Proc. of the 28th International Conference on Software Maintenance*, pages 584–587, 2012.

[105] D. M. Pletcher and D. Hou. BCC: Enhancing Code Completion for Better API Usability. In *Proc. of the International Conference on Software Maintenance*, pages 393–394, 2009.

[106] M. Pradel and T. R. Gross. Detecting Anomalies in the Order of Equally-typed Method Arguments. In *Proc. of the International Symposium on Software Testing and Analysis*, pages 232–242, 2011.

[107] M. Pradel, S. Heiniger, and T. R. Gross. Static Detection of Brittle Parameter Typing. In *Proc. of the International Symposium on Software Testing and Analysis*, pages 265–275, 2012.

[108] S. Proksch, S. Amann, S. Nadi, and M. Mezini. Evaluating the Evaluations of Code Recommender Systems: A Reality Check. In *Proc. of the 31st International Conference on Automated Software Engineering*, pages 111–121, 2016.

[109] S. Proksch, J. Lerch, and M. Mezini. Intelligent Code Completion with Bayesian Networks. *ACM Transactions on Software Engineering and Methodology*, 25(1):3:1—-3:31, 2015.

[110] V. Raychev, Martin V., and E. Yahav. Code Completion with Statistical Language Models. In *Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–428. ACM, 2014.

[111] S. Reiss. Tracking Source Locations. In *Proc. of the 2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 11–20, 2008.

[112] M. Rieger, S. Ducasse, and M. Lanza. Insights into System-Wide Code Duplication. In *Proc. of the 11th Working Conference on Reverse Engineering*, pages 100–109, 2004.

[113] R. Robbes and M. Lanza. How Program History Can Improve Code Completion. In *Proc. of the 23rd International Conference on Automated Software Engineering*, pages 317–326, 2008.

[114] R. Robbes and M. Lanza. How Program History Can Improve Code Completion. In *Proc. of the 23rd International Conference on Automated Software Engineering*, pages 317–326, 2008.

[115] R. Robbes and M. Lanza. Improving Code Completion with Program History. *Automated Software Engineering*, 17(2):181–212, 2010.

[116] M. P. Robillard. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Softw.*, 26(6):27–34, 2009.

[117] MartinP. Robillard and Robert DeLine. A Field Study of API Learning Obstacles. *Empirical Software Engineering*, 16(6):703–732, 2010.

[118] C. K. Roy and J. R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *Proc. of the 16th IEEE International Conference on Program Comprehension*, pages 172–181, 2008.

[119] C. K. Roy and J. R. Cordy. A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 157–166, 2009.

[120] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[121] C. R. Rupakheti and D. Hou. Evaluating Forum Discussions to Inform the Design of an API Critic. In *Proc. of the 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 53–62, 2012.

[122] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, and K. A. Schneider. Evaluating Code Clone Genealogies at Release Level: An Empirical Study. In *Proc. of the 10th IEEE Working Conference on Source Code Analysis and Manipulation*, pages 87–96, 2010.

[123] N. Sahavechaphan and K. Claypool. XSnippet: Mining for Sample Code. In *Proc. of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 413–430, 2006.

[124] V. Saini, H. Sajnani, J. Kim, and C. Lopes. SourcererCC and SourcererCC-I: Tools to Detect Clones in Batch Mode and During Software Development. In *Proc. of the 38th International Conference on Software Engineering Companion*, pages 597–600, 2016.

[125] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. Lopes. SourcererCC: Scaling Code Clone Detection to Big-Code. In *Proc. of the 38th International Conference on Software Engineering*, pages 1157–1168, 2016.

[126] Y. Sakamoto, H. Sato, and M. Kurihara. Improvement and Implementation of Keyword Programming. In *Proc. of the International Conference on Systems Man and Cybernetics*, pages 474–480, 2010.

[127] J. Spacco and C. Williams. Lightweight Techniques for Tracking Unique Program Statements. In *Proc. of the Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 99–108, 2009.

[128] M. Storey, L. Cheng, I. Bull, and P. Rigby. Shared Waypoints and Social Tagging to Support Collaboration in Software Development. In *Proc. of the 20th Anniversary Conference on Computer Supported Cooperative Work*, pages 195–198, 2006.

[129] J. Stylos, A. Faulring, Z. Yang, and B. A. Myers. Improving API Documentation Using API Usage Information. In *Proc. of the Symposium on Visual Languages and Human-Centric Computing*, pages 119–126, 2009.

[130] J Stylos and B A Myers. Mica: A Web-Search Tool for Finding API Components and Examples. In *Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 195–202, 2006.

[131] J. Stylos and B. A. Myers. The Implications of Method Placement on API Learnability. In *Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 105–112, 2008.

[132] J. Svajlenko and C. K. Roy. CloneWorks: A Fast and Flexible Large-Scale Near-Miss Clone Detection Tool. In *Proc. of the 39th International Conference on Software Engineering Companion*, pages 177–179, 2017.

[133] S. Thummalapenta and T. Xie. Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proc. of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 204–213, 2007.

[134] S. Thummalapenta and T. Xie. SpotWeb: Detecting Framework Hotspots and Coldspots via Mining Open Source Code on the Web. In *Proc. of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 327–336, 2008.

[135] S. Thummalapenta and T. Xie. SpotWeb: Detecting Framework Hotspots via Mining Open Source Repositories on the Web. In *Proc. of the 2008 International Working Conference on Mining Software Repositories*, pages 109–112, 2008.

[136] F. Thung, S. Wang, D. Lo, and J. Lawall. Automatic Recommendation of API Methods from Feature Requests. In *Proc. of the 28th International Conference on Automated Software Engineering*, pages 290–300, 2013.

[137] Z. Tu, Z. Su, and P. Devanbu. On the Localness of Software. In *Proc. of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280, 2014.

[138] M. S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle. On the Effectiveness of Simhash for Detecting Near-Miss Clones in Large Scale Software Systems. In *Proc. of the 18th Working Conference on Reverse Engineering*, pages 13–22, 2011.

[139] J. Viljamaa. Reverse Engineering Framework Reuse Interfaces. *SIGSOFT Softw. Eng. Notes*, 28(5):217–226, 2003.

[140] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining Succinct and High-Coverage API Usage Patterns from Source Code. In *Proc. of the 10th IEEE Working Conference on Mining Software Repositories*, pages 319–328, 2013.

[141] Z. Xing and E. Stroulia. UMLDiff: An Algorithm for Object-Oriented Design Differencing. In *Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 54–65, 2005.

[142] T. Yamamoto, N. Yoshida, and Y. Higo. Seamless Code Reuse with Source Code Corpus. In *Proc. of the 20th Asia-Pacific Software Engineering Conference*, pages 31–36, 2013.

[143] X. Yan and J. Han. gSpan: Graph-Based Substructure Pattern Mining. In *Proc. of the IEEE International Conference on Data Mining*, pages 721–724, 2002.

[144] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic Parameter Recommendation for Practical API Usage. In *Proc. of the 34th International Conference on Software Engineering*, pages 826–836, 2012.

[145] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and Recommending API Usage Patterns. In *Proc. of the 23rd European Conference on Object-Oriented Programming*, pages 318–343, 2009.

[146] M. F. Zibran and C. K. Roy. IDE-Based Real-Time Focused Search for Near-Miss Clones. In *Proc. of the 27th Annual ACM Symposium on Applied Computing*, pages 1235–1242, 2012.

[147] T. Zimmermann, S. Kim, A. Zeller, and J. E. J. Whitehead. Mining Version Archives for Co-Changed Lines. In *Proc. of the 2006 International Workshop on Mining Software Repositories*, pages 72–75, 2006.