Titre: Title:	Context-Aware Source Code Identifier Splitting and Expansion for Software Maintenance
Auteur: Author:	Latifa Guerrouj
Date:	2013
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Guerrouj, L. (2013). Context-Aware Source Code Identifier Splitting and Expansion for Software Maintenance [Ph.D. thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1203/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/1203/
Directeurs de recherche: Advisors:	Giuliano Antoniol, & Yann-Gaël Guéhéneuc
Programme: Program:	Génie informatique

UNIVERSITÉ DE MONTRÉAL

CONTEXT-AWARE SOURCE CODE IDENTIFIER SPLITTING AND EXPANSION FOR SOFTWARE MAINTENANCE

LATIFA GUERROUJ
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
AOÛT 2013

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

CONTEXT-AWARE SOURCE CODE IDENTIFIER SPLITTING AND EXPANSION FOR SOFTWARE MAINTENANCE

présentée par: GUERROUJ Latifa
en vue de l'obtention du diplôme de: Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de:

Mme BOUCHENEB Hanifa, Doctorat, présidente
M. ANTONIOL Giuliano, Ph.D., membre et directeur de recherche
M. GUÉHÉNEUC Yann-Gaël, Doct., membre et codirecteur de recherche
M. DESMARAIS Michel C., Ph.D., membre
Mme LAWRIE Dawn J., Ph.D., membre

This dissertation is dedicated to my parents. For their endless love, support and encouragement.

ACKNOWLEDGMENTS

I am very grateful to both Giulio and Yann for their support, encouragement, and intellectual input. I worked with you for four years or even less, but what I learned from you will last forever. Giulio, your passion about research was a source of inspiration and motivation for me. Also, your mentoring and support have been instrumental in achieving my goals. Yann, your enthusiasm and guidance have always been a strength for me to keep moving forward.

Research would not be as much fun without students and researchers to collaborate with. It has been a real pleasure and great privilege working with Massimiliano Di Penta (University of Sannio), Denys Poshyvanyk (College of William and Mary), and their teams. In particular, I would like to thank Max for being always available to provide help and excellent advice and for hosting me in Europe and sharing with me not only the research expertise but also a joyful time and fun. Many thanks also to Denys and his wonderful team for all the great collaborations we achieved together.

Thanks to all present and past SOCCER and Ptidej groups' members, in particular, Foutse Khomh and Bram Adams, for their help, relevant opinions and especially their encouragements during all the past years of my Ph.D., and to all my friends who inspired me along the way and never hesitated to share ideas and fun. I would also like to thank my professors at École Polytechnique de Montréal, the department of Software Engineering and Computer Science (DGIGL), and all the people that helped in the administrative process of this thesis.

I am very thankful to Radouane Mrabet and Line Dubé who always believed in me and gave me excellent advice. And a heartfelt thank to my Mother and Father, who instilled the following in me as a child: an achievement is an achievement when mind, heart and principles agree.

The most important thanks goes to my family. My two little nephews: Amine and Omar, and beautiful niece: kawtar. Thanks for your innocent smiles that were my source of motivation in the hard time. You always tried to call at just the right time.

To all my friends, students and beloved, thank you for your friendship, love, and appreciation. I can not list all your names here but you know well that you are always in my mind.

Finally, I would like to gratefully thanks the jury members who accepted to evaluate this thesis.

RÉSUMÉ

La compréhension du code source des programmes logiciels est une étape nécessaire pour plusieurs tâches de compréhension de programmes, rétro-ingénierie, ou re-documentation. Dans le code source, les informations textuelles telles que les identifiants et les commentaires représentent une source d'information importante.

Le problème d'extraction et d'analyse des informations textuelles utilisées dans les artefacts logiciels n'a été reconnu par la communauté du génie logiciel que récemment. Des méthodes de recherche d'information ont été proposées pour aider les tâches de compréhension de programmes telles que la localisation des concepts et la traçabilité des exigences au code source. Afin de mieux tirer bénéfice des approches basées sur la recherche d'information, le langage utilisé au niveau de tous les artefacts logiciels doit être le même. Ceci est dû au fait que les requêtes de la recherche d'information ne peuvent pas retourner des documents pertinents si le vocabulaire utilisé dans les requêtes contient des mots qui ne figurent pas au niveau du vocabulaire du code source. Malheureusement, le code source contient une proportion élevée de mots qui ne sont pas significatifs, *e.g.*, abréviations, acronymes, ou concaténation de ces types. En effet, le code source utilise un langage différent de celui des autres artefacts logiciels. Cette discordance de vocabulaire provient de l'hypothèse implicite faite par les techniques de recherche de l'information et du traitement de langage naturel qui supposent l'utilisation du même vocabulaire. Ainsi, la normalisation du vocabulaire du code source est un grand défi.

La normalisation aligne le vocabulaire utilisé dans le code source des systèmes logiciels avec celui des autres artefacts logiciels. La normalisation consiste à décomposer les identifiants (*i.e.*, noms de classes, méthodes, variables, attributs, paramètres, etc.) en termes et à étendre ces termes aux concepts (*i.e.*, mots d'un dictionnaire spécifique) correspondants.

Dans cette thèse, nous proposons deux contributions à la normalisation avec deux nouvelles approches contextuelles : TIDIER et TRIS. Nous prenons en compte le contexte car nos études expérimentales ont montré l'importance des informations contextuelles pour la normalisation du vocabulaire du code source. En effet, nous avons effectué deux études expérimentales avec des étudiants de baccalauréat, maîtrise et doctorat ainsi que des stagiaires post-doctoraux. Nous avons choisi aléatoirement un ensemble d'identifiants à partir d'un corpus de systèmes écrits en C et nous avons demandé aux participants de les normaliser en utilisant différents niveaux de contexte. En particulier, nous avons considéré un contexte interne qui consiste en le contenu des fonctions, fichiers et systèmes contenant les identifiants ainsi qu'un niveau externe sous forme de documentation externe. Les résultats montrent

l'importance des informations contextuelles pour la normalisation. Ils révèlent également que les fichiers de code source sont plus utiles que les fonctions et que le contexte construit au niveau des systèmes logiciels n'apporte pas plus d'amélioration que celle obtenue avec le contexte construit au niveau des fichiers. La documentation externe, par contre, aide parfois. En résumé, les résultats confirment notre hypothèse sur l'importance du contexte pour la compréhension de programmes logiciels en général et la normalisation du vocabulaire utilisé dans le code source systèmes logiciels en particulier.

Ainsi, nous proposons une approche contextuelle TIDIER, inspirée par les techniques de la reconnaissance de la parole et utilisant le contexte sous forme de dictionnaires spécialisés (*i.e.*, contenant des acronymes, abréviations et termes spécifiques au domaine des système logiciels). TIDIER est plus préformante que les approches qui la précédent (*i.e.*, CamelCase et samurai). Spécifiquement, TIDIER atteint 54% de précision en termes de décomposition des identifiants lors de l'utilisation un dictionnaire construit au niveau du système logiciel en question et enrichi par la connaissance du domaine. CamelCase et Samurai atteint seulement 30% et 31% en termes de précision, respectivement. En outre, TIDIER est la première approche qui met en correspondance les termes abrégés avec les concepts qui leurs correspondent avec une précision de 48% pour un ensemble de 73 abréviations.

La limitation principale de TIDIER est sa complexité cubique qui nous a motivé à proposer une solution plus rapide mais tout aussi performante, nommée TRIS. TRIS est inspirée par TIDIER, certes elle traite le problème de la normalisation différemment. En effet, elle le considère comme un problème d'optimisation (minimisation) dont le but est de trouver le chemin le plus court (*i.e.*, décomposition et extension optimales) dans un graphe acyclique. En outre, elle utilise la fréquence des termes comme contexte local afin de déterminer la normalisation la plus probable. TRIS est plus performante que CamelCase, Samurai et TIDIER, en termes de précision et de rappel, pour des systèmes logiciels écrits en C et C++. Aussi, elle fait mieux que GenTest de 4% en termes d'exactitude de décomposition d'identifiants. L'amélioration apportée par rapport à GenTest n'est cependant pas statistiquement significative. TRIS utilise une représentation basée sur une arborescence qui réduit considérablement sa complexité et la rend plus efficace en terme de temps de calcul. Ainsi, TRIS produit rapidement une normalisation optimale en utilisant un algorithme ayant une complexité quadratique en la longueur de l'identifiant à normaliser.

Ayant développé des approches contextuelles pour la normalisation, nous analysons alors son impact sur deux tâches de maintenance logicielle basées sur la recherche d'information, à savoir, la traçabilité des exigences au code source et la localisation des concepts. Nous étudions l'effet de trois stratégies de normalisation : CamelCase, Samurai et l'oracle sur deux techniques de localisation des concepts. La première est basée sur les informations textuelles

seulement, quant à la deuxième, elle combine les informations textuelles et dynamiques (traces d'exécution). Les résultats obtenus confirment que la normalisation améliore les techniques de localisation des concepts basées sur les informations textuelles seulement. Quand l'analyse dynamique est prise en compte, n'importe qu'elle technique de normalisation suffit. Ceci est du au fait que l'analyse dynamique réduit considérablement l'espace de recherche et donc l'apport de la normalisation n'est pas comparable à celui des informations dynamiques. En résumé, les résultats montrent l'intérêt de développer des techniques de normalisation avancées car elles sont utiles dans des situations où les traces d'exécution ne sont pas disponibles.

Nous avons aussi effectué une étude empirique sur l'effet de la normalisation sur la traçabilité des exigences au code source. Dans cette étude, nous avons analysé l'impact des trois approaches de normalisation précitées sur deux techniques de traçabilité. La première utilise une technique d'indexation sémantique latente (LSI) alors que la seconde repose sur un modèle d'espace vectoriel (VSM). Les résultats indiquent que les techniques de normalisation améliorent la précision et le rappel dans quelques cas. Notre analyse qualitative montre aussi que l'impact de la normalisation sur ces deux techniques de traçabilité dépend de la qualité des données étudiées.

Finalement, nous pouvons conclure que cette thèse contribue à l'état de l'art sur la normalisation du vocabulaire de code source et l'importance du contexte pour la compréhension de programmes logiciels. En plus, cette thèse contribue à deux domaines de la maintenance logicielle et spécifiquement à la localisation des concepts et à la traçabilité des exigences au code source. Les résultats théoriques et pratiques de cette thèse sont utiles pour les praticiens ainsi que les chercheurs.

Nos travaux de recherche futures relatifs à la compréhension de programmes logiciels et la maintenance logicielle consistent en l'évaluation de nos approches sur d'autres systèmes logiciels écrits en d'autres langages de programmation ainsi que l'application de nos approches de normalisation sur d'autres tâches de compréhension de programmes logiciels (*e.g.*, récapitulation de code source).

Nous sommes aussi en cours de la préparation d'une deuxième étude sur l'effet du contexte sur la normalisation du vocabulaire de code source en utilisant l'oculométrie afin de mieux analyser les stratégies adoptées par les développeurs lors de l'utilisation des informations contextuelles.

Le deuxième volet que nous avons entamé actuellement concerne l'impact des styles des identifiants sur la qualité des systèmes logiciels. En effet, nous sommes entrain d'inférer, en utilisant un modèle statistique (*i.e.*, modèle de Markov caché), les styles des identifiants adoptés par les développeurs dans les systèmes logiciels. Nous sommes également entrain d'étudier l'impact de ces styles sur la qualité des systèmes logiciels. L'idée est de montrer,

d'abord, si les développeurs utilisent leur propre style de nommage issu de leur propre expérience ou s'ils s'adaptent au projet, *i.e.*, aux conventions de nommage suivies (s'il y en a) et d'analyser, ensuite, les styles d'identifiants (*e.g.*, abréviations ou acronymes) qui mènent à l'introduction de bogues et à la dégradation des attributs de qualité internes, notamment, le couplage et cohésion sémantiques.

ABSTRACT

Understanding source code is a necessary step for many program comprehension, reverse engineering, or redocumentation tasks. In source code, textual information such as identifiers and comments represent an important source of information.

The problem of extracting and analyzing the textual information in software artifacts was recognized by the software engineering research community only recently. Information Retrieval (IR) methods were proposed to support program comprehension tasks, such as feature (or concept) location and traceability link recovery. However, to reap the full benefit of IR-based approaches, the language used across all software artifacts must be the same, because IR queries cannot return relevant documents if the query vocabulary contains words that are not in the source code vocabulary. Unfortunately, source code contains a significant proportion of vocabulary that is not made up of full (meaningful) words, *e.g.*, abbreviations, acronyms, or concatenation of these. In effect, source code uses a different language than other software artifacts. This vocabulary mismatch stems from the implicit assumption of IR and Natural Language Processing (NLP) techniques which assume the use of a single natural-language vocabulary. Therefore, vocabulary normalization is a challenging problem.

Vocabulary normalization aligns the vocabulary found in the source code with that found in other software artifacts. Normalization must both split an identifier into its constituent parts and expand each part into a full dictionary word to match vocabulary in other artifacts. In this thesis, we deal with the challenge of normalizing source code vocabulary by developing two novel context-aware approaches. We use the context because the results of our experimental studies have shown that context is relevant for source code vocabulary normalization. In fact, we conducted two user studies with 63 participants who were asked to split and expand a set of 50 identifiers from a corpus of open-source C programs with the availability of different context levels. In particular, we considered an internal context consisting of the content of functions, source code files, and applications where the identifiers appear and an external context involving external documentation. We reported evidence on the usefulness of contextual information for source code vocabulary normalization. We observed that the source code files are more helpful than just looking at function source code, and that the application-level contextual information does not help any further. The availability of external sources of information (*e.g.*, thesaurus of abbreviations and acronyms) only helps in some circumstances. The obtained results confirm the conjecture that contextual information is useful in program comprehension, including when developers split and expand identifiers to understand them. Thus, we suggest a novel contextual approach for vocabulary

normalization, TIDIER. TIDIER is inspired by speech recognition techniques and exploits contextual information in the form of specialized dictionaries (*e.g.*, acronyms, contractions and domain specific terms). TIDIER significantly outperforms its previous approaches (*i.e.*, CamelCase and Samurai which are the approaches that exist before TIDIER). Specifically, TIDIER achieves with a program-level dictionary complemented with domain knowledge, 54% of correct splits, compared to 30% obtained with CamelCase and 31% of correct splits attained using Samurai. Moreover, TIDIER was able to correctly map identifiers' terms to dictionary words with a precision of 48% for a set of 73 abbreviations. The main limitations of TIDIER is its cubic complexity that leads us to propose a fast solution, namely, TRIS.

TRIS is inspired by TIDIER, but it deals with the vocabulary normalization problem differently. It maps it to a graph optimization (minimization) problem to find the optimal path (*i.e.*, optimal splitting-expansion) in an acyclic weighted graph. In addition, it uses the relative frequency of source code terms as a local context to determine the most likely identifier splitting-expansion. TRIS significantly outperforms CamelCase and Samurai in terms of identifier splitting. It also outperforms TIDIER in terms of identifier expansion, with a medium to large effect size, for C and C++ systems. In addition, TRIS shows an improvement of 4%, in terms of identifier splitting correctness, over GenTest (a more recent splitter suggested after TIDIER). The latter improvement is not statistically significant. TRIS uses a tree-based representation that makes it—in addition to being more accurate than other approaches—efficient in terms of computation time. Thus, TRIS produces one optimal split and expansion fast using an identifier processing algorithm having a quadratic complexity in the length of the identifier to split/expand.

We also investigate the impact of identifier splitting on two IR-based software maintenance tasks, namely, feature location and traceability recovery. Our study on feature location analyzes the effect of three identifier splitting strategies: CamelCase, Samurai, and an Oracle on two feature location techniques (FLTs). The first is based on IR while the second relies on the combination of IR and dynamic analysis (*i.e.*, execution traces). The obtained results support our conjecture that when only textual information is available, an improved splitting technique can help improve effectiveness of feature location. The results also show that when both textual and execution information are used, any splitting algorithm will suffice, as FLTs produced equivalent results. In other words, because dynamic information helps pruning the search space considerably, the benefit of an advanced splitting algorithm is comparably smaller than that of the dynamic information; hence the splitting algorithm will have little impact on the final results. Overall, our findings outline potential benefits of creating advanced preprocessing techniques as they can be useful in situations where execution information cannot be easily collected.

In addition, we study the impact of identifier splitting on two traceability recovery techniques utilizing the same three identifier splitting strategies that we used in our study on feature location. The first traceability recovery technique uses Latent Semantic Indexing (LSI) while the second is based on Vector Space Model (VSM). The results indicate that advanced splitting techniques help increase the precision and recall of the studied traceability techniques but only in some cases. In addition, our qualitative analysis shows that the impact or improvement brought by such techniques depends on the quality of the studied data.

Overall, this thesis contributes to the state-of-the-art on identifier splitting and expansion, context, and their importance for program comprehension. In addition, it contributes to the fields of feature location and traceability recovery. Theoretical and practical findings of this thesis are useful for both practitioners and researchers.

Our future research directions in the areas of program comprehension and software maintenance will extend our empirical evaluations to other software systems belonging to other programming languages. In addition, we will apply our source code vocabulary normalization approaches on other program comprehension tasks (*e.g.*, code summarization).

We are also preparing a replication of our study on the effect of context on vocabulary normalization using Eye-Tracking to analyze the different strategies adopted by developers when exploring contextual information to perform identifier splitting and expansion.

A second research direction that we are currently tackling concerns the impact of identifier style on software quality using mining software repositories. In fact, we are currently inferring the identifier styles used by developers in open-source projects using a statistical model, namely, the Hidden Markov Model (HMM). The aim is to show whether open-source developers adhere to the style of the projects they join and their naming conventions (if any) or they bring their own style. In addition, we want to analyze whether a specific identifier style (*e.g.*, short abbreviations or acronyms) introduces bugs in the systems and whether it impacts internal software quality metrics, in particular, the semantic coupling and cohesion.

TABLE OF CONTENT

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

LIST OF ACRONYMS

NLP	Natural Language Processing
IR	Information Retrieval
FL	Feature Location
FLT	Feature Location Technique
LSI	Latent Semantic Indexing
VSM	Vector Space Model
LDA	Latent Dirichlet Allocation
TIDIER	Term IDentifier RecognIzER
TRIS	TRee-based Identifier Splitter
OSE	Optimal Splitting-Expansion
LINSEN	Linear IdeNtifier Splitting and Expansion
ANOVA	Analysis of Variance
HSD	Honest Significant Differences
OR	Odds Ratio
IDE	Integrated Development Environment
IT	Information Technology
CVS	Concurrent Versions System
SVN	Subversion
TF-IDF	Term Frequency Inverse Document Frequency
HMM	Hidden Markov Model

CHAPTER 1

INTRODUCTION

Source code vocabulary normalization consists of two tasks: splitting and expansion. Splitting divides identifiers into parts, and expansion expands parts that are abbreviations or acronyms into full words. For example, *compStats* is split into *comp-stats* and then expanded to *compute-statistics*. Most often, identifiers are not made up of full (natural-language) words and–or recognizable abbreviations. In fact, identifiers can be abbreviations such as *cntr* or acronyms like *cwdfn* and, thus, the context, *e.g.*, neighbor source code (including other identifiers), source code comments or external documentation can help expand them. To the best of our knowledge, no previous work has shown the relevance of context for source code vocabulary normalization. Thus, we conducted two user studies to show the extent to which different levels of context can help improve vocabulary normalization. The results bring empirical evidence on the usefulness of contextual information for identifier splitting and acronym/abbreviation expansion, they indicate that source code files are more helpful than functions, and that the application-level contextual information does not help any further (Guerrouj *et al.*, 2013b).

CamelCase, the widely adopted identifier splitting technique does not take into account context and it relies on the use of naming conventions (*e.g.*, CamelCase and–or separators). Samurai (Enslen *et al.*, 2009) is built on CamelCase and splits identifiers by mining term frequencies. It builds two term-frequency tables: a program-specific and a global-frequency table. The first table is built by mining terms in the program under analysis. The second table is made by mining the set of terms in a large corpus of programs. The main weakness of Samurai is its reliance on frequency tables. These tables could lead to different splits for the same identifier depending on the tables. Tables built from different programs may lead to different splits. Also, if an identifier contains terms with frequencies higher than the frequency of the identifier itself, Samurai may over-split it, thus providing several terms not necessarily reflecting the most obvious split (Enslen *et al.*, 2009). To overcome these shortcomings, we suggest two novel contextual approaches, TIDIER and TRIS. Our approaches perform both the splitting and expansion of identifiers even in the absence of naming conventions and the presence of abbreviations. More recently, other context-aware approaches have been suggested to normalize source code vocabulary, *e.g.*, Normalize (Lawrie et Binkley, 2011), a refinement of an identifier splitter, GenTest (Lawrie *et al.*, 2010), towards the expansion of identifiers using a machine translation technique, namely the maximum coherence model (Gao

et al., 2002) and LINSEN, a novel approach based on a graph model using an approximate string matching technique (Corazza *et al.*, 2012).

In this thesis, we also investigate the impact of identifier splitting on two software maintenance tasks, *i.e.*, feature location and traceability recovery. Specifically, our studies analyze the effect of three identifier splitting strategies: CamelCase, Samurai, and an Oracle (built using TIDIER). Our study on feature location used two FLTs. The first FLT is based on IR while the second combines IR and dynamic information, for locating bugs and features. The FLTs that use the simple CamelCase splitter were baselines in our studies (Dit *et al.*, 2011).

The study on traceability recovery uses two IR techniques, *i.e.*, LSI (Liu *et al.*, 2007) and VSM (Eaddy *et al.*, 2008a) while investigating the impact of the same identifier splitting strategies used in our study on feature location.

1.1 Challenges

Overall, the main challenges related to our thesis are:

- Very little empirical evidence on the extent to which context helps source code vocabulary normalization;
- Lack of context-aware source code vocabulary normalization approaches;
- Lack of studies on the impact of identifier splitting and expansion on IR-based software maintenance tasks.

The overarching research question addressed is:

> How to automatically resolve the vocabulary mismatch that exists between source code and other software artifacts, using context, to support software maintenance tasks such as feature location and traceability recovery?

1.2 Contributions

The main contribution of this thesis are an awareness of the context relevance for source code vocabulary normalization, context-aware approaches for vocabulary normalization, *i.e.*, TIDIER and TRIS, plus empirical evidence on the impact of identifier splitting on feature location and traceability recovery.

1.2.1 Context-Awareness for Source Code Vocabulary Normalization

To study the extent to which context helps when normalizing source code identifiers, *i.e.*, when splitting and expanding them, we performed two user studies with 63 participants, including Bachelor, Master, Ph.D. students, and post-docs. We randomly sampled a set of 50 identifiers from a corpus of open-source C programs, and we asked participants to split and expand them with the availability (or not) of internal and external contexts. In particular, we considered (i) an internal context consisting of the content of functions and source code files in which the identifiers are located, and (ii) an external context involving external documentation. The results of our studies show the usefulness of contextual information for identifier splitting and acronym/abbreviation expansion. We found that the source code files are more helpful than the functions and that the application-level contextual information does not help any further. The availability of external sources of information only helps in some circumstances. Overall, the obtained results confirm the conjecture that contextual information is useful in program comprehension, including when developers normalize source code identifiers to understand them (Guerrouj *et al.*, 2013b).

1.2.2 TIDIER

We propose TIDIER, an approach inspired by speech recognition techniques. It uses a thesaurus of words and abbreviations, plus a string-edit distance between terms and words computed via Dynamic Time Warping algorithm proposed by Herman Ney for connected speech recognition (*i.e.*, for recognizing sequences of words in a speech signal) (Ney, 1984). TIDIER exploits contextual information in the form of contextual dictionaries enriched by the use domain knowledge (*e.g.*, acronyms and domain specific terms). Its main assumption is that it is possible to mimic developers when creating an identifier relying on a set of word transformations.For example, to create an identifier for a variable that counts the number of software, a developer may drop vowels and (or) characters to shorten one or both words of the identifier, thus creating *bugsnbr*, *nbrofbugs*, or *numberBugs*. TIDIER significantly outperforms its prior approaches (*i.e.*, CamelCase and Samurai) on C systems. In addition, it reaches its best performances when using contextual-aware dictionaries enriched with domain knowledge (Guerrouj *et al.*, 2013a). However, TIDIER computation time increases with the dictionary size due to its cubic distance evaluation cost plus the search time.

1.2.3 TRIS

TRIS is a fast and accurate solution for vocabulary normalization. It uses the relative frequency of source code terms as a local context to determine the most likely identifier

splitting-expansion. TRIS takes as input a dictionary of words and the source code of the program to analyze. It represents transformations as a rooted tree where every node is a letter and every path in the tree represents a transformation having a given cost. Based on such transformations, possible splittings and expansions of an identifier are represented as an acyclic direct graph. Once such a graph is built, solving the optimal splitting and expansion problem means determining the shortest path, *i.e.*, the optimal split and expansion in the identifier graph (Guerrouj *et al.*, 2012).

1.2.4 Impact of Identifier Splitting on Feature Location

To analyze the impact of identifier splitting on feature location, we investigate three identifier splitting strategies, *i.e.*, CamelCase, Samurai, and an Oracle ("perfect split/expansion") built using TIDIER, on two feature location techniques for locating bugs and features. The first is based on LSI (Marcus et Maletic, 2003) while the second uses the combination of LSI and dynamic analysis (Poshyvanyk *et al.*, 2007). The results indicate that feature location techniques using IR can benefit from better preprocessing algorithms in some cases and that their improvement in effectiveness while using manual splitting over state-of-the-art approaches is statistically significant in those cases. However, the results for feature location technique using the combination of IR and dynamic analysis do not show any improvement while using manual splitting, indicating that any preprocessing technique will suffice if execution data is available (Dit *et al.*, 2011).

1.2.5 Impact of Identifier Splitting on Traceability Recovery

We also investigate the impact of splitting on two traceability recovery techniques. The first uses LSI (Liu *et al.*, 2007) while the second relies on VSM (Eaddy *et al.*, 2008a). We apply the three strategies we used in our study on feature location, *i.e.*, CamelCase, Samurai, and manual splitting of identifiers (built using TIDIER). The results demonstrate that advanced splitting techniques help increase the precision and recall of the studied traceability recovery techniques but only in a few cases. Our qualitative analysis shows that the impact of source code identifier splitting approaches or the improvement they brought depends also on the quality of the studied data.

1.3 Outline of the Thesis

Chapter 2 - Background: This chapter first defines the techniques and concepts used in this thesis. Then, it presents the source code vocabulary normalization approaches that

exist in the literature. Finally, the chapter explains the performance measures, statistical tests, and effect size measures used in our empirical and user studies.

Chapter 3 - Related Work: This chapter states the existing works in our research areas, it first presents the state-of-the-art on the role of textual information on program comprehension and software quality. Then, it shows the most relevant research contributions to context and program comprehension. The chapter also enumerates the existing source code vocabulary normalization approaches. Finally, this chapter presents related works on feature location and traceability.

Chapter 4 - Context-Awareness for Source Code Vocabulary Normalization: This chapter describes, in detail, our user studies on the effect of contexts on source code vocabulary normalization. It also shows the obtained quantitative and qualitative results plus the threats to validity related to our user studies.

Chapter 5 - Context-Aware Source Code Vocabulary Normalization: This chapter describes our context-aware source code vocabulary normalization approaches, *i.e.*, TIDIER and TRIS.

Chapter 6 - TIDIER and TRIS: Evaluation, Results, and Discussion: This chapter first presents the empirical studies performed to evaluate our vocabulary normalization approaches, *i.e.*, TIDIER and TRIS, the obtained results and their discussion. Then, it explains the threats to validity related to our studies.

Chapter 7 - Impact of Identifier Splitting on Feature Location: This chapter presents the empirical study we conducted to analyze the impact of source code identifier splitting on feature location. It also shows the obtained quantitative findings and the qualitative analysis performed in support of our quantitative analysis. The chapter enumerates some of the threats to validity related to this study.

Chapter 8 - Impact of Identifier Splitting on Traceability Recovery: This chapter describes the empirical study performed to analyze the impact of source code identifier splitting on traceability recovery. It shows both our quantitative and qualitative analyses. Then, the chapter explains some of the threats to validity related to this study. Finally, it concludes the work.

Chapter 9 - Conclusion and Future Work: Finally, this chapter revisits the main contributions of this thesis, explains our on-going work, and continues to describe potential opportunities for future research.

Appendix A: This appendix provides descriptive statistics of precision and recall obtained with TIDIER and its prior approaches.

Appendix B: This appendix provides details about the characteristics of the applications from which we sampled the identifiers used in our study on the effect of contexts on vocabulary

normalization. In addition, it shows the oracle of the used identifiers. Finally, this appendix shows the boxplots of precision and recall obtained with the different levels of context studied.

CHAPTER 2

Background

This chapter presents the main source code identifier splitting and expansion approaches suggested in the literature. It also provides details about the IR techniques used in our works. In addition, the chapter overviews some of the existing works in the fields of feature location and traceability. Furthermore, it explains the measures computed to evaluate the performance of our approaches and the statistical tests, plus effect-size measures used to compare our approaches with alternative ones.

2.1 Source Code Vocabulary Normalization

Vocabulary normalization consists of two tasks. The first task splits compound identifiers into their constituent terms. In the following, the strings of characters between division markers (*e.g.*, underscores and camel-casing) and the endpoints of an identifier are referred to as "hard-words" (Lawrie *et al.*, 2006). For example, *fix_bug* and *fixBug* include the hard-words *fix* and *bug*. Sometimes splitting into hard-words is sufficient (*e.g.*, when all hard-words are dictionary words); however, other times hard-word splitting is not sufficient, as with identifiers composed of juxtaposed lowercase words (*e.g.*, fixbug). In this case further division is required. The resulting strings of characters are referred to as "soft-words" (Lawrie *et al.*, 2006). Thus, a soft-word is either the entire hard-word or a sub-string of a hard-word. Let us consider, for example, the identifier *hashmap_entry*. This identifier consists of one division marker (an underscore) and, thus, two hard-words, *hashmap* and *entry*. The hard-word *hashmap* is composed of two soft-words, *hash* and *map*, while the hard-word *entry* is composed of a single soft-word.

The second task maps soft-words to their corresponding dictionary words and is helpful for programming languages (*e.g.*, C, C++) that favor the use of short identifiers. In such languages, the use of abbreviations and acronyms is likely a heritage of the past, when certain operating systems and compilers limited the maximum length of identifiers. In fact, a C developer may use *dir* instead of the hard-word *directory*, *pntr* instead of *pointer*, or *net* instead of *network*.

In the following, we will refer to any substring in a compound identifier as a *term* while an entry in a dictionary (*e.g.*, the English dictionary) will be referred to as a *word*. A term may or may not be a dictionary word. A term carries a single meaning in the context where it

is used while a word may have multiple meanings (upper ontologies like WordNet[1] associate multiple meanings to words).

In the following, we present the main approaches proposed to split and–or expand source code identifiers.

2.1.1 CamelCase

CamelCase is the de-facto splitting algorithm. This simple, fast, and widely used preprocessing algorithm has been previously applied in multiple approaches to feature location and traceability link recovery (Antoniol *et al.*, 2002; Marcus *et al.*, 2004, 2005; Liu *et al.*, 2007; Poshyvanyk *et al.*, 2007; Revelle et Poshyvanyk, 2009; Revelle *et al.*, 2010).

CamelCase splits compound identifiers according to the following rules:

RuleA: Identifiers are split by replacing underscore (*i.e.*, "_"), structure and pointer access (*i.e.*, "." and "->"), and special symbols (*e.g.*, $) with the space character. A space is inserted before and after each sequence of digits. For example, *counter_pointer4users* is split into *counter*, *pointer*, *4*, and *users* while *rmd128_update* is split into *rmd*, *128*, and *update*.

RuleB: Identifiers are split where terms are separated using the CamelCase convention, *i.e.*, the algorithm splits sequences of characters when there is a sequence of lower-case characters followed by one or more upper-case characters. For example, *counterPointer* is split into *counter* and *Pointer* while *getID* is split into *get* and *ID*.

RuleC: When two or more upper case characters are followed by one or more lower case characters, the identifier is split at the last-but-one upper-case character. For example, *USRPntr* is split into *USR* and *Pntr*.

Default: Identifiers composed of multiple terms that are not separated by any of the above separators are left unaltered. For example, *counterpointer* remains as it is.

Based on these rules, identifiers such as *FFEINFO_kindtypereal3*, *apzArgs*, or *TxRingPtr* are split into *FFEINFO kindtypereal*, *apz Args*, and *Tx Ring Ptr*, respectively. The CamelCase splitter cannot split same-case words, *i.e.*, *FFEINFO* or *kindtypereal* into terms, *i.e.*, the acronym *FFE* followed by *INFO* and the terms *kind*, *type*, and *real*.

The main shortcoming of CamelCase is its reliance on naming conventions.

[1] `http://wordnet.princeton.edu`

2.1.2 Samurai

Samurai (Enslen *et al.*, 2009) is an automatic approach to split identifiers into sequences of terms by mining terms frequencies in a large source code base. It relies on two assumptions:

1. A substring composing an identifier is also likely to be used in other parts of the program or in other programs alone or as a part of other identifiers.

2. Given two possible splits of a given identifier, the split that most likely represents the developer's intent partitions the identifier into terms occurring more often in the program. Thus, term frequency is used to determine the most-likely splitting of identifiers.

Samurai also exploits identifier context. It mines term frequency in the source code and builds two term-frequency tables: a program-specific and a global-frequency table. The first table is built by mining terms in the program under analysis. The second table is made by mining the set of terms in a large corpus of programs.

Samurai ranks alternative splits of a source code identifier using a scoring function based on the program-specific and global frequency tables. This scoring function is at the heart of Samurai. It returns a score for any term based on the two frequency tables representative of the program-specific and global term frequencies. Given a term t appearing in the program p, its score is computed as follows:

$$Score(t,p) = Freq(t,p) + \frac{globalFreq(t)}{log_{10}(AllStrsFreq(p))} \tag{2.1}$$

where:

- p is the program under analysis;
- $Freq(t,p)$ is the frequency of term t in the program p;
- $globalFreq(t)$ is the frequency of term t in a given set of programs; and
- $AllStrsFreq(p)$ is the cumulative frequency of all terms contained in the program p.

Using this scoring function, Samurai applies two algorithms, the *mixedCaseSplit* and the *sameCaseSplit* algorithm. It starts by executing the *mixedCaseSplit* algorithm, which acts in a way similar to the CamelCase splitter but also uses the frequency tables. Given an identifier, first, Samurai applies **RuleA** and **RuleB** from the CamelCase splitter: all special characters are replaced with the space character. Samurai also inserts a space character before and after each digit sequence. Then, Samurai applies an extension of **RuleC** to deal with multiple possible splits.

Let us consider the identifier *USRpntr*. **RuleC** would wrongly split it into *US* and *Rpntr*. Therefore, Samurai creates two possible splits: *US Rpntr* and *USR pntr*. Each possible term on the right side of the splitting point is then assigned a score based on Equation 2.1 and the highest score is preferred. The frequency of *Rpntr* would be much lower than that of *pntr*, consequently the most-likely split is obtained by splitting *USRpntr* into *USR* and *pntr*.

Following this first algorithm, Samurai applies the *sameCaseSplit* algorithm to find the split(s) that maximize(s) the score when splitting a same-case identifier, such as *kindtypereal* or *FFEINFO*. The terms in which the identifier is split can only contain lower-case characters, upper-case characters, or a single upper-case character followed by same-case characters.

The starting point of this algorithm is the first position in the identifier. The algorithm considers each possible split point in the identifier. Each split point would divide the identifier into a left-side and a right-side term. Then, the algorithm assigns a score for each possible left and right term and the split is performed where the split achieves the highest score. (Samurai uses a predefined lists[2] of common prefixes (*e.g.*, *demi*, *ex*, or *maxi*) and suffixes (*e.g.*, *al*, *ar*, *centric*, *ly*, *oic*) and the split point is discarded if a term is classified as a common prefix or suffix.)

Let us consider for example the identifier *kindtypereal* and assume that the first split is *kind* and *typereal*. Because neither *kind* nor *typereal* are common prefix/suffix, this split is kept. Now, let us further assume that the frequency of *kind* is higher than that of *kindtypereal* (*i.e.*, of the original identifier) and that the frequency of *typereal* is lower than that of *kindtypereal*. Then, the algorithm keeps *kind* and attempts to split *typereal* as its frequency is lower than that of the original identifier. When it will split *typereal* into *type* and *real*, the score of *type* and *real* will be higher than the score of the original identifier *kindtypereal* and of *typereal* and, thus, *typereal* will be split into *type* and *real*. Because the terms *kind*, *type*, and *real* have frequencies higher than that of *kindtypereal*, the obtained split corresponds to the expected result.

The main weakness of Samurai is the fact that it may oversplit identifiers in some cases. In fact, if an identifier contains terms with frequencies higher than the frequency of the identifier itself, Samurai may split it into several terms not necessarily reflecting the most obvious split.

In this work, we used the local and global frequency lists provided by the authors when dealing with the same Java systems used in their previous work (Enslen *et al.*, 2009). In all the other cases, we generated the local frequency table of the applications that we dealt with by mining terms frequencies in the application under analysis and we used as a global frequency list, a table generated by mining terms frequencies in a large corpus of GNU projects.

[2] `http://www.cis.udel.edu/~enslen/Site/Samurai_files/`

2.1.3 GenTest and Normalize

GenTest (Lawrie *et al.*, 2010) is an identifier splitter, which builds on ideas presented in two prior algorithms: Greedy and Samurai. The Greedy algorithm (Feild *et al.*, 2006) relies on a dictionary to determine where to insert a split in a hard-word. Samurai scores potential splits using the frequencies of the occurrence of strings from two sources: those appearing in the program being analyzed and those appearing in a large corpus of programs. GenTest is therefore used to accomplish the splitting task. The generation part of the algorithm generates all possible splittings. The test evaluates a scoring function against each proposed splitting. GenTest uses a set of metrics to characterize a high quality splitting of a hard-word. These metrics belongs to three categories: soft-word characteristics, metrics incorporating external information, and metrics incorporating internal information. Soft-word characteristics are characteristics of the strings produced by the splitting. External information includes dictionaries and other information that is either human engineered or extracted from non-source code sources. Internal information is derived from the source code, either the program itself or a collection of programs. Normalize (Lawrie et Binkley, 2011) is a refinement of GenTest to include source code identifier expansion. Thus, Normalize aligns the vocabulary found in source code with that found in other software artifacts. It is based on a machine translation technique, namely, the maximum coherence model (Gao *et al.*, 2002). The heart of normalization is a similarity metric computed from co-occurrence data. In other words, Normalize relies on the fact that expanded soft-words should be found co-located in general text. In the algorithm, the similarity between two expansions is the probability that the two expansions co-occur in a five word window in the Google data set (Brants et Franz, 2006). Co-occurrence data with contextual information has been exploited to select the best candidate among several possible expansions. Normalize has been recently applied to an IR-based tool with the aim of analyzing the impact of vocabulary normalization on feature location. Normalize was able to improve the ranks of relevant documents in the considered IR environment. This improvement was most pronounced for shorter, more natural, queries, where there is a 182% improvement (Binkley *et al.*, 2012).

2.1.4 LINSEN

LINSEN (Corazza *et al.*, 2012) is a novel technique that splits and expands source code identifiers; it is based on a graph model and performs the identifier splitting and expansion in linear time with respect to the size of the dictionary, taking advantage of an approximate string matching technique, the Baeza-Yates and Perleberg (Baeza-yates et Perleberg, 1992). The main advantage provided by such efficiency regards the possibility of exploiting a larger

number of dictionaries for the matching. In fact, LINSEN uses several dictionaries containing terms gathered from the source code comments, a dictionary of IT and programming terms, an English dictionary, and a list of well-known abbreviations. These sources are prioritized from the most specific to the most general one, with the idea that in presence of ambiguities, the most specific, domain dependent context should be preferred.

We share with these works the assumption that identifier splitting and expansion is essential for program comprehension as well as software maintenance and evolution tasks.

2.2 IR Techniques

IR is the activity of finding information resources (usually documents) of an unstructured nature (usually text) that satisfies an information need from large collections of information resources.

In this thesis, we use two IR techniques, in particular VSM (Antoniol *et al.*, 2002) and LSI (Marcus *et al.*, 2003). Both techniques essentially use term-by-document matrices. Consequently, we choose the well-known Term Frequency Inverse Document Frequency (TF-IDF) weighting scheme (Antoniol *et al.*, 2002). The latter measure and IR techniques are state-of-the-art for traceability recovery and feature location. In the following, we explain, in detail, these two techniques and the weighting scheme used.

2.2.1 Vector Space Model

VSM has been adopted in IR as a means of coping with inexact representation of documents and queries, and the resulting difficulties in determining the relevance of a document relative to a given query. In VSM (Baeza-Yates et Ribeiro-Neto, 1999; Antoniol *et al.*, 2002; De Lucia *et al.*, 2007), documents are represented as vector in the space of all the terms. Various term weighting schemes can be used to create these vectors. In our case, we use TF-IDF (Salton et Buckley, 1988) that we explain in Section 2.2.3. If a term belongs to a document, then it gets a non-zero value in the VSM along the dimension corresponding to the term. A document collection in VSM is represented by a term by document matrix, *i.e.*, $m \times n$ matrix, where m is the number of terms and n is the number of documents in the corpus.

Once documents are represented as vectors of terms in a VSM, traceability links are created between every pair of documents, *e.g.*, a requirement and a source code class, with different similarity values depending on each pair of documents. The similarity value between two documents is measured by the cosine of the angle between their corresponding vectors. Cosine values are in $[0, 1]$. Finally, the ranked list of recovered links and a similarity threshold

are used to select a set of candidate links to be manually verified (Antoniol *et al.*, 2002).

Let us consider R a requirement vector and C a source code vector. The similarity of the requirement R to source code C can be computed as follows (Baeza-Yates et Ribeiro-Neto, 1999).

$$sim(R, C) = \frac{R \cdot C}{\|R\| \cdot \|C\|} = \frac{\sum_{t_{i \in R}} w_{t_{iR}} \cdot \sum_{t_{i \in C}} w_{t_{iC}}}{\sqrt{\sum_{t_{i \in R}} w_{t_{iR}}^2} \cdot \sqrt{\sum_{t_{i \in C}} w_{t_{iC}}^2}} \tag{2.2}$$

where $w_{t_{iR}}$ is the weight of the i^{th} term in the query vector R, and $w_{t_{iC}}$ is the weight of the i^{th} term in the query vector C. Smaller the vector angle is, higher the similarity between two documents.

2.2.2 Latent Semantic Indexing

LSI overcomes the shortcoming of VSM, which does not address the synonymy and polysemy problems and relations between terms (Deerwester *et al.*, 1990). It assumes that there is an underlying latent structure in word usage for every document set (Deerwester *et al.*, 1990) and works as follows:

The processed corpus is transformed into a term-by-document ($m \in x$) matrix A, where each document is represented as a vector of terms. The values of the matrix cells are the weights of the terms, which are computed using the traditional TF-IDF weighting schemes (cf. Section 2.2.3 of this chapter) in our studies.

The matrix is then decomposed, using Singular Value Decomposition (SVD) (Deerwester *et al.*, 1990), into the product of three other matrices:

$$A = U \times S \times V \tag{2.3}$$

where U is the $m \times r$ matrix of the terms (orthogonal columns) containing the left singular vectors, V is the $r \times n$ matrix of the documents (orthogonal columns) containing the right singular vectors, S is an $r \times r$ diagonal matrix of singular values, and r is the rank of A. To reduce the matrix size, all the singular values in S are ordered by size. All the values after the largest k value could be set to zero. Thus, deleting the zero rows and columns of S and corresponding columns of U and rows of V would produce the following reduced matrix:

$$A_k = U_k \times S_k \times V_k \tag{2.4}$$

where the matrix A_k is approximately equal to A and is of rank $k < r$. The choice of k value, *i.e.*, the SVD reduction of the latent structure, is still an open issue in the natural language processing literature.

2.2.3 Term Frequency Inverse Document Frequency weighting scheme

TF-IDF (Salton et Buckley, 1988) is the standard weighting scheme adopted in IR and also software engineering research. TF-IDF emphasizes terms that appear frequently in a document but decreases the contribution of terms that are common across all documents. In this scheme, documents in the matrix are normalized by setting the most common term to 1 and dividing all of the other terms in the document by its former value (Equation 2.5). This results in a document consisting of term frequencies (tf). In fact, the term frequency $tf(f, d)$ uses the raw frequency of a term in a document, *i.e.*, the number of times that term t occurs in document d. If we denote the raw frequency of t by $f(t, d)$, then $tf(f, d)$ is the raw frequency $f(t, d)$ divided by the maximum raw frequency of any term in the document (Equation 2.5). The document frequencies (df) are computed by recording the number of documents a term occurs in (Equation 2.6). The df are used to calculate the inverse document frequencies (idf) (Equation 2.7). The inverse document frequency is a measure of whether a term is common or rare across all documents. It is obtained by dividing the total number of documents by the number of documents containing a term (*i.e.*, df), and then taking the logarithm of that quotient. Finally, each tf weighted term in the document is multiplied by its idf, resulting in a TF-IDF weight for each term in the document (Equation 2.8).

$$tf(t, d) = \frac{f(t, d)}{max\{f(w, d) \mid w \in d\}} \tag{2.5}$$

$$df(t) = |\{t \in d, d \in \{D\} \mid tf(t, d) \neq 0\}| \tag{2.6}$$

$$idf(t, \{D\}) = \log_2 \frac{|D|}{df(t)} \tag{2.7}$$

$$TF - IDF(t, d, \{D\}) = tf(t, d) \times idf(t, \{D\}) \tag{2.8}$$

2.3 Feature Location

In software systems, a feature represents a functionality that is defined by software requirements and accessible to end users. Software maintenance and evolution involves adding new features to programs, improving existing features, and removing unwanted features (*e.g.*, bugs). The practice that consists of identifying code elements that implement a specific feature is known as feature location (Biggerstaff *et al.*, 1994; Rajlich et Wilde, 2002). In our study described in Chapter 7, we rely on two feature location approaches. The first uses IR while the second combines IR and dynamic analysis. While there are several IR techniques

that have been successfully applied in the context of feature location, such as VSM (Eaddy *et al.*, 2008a), LSI (Liu *et al.*, 2007; Poshyvanyk *et al.*, 2007; Revelle et Poshyvanyk, 2009; Revelle *et al.*, 2010), and LDA (Lukins *et al.*, 2010), in this thesis, we focus on evaluating LSI for feature location. LSI-based feature location follows five main steps: generating a corpus, preprocessing the corpus, indexing the corpus using LSI, formulating a search query and generating similarities and finally, examining the results.

Step one - generating the corpus. The source code of a software system is parsed, and all the information associated with a method (*i.e.*, comments, method declaration, signature and body) will become a document in the system corpus. In other words, we are using a method-level granularity for the corpus, so each method from the source code has a corresponding document in the corpus.

Step two - preprocessing the corpus. The generated corpus is then preprocessed in order to normalize the text contained in the documents. This step includes removing operators, programming language keywords, or special characters. Additionally, compound identifiers could be split using different identifier splitting techniques. The split identifiers are then stemmed (*i.e.*, reduced to their root form) using the Porter stemmer (Porter, 1980), and finally the words that appear commonly in English (*e.g.*, "a", "the", etc.) are eliminated.

Step three - indexing the corpus using LSI. The preprocessed corpus is transformed into a term-by-document matrix, where each document (*i.e.*, method) from the corpus is represented as a vector of terms (*e.g.*, identifiers). The values of the matrix cells represent the weights of the terms from the documents, which are computed using the term frequency-inverse document frequency TF-IDF weight. The matrix is then decomposed using Singular Value Decomposition (Deerwester *et al.*, 1990) which decreases the dimensionality of the matrix by exploiting statistical co-occurrences of related words across the documents.

Step four - formulating a search query and generating similarities. The software developer chooses a query that describe the feature or bug being sought (*e.g.*, "print page"). The query is converted into a vector-based representation, and the cosine similarity between the query and every document in the reduced space is computed. In other words, the textual similarity between the bug description and every method from the software system is computed in the LSI subspace.

Step five - examining the results. The list of methods is ranked based on their cosine similarities with the user query. The developer starts investigating the methods in order, from the top of the list (*i.e.*, most relevant methods first). After examining each method the developer decides if that method belongs to the feature of interest or not. If it does, the feature location process terminates. Otherwise, the developer can continue examining other methods, or refine the query based on new information gathered from examining the methods

and starting from Step 4 again.

Feature location via LSI and dynamic information has one additional step, which can take place before the Step 4 described earlier.

Step for collecting execution information. The software developer triggers the bug, or exercises the feature by running the software system and executing the steps to reproduce from the description of the feature or bug. This process invokes the methods that are responsible for the bug or feature and, these methods are collected in an execution trace. The developer can take advantage of this information by formulating a query (Step 4) and examining the results (Step 5) produced by ranking only the methods found in the execution trace (as opposed to ranking all the methods of the software system). The advantage of using execution information is that it reduces the search space, thus increasing the performance of feature location.

We consider both FLT based on IR only and FLT based on the combination of IR and execution information. While previous studies have shown that the FLT based on execution information outperforms its basic version (*i.e.*, FLT based on IR only) (Liu *et al.*, 2007; Poshyvanyk *et al.*, 2007; Revelle et Poshyvanyk, 2009; Revelle *et al.*, 2010), the goal of the study described in this thesis (cf. Chapter 7) was to analyze the impact of the identifier splitting techniques from Step 2 on the accuracy of feature location.

2.4 Traceability Recovery

Requirement traceability is defined as "the ability to describe and follow the life of a requirement", in both forwards and backwards directions (*i.e.*, from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases) (Gotel et Finkelstein, 1994).

Promising results have been achieved using IR methods (*e.g.*, (Antoniol *et al.*, 2002)), because pairs of source-target artifacts having higher textual similarities have a high probability to be linked. A premise of the latter work is that programmers use meaningful names for program items, such as functions, variables, types, classes, and methods. In fact, Antoniol *et al.* believe that the application-domain knowledge that programmers process when writing the code is often captured by the mnemonics for identifiers; therefore, the analysis of these mnemonics can help to associate high-level concepts with program concepts and vice-versa (Antoniol *et al.*, 2002).

Recently, researchers (De Lucia *et al.*, 2011) have used smoothing filters to improve the precision of IR-based traceability. In addition to these technical improvements, other works have focused on human factors in traceability, such as how to help programmers understand

how to use the links for a specific task (Hayes *et al.*, 2004; Mader *et al.*, 2009; Panis, 2010; De Lucia *et al.*, 2011).

In the fields of feature location, we are the first to analyze the impact of identifier splitting on such a task (Dit *et al.*, 2011). We investigate the impact of three different identifier splitting strategies, namely, CamelCase, Samurai, and manual splitting of identifiers on two LSI-based FLTs. One based on IR only while the second uses IR and dynamic analysis. Recently, Binkley *et al.* (Binkley *et al.*, 2012) replicated an experiment with an LSI-based feature locator performed by Marcus *et al.* (Marcus *et al.*, 2004). They applied their identifier splitting and expansion technique, *i.e.*, Normalize (Lawrie et Binkley, 2011) on this IR-based tool. The results of their study show that normalization is able to recover key domain terms that were shrouded in invented vocabulary, thus, it was able to improve the ranks of relevant documents in the IR environment considered. However, this improvement was most pronounced for shorter, more natural queries where there was a 182% improvement (Binkley *et al.*, 2012).

2.5 Building Dictionaries

Our source code vocabulary normalization, *i.e.*, TIDIER (Guerrouj *et al.*, 2013a) and TRIS (Guerrouj *et al.*, 2012) aim at expanding identifiers by trying to match their terms with words contained in a dictionary. To perform the expansion task, we use dictionaries built for the analyzed software systems. Dictionaries are built by tokenizing source code, extracting identifiers and comment terms, and saving them into specialized dictionaries (Guerrouj *et al.*, 2013a, 2012). In TIDIER, we also built context-aware dictionaries at the level of functions, files, or C programs since one of our objectives was to analyze the sensitiveness of TIDIER to contextual information. The context-aware dictionaries construction phase will be explained in details in the chapter dedicated to TIDIER (cf. Chapter 5).

2.6 Building Oracles

To validate the obtained identifier splitting results, we need an oracle. This means that for each identifier, we will have a list of terms obtained after splitting it and, wherever needed, expanding contracted words. We produce the oracle following a consensus approach: (i) a splitting of each sampled identifier, and expanded abbreviations is produced independently (ii) In a few cases, disagreements are discussed among all the authors.

We adopted this approach in order to minimize the bias and the risk of producing erroneous results. This decision was motivated by the complexity of identifiers, which capture developers domain and solution knowledge, experience, and personal preference (Dit *et al.*,

2011; Guerrouj *et al.*, 2012, 2013a,b).

2.7 Generating Traceability Links' Sets

To investigate the impact of identifier splitting on traceability recovery, we need to evaluate the performance of the studied traceability recovery techniques (*e.g.*, one using CamelCase and the other Samurai or the Oracle). To do so, we generate various traceability links' sets at different thresholds. We then use these sets to compute precision, recall, and F-measure values. These sets help us to evaluate, which approach is better than the other at all the threshold values or some specific thresholds values. To perform statistical tests, we conduct several experiments with different threshold values on links recovered by two traceability recovery techniques. In the software engineering literature, three main threshold strategies have been suggested by researchers:

Scale threshold: It is computed as the percentage of the maximum similarity value between two software artefacts, where threshold t is $0 \leq t \leq 1$ (Antoniol *et al.*, 2002). In this case, the higher the value of the threshold t, the smaller the set of recovered links returned by an IR query.

Constant threshold: It has values between $[0, 1]$ (Marcus et Maletic, 2003); a widely used threshold is $t = 0.7$. However, the latter value is not convenient when the maximum similarity between two software artefacts is less than 0.7.

Variable threshold: It is an extension of the constant threshold approach (De Lucia *et al.*, 2004). When using a variable threshold, the constant threshold is projected onto particular interval, where the lower bound is the minimum similarity and upper bound is the maximum similarity between two software artefacts. Hence, the variable threshold has values between 0% to 100% and on the basis of this value this method determines a cosine threshold.

In Chapter 8 of this thesis, we use the scale threshold. We considered a threshold t to prune the set of traceability links, keeping only links whose similarities values are greater than or equal to $t \in [0, 1]$. We used different values of t from 0.01 to 1 per step of 0.01 to obtain different sets of traceability links with varying precision, recall, and–or F-measure values, for our approaches.

2.8 Performance Measures

2.8.1 Correctness of Identifier Splitting/Expansion

In some of our empirical evaluations, we compute the *correctness* of the splitting/mapping to dictionary words produced by the identifier splitting/expansion approach with respect to the oracle. To do so, we use a Boolean variable meaning that the split/expansion is correct (true) or not (false).

Let us define the correct expansion of the identifier *cntrPtr* as *counter* and *pointer*; if the studied approach produces exactly the expected expansions and, thus, the correct splits, then the correctness is true, else it is false, *e.g.*, *counter* and *ptr*. The weakness of this correctness measure is that it only provides a Boolean evaluation of the splitting/expansion. If the split is *almost* correct, *i.e.*, most of the terms are correctly identified, then correctness would still be false.

2.8.2 Precision, Recall, and F-measure of Identifier Splitting/Expansion

To overcome the limitation of the correctness measure and provide a more insightful evaluation, we use the *precision* and *recall* measures.

Given an identifier s_i to be split, $o_i = \{oracle_{i,1}, \ldots oracle_{i,m}\}$ the splitting in the manually-produced oracle, and $t_i = \{term_{i,1}, \ldots term_{i,n}\}$ the set of terms obtained by an approach, we define the precision and recall as follows:

$$precision_i = \frac{|t_i \cap o_i|}{|t_i|}, \quad recall_i = \frac{|t_i \cap o_i|}{|o_i|}$$

To provide an aggregated, overall measure of precision and recall, we use the F-measure, which is the harmonic mean of precision and recall:

$$F - measure_i = \frac{2 \cdot precision_i \cdot recall_i}{precision_i + recall_i}$$

2.8.3 Effectiveness Measure of Feature Location

In our empirical study on feature location and identifier splitting, we use the effectiveness measures (Liu *et al.*, 2007) to compare which configuration of the considered FLTs is more accurate than another. The effectiveness measure is the best rank (*i.e.*, lowest rank) among all the methods from the gold set for a specific feature. Intuitively, the effectiveness measure shows how many methods must be investigated before the first method relevant to the feature is located (Eisenbarth *et al.*, 2003; Antoniol et Guéhéneuc, 2005). Obviously, a technique that consistently places relevant methods towards the top of the ranked list (*i.e.*, lower ranks) is

more effective than a technique that contains relevant methods towards the middle or the bottom of the ranked list (*i.e.*, higher ranks).

Formally, we define the effectiveness of a technique j, E_j, as the rank $r(m_i)$ of the method m_i where m_i is the top ranked method among the methods that must be considered for a specific feature.

We consider the effectiveness measure of FLTs because, first, we are focusing on concept location, rather than impact analysis. Second, once a relevant method has been identified, it is much easier to find other related methods by following program dependencies from the relevant method, or by using other heuristics.

2.8.4 Precision and Recall of Traceability Recovery

In our empirical study on traceability recovery and identifier splitting, we use two well-known IR metrics, namely, precision and recall, to evaluate the accuracy of our experiment results. Both measures produce values in the interval $[0, 1]$. Precision and recall values are computed for all the traceability links retrieved above a threshold. The threshold value could be determined based on the project scope and–or retrieved documents.

$$precision = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{retrieved\ documents\}|} \tag{2.9}$$

Precision is defined as the total number of relevant documents retrieved divided by the total number of retrieved documents by an approach. Precision considers all retrieved documents above the threshold value. This measure is called precision at n or $P@n$. A precision value equal to 1 means that all the recovered documents are correct.

$$recall = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{relevant\ documents\}|} \tag{2.10}$$

Recall is defined as the relevant documents retrieved divided by the total number of relevant documents. It is, in fact, the ratio between the number of documents that are successfully retrieved and the number of documents that should be retrieved. A recall of a value equal to 1 means that all relevant documents have been retrieved. The obtained documents are, in general, the result of an IR query execution.

2.9 Statistical Hypothesis Testing

In this thesis, to compare the performance of an approach to another, we use statistical hypothesis testing. To perform statistical tests, we first formulate a null hypothesis, *e.g.*, there is no difference between Samurai and GenTest in terms of their identifier splitting

correctness. To reject a null hypothesis, we define a significance level of a test, *i.e.*, *alpha*. It is an upper bound of the probability for rejecting the null hypothesis. Second, we analyze whether the the data is normally distributed or not to select an appropriate statistical test. Finally, we perform a convenient statistical test to get a probability value, *i.e.*, p-value, to verify our hypothesis; p-value is compared against the significance level. We reject the null hypothesis if the p-value is less than the significance level, *e.g.*, 0.05. Otherwise, we accept the alternate hypothesis or provide an explanation if we do not reject the null hypothesis.

In this thesis, we perform a set of statistical tests depending on the addressed problem. These tests allow us to assess whether the obtained results are statistically significant or not. In addition, we measure the effect size (*i.e.*, the magnitude) of the difference between two approaches.

2.9.1 Statistical Tests

We perform appropriate statistical tests to analyze whether the improvement in accuracy brought by a proposed approach is indeed an improvement or it is obtained by chance. In the following, we discuss the statistical tests used in this thesis.

Fisher Exact Test

Fisher's exact test is a non-parametric test, which evaluates the hypothesis of independence between two categorical random variables. We use it in our thesis to test the differences among different identifier splitting and expansion approaches, in terms of identifier splitting/expansion correctness since correctness is a categorical measure. Specifically, we use Fisher's exact test to test the following null hypothesis H_0*: the proportion of correct splits/expansions between two approaches do not significantly change.*

Wilcoxon Paired Test

The Wilcoxon paired test is a non-parametric test for pair-wise median comparison, it reports whether the median difference between two approaches is significantly different from zero: $H_0 : \mu_d = 0$, where μ_d is the median of the differences (Wohlin *et al.*, 2000).

Wilcoxon Signed-rank Test

The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used when comparing two related samples, matched samples, or repeated measurements on a single sample to assess whether their population mean ranks differ (*i.e.*, it is a paired difference test) (Wohlin *et al.*, 2000).

Two-way Analysis of Variance Test

The two-way analysis of variance (ANOVA) is a parametric test, which is an extension of the one-way ANOVA test that examines the influence of different categorical independent variables on one dependent variable. While the one-way ANOVA measures the significant effect of one independent variable, the two-way ANOVA is used when there are more than one independent variable and multiple observations for each independent variable. The two-way ANOVA does not only determine the main effect of contributions of each independent variable but also analyzes if there is a significant interaction effect between the independent variables (Wohlin *et al.*, 2000).

Permutation Test

The permutation test is a non-parametric alternative to ANOVA; differently from ANOVA, it does not require data to be normally distributed. The general idea behind such a test is that the data distributions are built and compared by computing all possible values of the statistical test while re-arranging the labels (representing the various factors being considered) of the data points (Baker, 1995).

Tukey's Honest Significant Differences Test

Tukey's Honest Significant Differences (HSD) is a post-hoc testing process which allows the comparison of all pairs of groups while preserving the total importance degree of the set of analyses at a prescribed degree. It is a single-step multiple comparison procedure and statistical test used in conjunction with an ANOVA to find means that are significantly different from each other. In fact, Tukey's test compares the means of every treatment to the means of every other treatment; that is, it applies simultaneously to the set of all pairwise comparisons, and identifies any difference between two means that is greater than the expected standard error. The confidence coefficient for the set, when all sample sizes are equal, is exactly 1-*alpha*. For unequal sample sizes, the confidence coefficient is greater than 1-*alpha* (Sheskin, 2007).

Mann-Whitney test

The Mann-Whitney is a non-parametric test, which assesses how many times a set Y precedes a set X in two samples. It is a robust statistical test and could also be used for small sample sizes, 5 to 20 samples. Mann-Whitney could also be used when the sample values are captured using an arbitrary scale that cannot be measured accurately (Wohlin *et al.*, 2000).

2.9.2 Effect Size Measures

Other than showing the presence of significant differences between studied approaches, we also analyze the magnitude of the detected differences using appropriate effect-size measures.

Cohen's d

Cohen's d is parametric effect size measure for dependent variables, defined as the difference between two means (M_1 and M_2), divided by the standard deviation of the (paired) differences between samples (σ_D):

$$d = \frac{M_1 - M_2}{\sigma_D}$$

The Cohen's d effect size is considered small for $0.2 \leq d < 0.5$, medium for $0.5 \leq d < 0.8$, and large for $d \geq 0.8$ (Cohen, 1988).

Cliff's delta

Cliff's delta (d) is a non-parametric effect size measure (Grissom et Kim, 2005), defined as the probability that a randomly-selected member of one sample has a higher response than a randomly-selected member of a second sample, minus the reverse probability. Cliff's d ranges in the interval $[-1, 1]$ and is considered small for $0.148 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$.

Odds Ratio

Odds ratio (OR) is defined as the ratio of the odds p of an event occurring in one group to the odds q to it occurring in another group:

$$OR = \frac{p/(1-p)}{q/(1-q)}$$

$OR = 1$ indicates that the event is equally likely in both samples. $OR > 1$ indicates that the event is more likely with the first approach while an $OR < 1$ indicates the opposite Sheskin (2007).

2.9.3 Multiple Testing p-value Corrections

Whenever multiple tests are performed, we adjust p-values using the appropriate corrections.

Bonferroni correction

Bonferroni correction is an adjustment made to p-values when several dependent or independent statistical tests are being performed simultaneously on a single data set. To perform a Bonferroni correction, the critical p-value should be divided by the number of performed comparisons (*i.e.*, tests) (Bland, 2000).

Holm correction

Holm correction is similar to the Bonferroni correction, but less stringent. It works as follows: (i) the p-values obtained from multiple tests are ranked from the smallest to the largest, (ii) the first p-value is multiplied by the number of tests performed (n), and is deemed to be significant if it is less than 0.05, and (iii) the second p-value is multiplied by $n - 1$, and so on (Holm, 1979).

Our statistics have been applied using the *R* (version 2.15.0) statistical environment (Team, 2012).

CHAPTER 3

Related Work

This chapter describes the most relevant contributions to the role of textual information on program comprehension and software quality, source code vocabulary normalization approaches, relevance of context for program comprehension, as well as feature location and traceability recovery.

3.1 Role of Textual Information on Program Comprehension and Software Quality

Early work (Soloway *et al.*, 1983; Mayrhauser et Vans, 1995) on program comprehension and mental models, which are programmers' mental representation of the program being maintained, highlighted the significance of textual information to capture and encode programmers' intent and knowledge in software. The role of identifier naming was also investigated by Anquetil *et al.* (Anquetil et Lethbridge, 1998), who suggested the existence—in the source code lexicon—of "hard-terms" that encode core concepts.

Takang *et al.* (Takang *et al.*, 1996) empirically studied the role of identifiers and comments on source code understanding. They compared abbreviated identifiers to full-word identifiers and uncommented code to commented code. The results of their study showed that commented programs are more understandable than non-commented programs, and that programs containing full-word identifiers are more understandable than those with abbreviated identifiers.

Caprile and Tonella (Caprile et Tonella, 1999) performed an in-depth analysis of the internal structure of identifiers. They showed that identifiers are an important source of information about system concepts and that the information they convey is often the starting point of program comprehension. Other researchers (Caprile et Tonella, 2000; Merlo *et al.*, 2003) assessed the quality of identifiers, their syntactic structure, plus the information carried by the terms that compose them.

Deißenböck *et al.* (Deißenböck et Pizka, 2005) provided a set of guidelines to produce high-quality identifiers. With such guidelines, identifiers should contain enough information for a software engineer to understand the program concepts.

Lawrie *et al.* (Lawrie *et al.*, 2007b) attempted to assess the quality of source code identifiers. They suggested an approach, named QALP (Quality Assessment using Language

Processing), relying on the textual similarity between related software artifacts. The QALP tool leverages identifiers and related comments to characterize the quality of a program. The results of their empirical study indicated that full words as well as recognizable abbreviations contribute to better program understanding. Their work suggested that the recognition of words composing identifiers, and, thus, of the domain concepts associated with them could contribute to a better comprehension.

Methods related to source code identifier refactoring were proposed by Caprile and Tonella (Caprile et Tonella, 1999) and Demeyer *et al.* (Demeyer *et al.*, 2000).

De Lucia *et al.* (De Lucia *et al.*, 2006, 2010) proposed COCONUT, a tool highlighting to developers the similarity between source code identifiers and comments and words in high-level artifacts. They empirically showed that this tool is helpful to improve the overall quality of identifiers and comments.

Textual similarity between methods within a class, or among methods belonging to different classes, has been used to define new measures of cohesion and coupling, *i.e.*, the Conceptual Cohesion of Classes proposed by Marcus *et al.* (Marcus *et al.*, 2008) and the Conceptual Coupling of Classes proposed by Poshyvanyk *et al.* (Poshyvanyk et Marcus, 2006), which bring information complementary to structural cohesion and coupling measure.

De Lucia *et al.* (De Lucia *et al.*, 2007) used LSI to identify cases of low similarity between artifacts previously traced by software engineers. Their technique relies on the use of textual similarity to perform an off-line quality assessment of both source code and documentation, with the objective of guiding a software quality review process because the lack of textual similarity may be an indicator of low quality of traceability links. In fact, poor textual description in high-level artifacts or meaningless identifiers or poor comments in source code may point to a poor development process and unreliable traceability links.

Abebe *et al.* (Abebe *et al.*, 2008) analyzed how the source code vocabulary changes during evolution. They performed an exploratory study of the evolution of two large open-source programs. The authors observed that the vocabulary and the size of a program tend to evolve the same way and that the evolution of the source code vocabulary does not follow a trivial pattern. Their work was motivated by the importance of having meaningful identifiers and comments, consistent with high-level artifacts and with the domain vocabulary during the life of a program.

Abebe *et al.* (Abebe *et al.*, 2012) measured the quality of identifiers using the number of Lexicon Bad Smells (LBS) they contain. They investigated whether using LBS in addition to structural metrics improves fault prediction. To conduct their investigation, the authors assessed the prediction capability of a model while using only structural metrics, and structural metrics and LBS. The results of their study indicated that there is an improvement in

the majority of the cases.

Binkley *et al.* (Binkley *et al.*, 2009) investigated the use of the identifier separators, namely the CamelCase convention and underscores in program comprehension. They found that the CamelCase convention led to better understanding than underscores, and when participants are properly trained, that participants performed faster with identifiers built using the CamelCase convention rather than those with underscores.

Sharif and Maletic (Sharif et Maletic, 2010) replicated the pervious study using an eye-tracking system. The results of their study showed that participants recognized identifiers that used the underscore notation more quickly. They also reported that there is no difference in terms of accuracy between the CamelCase and underscore style.

A recent work by Binkley *et al.* (Binkley *et al.*, 2013) studied the impact of identifier style on program comprehension. In this work, the authors conducted five studies with 150 participants and examined two styles, namely the usage of CamelCase and underscore as separators. Their hypothesis was that the style of identifiers affects the speed and accuracy of comprehending source code. Their first study, which investigated how well humans read identifiers in the two different styles, focused on low-level readability issues. The remaining four studies built on the first to focus on the semantic implications of identifier style. The results of their studies showed that the tasks of reading and comprehending source code is fundamentally different from those of reading and comprehending natural language. In addition, the authors highlighted that as the task becomes similar to reading prose, the results become similar to work on reading natural-language text. Furthermore, the authors showed that, for more "source focused" tasks, identifier style affects only non-experienced software developers, who benefitted from the use of Camel casing, however, experienced ones appear to be less affected.

Overall, prior works agree on the fact that identifiers represent an important source of domain information, and that meaningful identifiers improve software quality and reduce the time and effort to acquire a basic comprehension level for any maintenance task.

3.2 Context Relevance for Program Comprehension

Many cognitive models have been recently proposed in the literature, and they all rely on the programmers' own knowledge, the source code and available software documentation (Mayrhauser et Vans, 1995). Disparities between various comprehension models can be explained in terms of differences in experimental factors such as programmer characteristics, program characteristics and task characteristics that influence the comprehension process (M-A.D. Storey, 1999).

Robillard *et al.* (Robillard *et al.*, 2004) performed an exploratory study to assess how developers investigate context, more precisely, source code when performing a software change task. Their study involved five developers performing a change task on a medium-size open source system. In their study, they isolated the factors related to effective program investigation behavior by performing a detailed qualitative analysis of the program investigation behavior of successful and unsuccessful developers. Their results support the intuitive notion that a methodical and structured approach to program investigation is the most effective. Their main findings related to our work is the fact that prior to performing a task, developers must discover and understand the subset of the system relevant to the task. Thus, task context is important to understand the task at hand and avoid information overload.

Kersten *et al.* (Kersten et Murphy, 2006) presented a mechanism that captures, models, and persists the elements and relations relevant to a task. They showed how their task context model reduces information overload and focuses a programmers' work by filtering and ranking the information presented by the development environment. They implemented their task context model as a tool, Maylar, for the Eclipse development environment. In their study, a task context represents the program elements and relationships relevant to completing a particular task.

Sillito *et al.* (Sillito *et al.*, 2008) provided an empirical foundation for tool design based on an exploration of what programmers need to understand and of how they use tools to discover that information while performing a change task. They collected and analyzed data from two observational studies. Their first study was carried out in a laboratory setting. However, the second study was carried out in an industrial work setting. The participants in the first study were observed as they worked on assigned change tasks to a code base that was new to them. The results of their study provide a more complete understanding of the information needed by programmers performing change tasks, and of how programmers use tools to discover that information. These results have several implications for tool design. In fact, they point to the need to move tools closer to programmers' questions and the process of answering those questions and also suggest ways that tools can do this, for example, by maintaining and using more types of context and by providing support for working with larger and more diverse groups of entities and relationships.

We share with the above-mentioned works the idea that task context is important when performing a software evolution task. However, our focus was not on building task context models for programming tasks but rather on discovering the contexts relevant for source code vocabulary normalization to help improve the accuracy of source code vocabulary normalization approaches and tools.

3.3 Source Code Vocabulary Normalization Approaches

Stemming from Deißenböck and Pizka's observation on the relevance of identifiers' terms for program comprehension, several approaches have been proposed to normalize source code vocabulary. These approaches are CamelCase, Samurai, GenTest, Normalize, and LINSEN. The simplest and widely adopted CamelCase technique is often sufficient to accomplish software evolution tasks (*e.g.*, (Dit *et al.*, 2011)). CamelCase relies on the use of naming conventions such as the CamelCase and underscore. In addition, CamelCase strategies do not use contextual information such as the case for Samurai which uses term frequencies as its local context (Enslen *et al.*, 2009). In a sense, Samurai can be thought as a clever CamelCase guided by global and local knowledge encoded into frequency tables. Indeed, Samurai local table is built by mining terms in the program under analysis while the global table is made by mining the set of terms in a large corpus of programs.

Recently, a machine-translation algorithm based on n-gram language models has been proposed to accurately split and expand identifiers (Lawrie et Binkley, 2011). The latter approach is called Normalize, it is a refinement of GenTest, an identifier splitting algorithm described in (Lawrie *et al.*, 2010). GenTest consists of two parts: the generation part generates all possible splittings and the test part evaluates a scoring function against each proposed splitting. The core part of Normalize is based on a machine translation technique, namely, the maximum coherence model (Gao *et al.*, 2002), which is based on co-occurrence data that captures local context information. The heart of normalization is a similarity metric computed from co-occurrence data, exploited to select the best candidate among several possible expansions.

LINSEN (Corazza *et al.*, 2012) is a novel technique that maps a given identifier to the set of corresponding dictionary words. The technique is based on a graph model and performs in linear time with respect to the size of the dictionary, taking advantage of an approximate string matching algorithm, the Baeza-Yates and Perleberg (Baeza-yates et Perleberg, 1992). LINSEN exploits a number of different dictionaries, referring to increasingly broader contexts, in order to achieve a disambiguation strategy based on the knowledge gathered from the most appropriate domain.

We share with the above-mentioned works the idea that source code vocabulary normalization is a challenging problem and that context is relevant for such a task.

3.4 Feature Location

Feature location is the activity of finding the source code elements (*i.e.*, methods or classes) that implement a specific feature (*e.g.*, "print page in a text editor" or "add bookmark in a web-browser") (Marcus *et al.*, 2004; Poshyvanyk *et al.*, 2007). In large software systems, there may be hundreds of classes and thousands of methods. Finding even one method that implements a feature can be extremely challenging and time consuming. Fortunately, for software engineers facing this situation, there are feature location techniques that automate, to a certain extent, the search for a feature's implementation. Existing feature location techniques rely on different tactics to find a feature's source code. IR-based approaches leverage identifiers and comments to locate source code that is textually similar to a query describing a feature (Marcus *et al.*, 2004).

Grant *et al.* (Grant *et al.*, 2008) used Independent Component Analysis for feature location, by separating the features (modeled as input signals) into independent components and estimating the relevance to each source code method.

Shepherd *et al.* (Shepherd *et al.*, 2007) proposed an approach to feature location that is based on the program model that captures action-oriented relations between identifiers in a program.

There are several feature location techniques that use more than one type of information (or underlying analysis). For example, SITIR (Liu *et al.*, 2007) and PROMESIR (Poshyvanyk *et al.*, 2007) both utilize textual and execution information. Execution information is gathered via dynamic analysis, which is commonly used in program comprehension (Cornelissen *et al.*, 2009) and involves executing a software system under specific conditions. For feature location, these conditions involve running a test case or scenario that invokes a feature in order to collect an execution trace. For example, if the feature of interest in a text editor is "printing", the test case or scenario would involve "printing a file". Invoking the desired feature during runtime generates a feature-specific execution trace.

Eisenbarth *et al.* (Eisenbarth *et al.*, 2003) proposed a technique that applies formal concept analysis to traces to generate a mapping between features and methods.

Cerberus (Eaddy *et al.*, 2008a) is another hybrid technique which combines static, dynamic and textual analysis. A comprehensive summary of feature location approaches can be found in (Revelle *et al.*, 2010).

A more recent work by Binkley *et al.* (Binkley *et al.*, 2012) investigated the used of vocabulary normalization on IR-based tools. The authors conducted an experiment where they applied the Normalize technique (Lawrie et Binkley, 2011), to an LSI-based feature locator. The replication of the study done by Marcus *et al.* (Marcus *et al.*, 2004) provides a

baseline for measuring the impact of normalization. Results of this replicated study show that normalization improves the ranks of relevant documents in the considered IR environment because it is able to recover key domain terms that were shrouded in non-aligned vocabulary. This improvement is most pronounced for shorter, more natural, queries.

We share with these works the idea that feature location is an important software engineering task and that identifier splitting/expansion is one of its essential ingredients.

3.5 Traceability

Traceability recovery has long been recognized as a major maintenance task in software engineering (Gotel et Finkelstein, 1993).

Several approaches (Antoniol *et al.*, 2000, 2001; Eaddy *et al.*, 2008a) have been suggested to recover traceability links between high-level documents, *e.g.*, requirements, and low-level documents, *e.g.*, source code.

Antoniol *et al.* (Antoniol *et al.*, 2001, 2002) proposed an approach, using class attributes as traceability anchors, to automatically recover traceability links between object-oriented design models and code.

Sherba and Anderson (Sherba et Anderson, 2003) proposed an approach, TraceM, to manage traceability links between requirements and architecture. TraceM is based on techniques from open hypermedia and information integration. Open hypermedia system enables the creation and viewing of relationships in heterogeneous systems. TraceM allows the creation, maintenance, and viewing of traceability relationships in tools that software professionals use on a daily basis.

Maider *et al.* (Mader *et al.*, 2009) encouraged a wider adoption of traceability and, thus, they refocused their attention on practical ways to apply traceability information models in practice. The authors highlighted the typical decisions involved in creating a basic traceability-information model, suggested a simple UML-based representation for its definition, and illustrated its central role in the context of a modeling tool.

Maletic *et al.* (Maletic et Collard, 2009) proposed, TQL, an XML-based traceability query language that supports queries across multiple artefacts and multiple traceability link types. TQL has primitives to allow complex queries construction and execution support.

Eddy et *et al.* (Eaddy *et al.*, 2008a) proposed a new technique, prune-dependency analysis that can be combined with existing techniques to dramatically improve the accuracy of concern location. The authors developed CERBERUS, a hybrid technique for concern location that combines information retrieval, execution tracing, and prune dependency analysis.

Andrea *et al.* (De Lucia *et al.*, 2010) proposed an approach to help developers maintain

source code identifiers and comments consistent with high-level artifacts. The approach uses textual similarity between source code and related high-level artifacts.

Zou *et al.* (Zou *et al.*, 2010) performed empirical studies to investigate Query Term Coverage, Phrasing, and Project Glossary term-based enhancement methods that are designed to enhance the performance of a probabilistic automated tracing tool. The authors suggested a procedure to automatically extract cryptic keywords and phrases from a set of traceable artifacts to improve the automated trace retrieval.

Ali al. (Ali *et al.*, 2013) proposed, Trustrace, an approach that is based on mining software repositories and combines mined results with IR techniques to improve the accuracy (*i.e.*, precision and recall) of requirements traceability links.

In this thesis, we do not suggest new traceability recovery approaches. However, we investigate the impact of source code vocabulary normalization on two traceability recovery techniques: one based on LSI while the second uses VSM.

CHAPTER 4

Context-Awareness for Source Code Vocabulary Normalization

Several research works such as Samurai (Enslen *et al.*, 2009), Normalize (Lawrie et Binkley, 2011), and LINSEN (Corazza *et al.*, 2012) have exploited contextual information in the splitting and expansion process to support program understanding and software maintenance. Despite the availability of identifier splitting/expansion tools that exploit context, there is very little empirical evidence on the extent to which context is relevant to lexicon normalization and, thus, ultimately to program understanding and software maintenance.

In this chapter, we describe the two user studies we performed to show the effect of context on source code vocabulary normalization.

4.1 Experiments' Definition and Planning

We conducted a family of two experiments with 63 participants, including graduate students and post-docs at École Polytechnique de Montréal. Each participant split and expanded a set of identifiers extracted from various C programs when working with different kinds of available contextual information.

Specifically, the two user studies aim at investigating:

1. The effect of contextual information. Contextual information is any information that developers can access when splitting and expanding identifiers, and, in general, information developers can have available during a program comprehension task. Normally, during such a task, a developer looks at neighboring source code, *i.e.*, the function where the identifier occurs, its file or even the entire project, by reading comments, other identifiers, etc. In the first experiment, we considered the internal context, *i.e.*, source code functions and files, and the internal plus external contexts, *i.e.*, source code files, plus the availability of an acronym dictionary, in the following referred to as "Acronym Finder". In the second experiment, we investigated two additional context levels, one consisting of all source code files from the application, and another consisting of such files plus the external context.

2. The accuracy in dealing with terms—composing identifiers—consisting of plain English words, abbreviations, and acronyms.

3. The effect of factors that might influence the participants' performances (Wohlin *et al.*, 2000): participants' background, programming expertise, domain knowledge, and English proficiency.

This section describes our experiment following the templates provided by Basili *et al.* (Basili *et al.*, 1994) and Wohlin *et al.* (Wohlin *et al.*, 2000).

4.1.1 Experiments' Definition

The *goal* of these two user studies is to investigate how developers split and expand source code identifiers, with the *purpose* of evaluating the impact of contextual information as well as of other factors, such as identifier characteristics and developers' background.

The *quality focus* is program understanding, which could be improved by adopting meaningful identifiers, or by designing tools that take into account contextual information to improve their accuracy.

The *perspective* is of researchers and practitioners interested in (i) investigating the extent to which contextual information helps developers properly understand and map identifiers to dictionary (or domain) words, and (ii) determining the developers' characteristics that are relevant for the identifier splitting and expansion task.

The *participants* involved in the study are Bachelor, Master, Ph.D. students, and post-docs. The *objects*, *i.e.*, identifiers to be split and expanded, and their related context (*e.g.*, source code files or functions in which they appear) have been extracted from C open source utilities.

Experiments' subjects and objects

We conducted two user studies—in the following referred to as *Exp I* and *Exp II*—which will vary the levels of context.

In both experiments, the participants are students and post-docs of the Computer and Software Engineering department of École Polytechnique de Montréal. Exp I had 42 participants: 28 Ph.D., eight Master, and Five bachelor students, plus one post-doctoral fellow. Exp II had 21 participants: 10 Ph.D., six Master, three Bachelor students, and two post-doctoral fellows. In total, we had 63 participants.

We collected information about all experiment participants, using a pre-experiment questionnaire in which we asked participants to self-evaluate themselves about their level of C programming knowledge, Linux knowledge, and English proficiency. All participants have at least a basic knowledge in the C programming language and have performed at least one maintenance task in previous years (*i.e.*, it was not the first time they had to deal with source

Table 4.1 Participants' characteristics and background.

Exp I (42 participants)		
Characteristic	Level	# of Participants
Program of Studies	Bachelor	5
	Master	9
	Ph.D.	28
	Post-doc	1
C Programming Experience	Basic	11
	Medium	23
	Expert	9
English Proficiency	Bad	8
	Good	8
	Very good	18
	Excellent	8 (7)
Linux Knowledge	Occasional	12
	Basic usage	13
	Knowledgeable but not expert	17
	Expert	0
Exp II (21 participants)		
Characteristic	Level	# of Participants
Program of Studies	Bachelor	3
	Master	6
	Ph.D.	10
	Post-doc	2
C Programming Experience	Basic	6
	Medium	5
	Expert	10
English Proficiency	Bad	1
	Good	9
	Very good	6
	Excellent	11 (6)
Linux Knowledge	Occasional	10
	Basic usage	6
	Knowledgeable but not expert	5
	Expert	0

code). In summary, the participants' population includes a variety of training levels (from Bachelor to post-doctoral) and, hence, it can be considered representative of young developers hired by companies in the Montréal area. Also, participants have different levels of English proficiency. The main characteristics of experiment participants and their background gained from the pre-experiment questionnaire are summarized in Table 4.1. In parenthesis, we report the total number of English native speakers, that are a subset of those who indicated an excellent English proficiency.

The participants split and expanded 50 identifiers extracted from 34 open-source C applications. Identifiers were selected as follows: we first extracted all identifiers, then we discarded English words since the purpose of our study is to study how developers expand identifiers with abbreviations and acronyms. Finally, after removing identifiers shorter than three characters, we randomly sampled identifiers from the obtained set. These applications

are some GNU Unix Utilities[1] (*e.g.*, *binutils*, *emacs*, *gcal*, *gcc*, *gcl*, *gmp*, *gnuradio*, *gnuspool*, *gprolog*, *gs*, *g77*, *lynx*, *sendmail*, or *sed*), and two operating system kernels (the Linux Kernel[2] 2.6.31.6 and FreeBSD[3]). The GNU utilities belong to various domains. For example, *gcal* is a calendar, *gcc* is a C compiler, *g77* is a Fortran to C translator, *lynx* is a textual browser, *sendmail* is a mail server, *sed* is a regular expression interpreter, and *binutils* are various command line utilities (*e.g.*, cat, wc, sort, etc.). The main characteristics of the applications from which we randomly sampled the 50 identifiers are summarized in Table B.1. We focused on C applications only because, as found in our previous work (Madani *et al.*, 2010), Java identifiers (almost) strictly adhere to the CamelCase convention and identifier construction rules as opposite to C and C++ where developers tend to use short identifiers.

It can be noticed that the number of identifiers sampled from each application is not perfectly uniform, *i.e.*, there are applications that one identifier and others contributed more than one. This is because our purpose was not to have a set of identifiers fully representative of these applications, but rather to represent different characteristics of C identifiers (separators and terms composing such identifiers). The random sample of identifiers chosen for this study is presented in Table B.1. There were not identifiers that were more useful than others because all of them fit with the purpose of our study, *i.e.*, they are/contain abbreviated words, acronyms, or concatenation of these types and English words.

4.1.2 Research Questions and Hypothesis Formulation

In the following, we present and motivate the research questions addressed in our work, and formulate the related null hypotheses.

1. **RQ1:** *To what extent does contextual information impact the splitting and expansion of source code identifiers?* This research question analyzes the developers' performances when splitting and expanding identifiers in absence and presence of contextual information. When developers split identifiers and expand abbreviations, it is possible that they do so relying on information (*e.g.*, comments or other identifiers) in the neighborhoods of the identifiers, or on other external sources of information (*e.g.*, an acronym dictionary). In this experimental investigation, we considered two types of contextual information (i) internal, *i.e.*, information a developer can get from source code or comments of the system itself, and (ii) external, *i.e.*, information a developer can retrieve from other sources, such as dictionaries, acronym lists, etc. The contexts considered in the two experiments are summarized in Table 4.2. In Exp I, we considered four different

[1] `http://www.gnu.org`
[2] `http://www.linux.org`
[3] `http://www.freebsd.org`

levels of context, *i.e.*, (i) the absence of contextual information (identifier in isolation with no context), which constitutes for us a control group; (ii) a context concerning the function where the identifier is located; (iii) a context concerning the file where the identifier is located; and (iv), file-level context augmented with the Acronym Finder to help the participant dealing with acronyms. It is important to note that we only provided the Acronym Finder with the file-level context to limit the number of possible treatments, and also because we were interested in observing the effect of such an additional context level beyond the widest (file-level) context of Exp I. For Exp II, in addition to (i) no context, (ii) file-level context, and (iii) file plus Acronym Finder-level context, we also considered (iv) application-level context and (v) application-level plus Acronym Finder.

In Exp II, we did not consider the function-level context because: i) we already concluded from the results of Exp I which are reported in Section 4.2 that the function-level context does not improve the participants performances when splitting and expanding source code identifiers in comparison with the other levels of context (*i.e.*, file and file plus external information levels); and ii) in Exp II we had only 21 participants instead of 42 for Exp I, and therefore we had to limit the number of possible treatments.

The null hypothesis being tested to address this research question is:

H_{01}: *There is no significant effect of the context on the participants' performances when splitting and expanding source code identifiers.*

Table 4.2 Context levels provided during Exp I and Exp II.

Context Levels	**Exp I**	**Exp II**
No context (control group)	√	√
Function	√	
File	√	√
File plus Acronym Finder	√	√
Application		√
Application plus Acronym Finder		√

2. **RQ2:** *To what extent do the characteristics of terms composing identifiers affect splitting and expansion performances?*

   This research question investigates whether particular characteristics of terms composing identifiers may favor or hinder the developer's capability of splitting and expanding such identifiers. Our conjecture is that, in general, identifiers composed of English

words would be easier to split, although this is not 100% guaranteed because one can be tempted to expand an English word, *e.g.*, add a plural, conjugate a verb. Also, sometimes multiple splits are possible, *e.g.*, *callableinterface* can be split into (*callable*, *interface*) or (*call*, *able*, *interface*). However, generally acronyms and abbreviations are the major cause of imprecision. In summary, this research question analyzes what percentage of English words, acronyms, and abbreviations composing identifiers are correctly split/expanded by the participants. Characteristics of the studied identifiers in terms of style (*e.g.*, separators such as underscore and–or CamelCase) and kind of terms they contain are summarized in Table B.2 (cf. Appendix B). In this work, we focused on the relevant contexts for the identifier splitting/expansion rather than the impact of identifier styles on such a task. However, recent research works have investigated the use of different identifier styles for program comprehension (Binkley *et al.*, 2013; Sharif et Maletic, 2010).

In summary, for what concerns **RQ2**, we test the following null hypothesis:

H_{02}: *there's no significant difference in the accuracy of splitting/expanding full English words, acronyms, and abbreviations.*

3. **RQ3:** *To what extent do participants' background and characteristics impact the performance of identifier splitting and expansion?* This research question investigates how variables—related to characteristics of the developers performing the tasks—impact the splitting and expansion performances and the extent to which such factors interact with the use of contextual information. In the following, we will refer such factors as "population variables". Specifically, the factors that we considered are: (i) level of experience, (ii) programming language (C) knowledge, (iii) domain knowledge (which concerns the knowledge of Unix/Linux utilities for the programs we considered in our experiments), and (iv) English proficiency.

   Given $factor_i$, one of these factors, we tested two null hypotheses:

   - H_{03a}: *there is no significant effect of* $factor_i$ *on identifier splitting and expansion accuracy.*
   - H_{03b}: *there is no significant interaction between* $factor_i$ *and the studied levels of context on identifier splitting and expansion accuracy.*

4.2 Variable Selection and Experiment Design

4.2.1 Variable Selection

Dependent Variables

Our first dependent variable is the participants' performances when performing identifier splitting and expansion. Such a performance is measured using precision and recall defined in Chapter 2, with respect to a manually built oracle. F-measure is often used in various studies, *e.g.*, related to program comprehension (Ricca *et al.*, 2010) and, in general, in information retrieval (Baeza-Yates et Ribeiro-Neto, 1999) to aggregate precision and recall (cf. Chapter 2). We introduced F-measure to provide an aggregate measure of precision and recall, where precision and recall show consistent trends with respect to our independent variable. However, wherever appropriate, we also reported results of precision and recall separately, which for all cases are available in Appendix B.

For **RQ2**, we considered the accuracy of splitting/expanding identifiers terms of a particular kind (English words, acronyms, and abbreviations):

$$Accuracy_j = \frac{CETERMS_j}{TERMS_j}$$

where $j = 1, 2, 3$; $TERMS_1$, $TERMS_2$, and $TERMS_3$ are the sets of all plain English words, acronyms, and abbreviations contained in the identifiers considered in our study respectively; $CETERMS_1$, $CETERMS_2$, and $CETERMS_3$ are the sets of correctly split/expanded English words, acronyms, and abbreviations. The identifier split/expansion correctness was determined in a binary way and the accuracy of splitting/expanding identifiers terms of a particular kind was computed over all identifiers considered in our study.

To measure our dependent variables, we manually built our oracle by associating each identifier with a list of terms obtained after splitting it and expanding them. We only created one oracle for both the splitting and expansion because we considered them as one task; we justified this assumption by the fact that identifier expansion involves identifier splitting. In fact, one has first to identify terms composing an identifier and then expand them to their corresponding domain concepts. For example, the oracle entry for *drawRec* would be *draw rectangle*, obtained by splitting the identifier after the seventh character and after expanding the abbreviation *Rect* into *rectangle*.

Independent Variables

The independent variables of our study are all the factors that we considered when testing the null hypotheses formulated in Section 4.1.2 and summarized in Table 4.3. Specifically, for

RQ1 we considered the different levels of context; for **RQ2** we considered the different kinds of terms composing identifiers; for **RQ3** we considerded population variables, *i.e.*, experience level, C knowledge, Linux knowledge, and English proficiency.

4.2.2 Experiment Procedure and Design

Exp I

In Exp I, each participant split and expanded 40 different identifiers from the original set of the 50 identifiers. On the one hand, the reason behind asking each subject to split/expand only 40 identifiers was to limit the fatigue effect. On the other hand, we made sure that each identifier were assigned to (roughly) the same number of subjects, with the same proportion of different contexts as shown in the experimental design.

Different participants operated having different levels of context available:

- without context *i.e.*, just an identifier in an empty page as shown in Fig. 4.1(a);
- within a *source code function*, *i.e.*, participants could browse the source code of the function containing the identifier;
- within a *source code file*, *i.e.*, participants could browse the source code of the file containing the identifier as shown in Fig. 4.1(b);
- with a *source code file plus external context*, *i.e.*, participants have access to an identifier source code file plus a list of widely used (cross-domain) acronyms and abbreviations, named Acronym Finder[4].

Participants had access to computers with two screens. On one screen, they had access to the Web application where all questions and tasks appear. When dealing with function, file, and file plus Acronym Finder context levels, we provided to participants pretty-printed source code in HTML format through a Web browser. On the second screen, they had access to the response form. We informed the participants before the experiment that the time available for the tasks was 120 minutes in total, and that they were free to leave at any time without incurring any penalty. Collected information was anonymous. We validated the response forms to make sure that participants correctly followed the experiment procedure. Participants were aware of the general goal of the study—*i.e.*, to investigate how developers split and expand identifiers—but did not know the exact hypotheses being tested.

We randomly gave to each group of participants different sets of 40 identifiers to make sure that, in our design, all the 50 identifiers of our samples were split/expanded by multiple

[4]http://www.acronymfinder.com

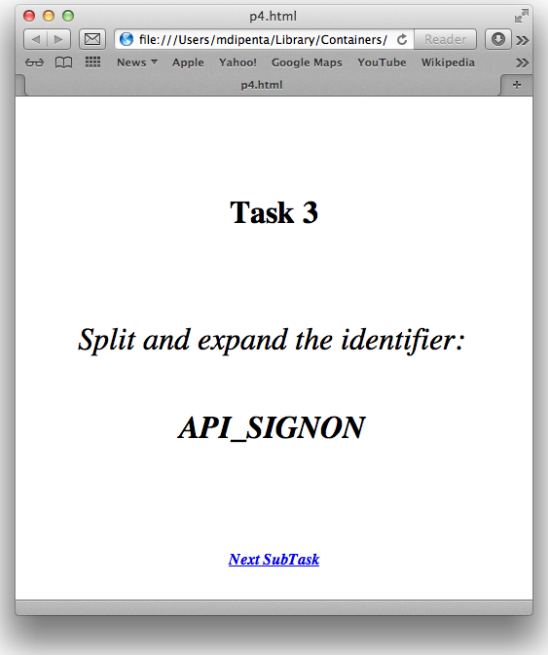
Table 4.3 Null hypotheses and independent variables.

RQs	Hyps.	Descriptions	Independent Variables	Independent Variable Levels
RQ1	H_{01}	Effect of context.	Context levels.	Exp I: No context, function, file, file plus Acronym Finder. Exp II: No context, file, file plus Acronym Finder, application, and application plus Acronym Finder.
RQ2	H_{02}	Effect of the kinds of terms composing identifiers.	Kind of terms.	Plain English words, acronyms, abbreviations.
RQ3	H_{03a}	Effect of participant characteristics/background (population variables).	Experience, C knowledge, domain (Linux) knowledge, English proficiency.	Experience levels: Bachelor, Master, Ph.D., post-doc. C knowledge levels: basic, medium, expert. Domain (Linux) knowledge levels: occasional, basic, knowledgeable, expert. English proficiency levels: bad, good, very good, excellent.
	H_{03b}	Interaction of population variables with use of context.		

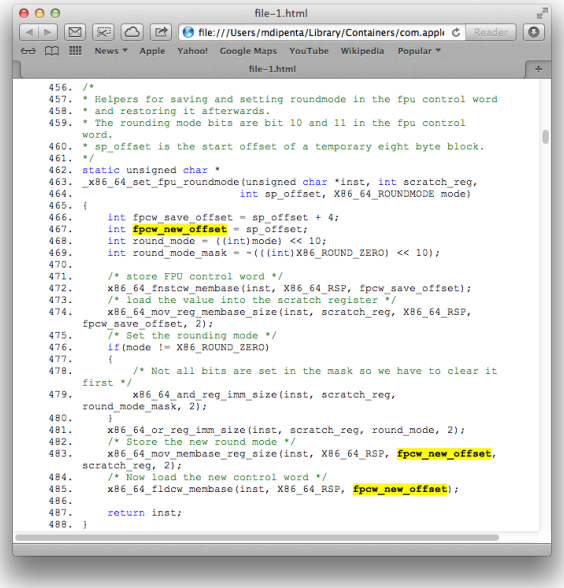
participants. Also, the task was organized in a way giving the participants different ordering of treatments to avoid fatigue and learning effects *i.e.*, we avoid giving participants identifiers to split/expand without context, then with function level and so on. To achieve such a goal, we created five sets of ten identifiers ($ids_1 \ldots ids_5$) and assigned to each participant four of these five groups. To allow each subject splitting/expanding 40 identifiers, and to make sure that each identifier were split by roughly the same number of people, we needed to group participants in a way that each group received four sets of identifiers. That is, we needed five groups of subjects *i.e.*, the combinations of five items by groups of four:

$$\binom{5}{4} = \frac{5!}{4! \cdot (5-4)!} = 5$$

Table 4.4 shows a summary of our experimental design. Given Group$_i$ the *i-th* group of participants, ids_j a set j of ten identifiers to be split and expanded by participants with $j \in \{1, 2, 3, 4, 5\}$, and cx_k effect of context k on the participants' performances with $k \in \{1, 2, 3, 4\}$. In our case, the different levels of context are cx_1: no contextual information; cx_2: contextual information related to the function where the identifier appears; cx_3: contextual information related to the file where the identifier appears; and cx_4: as cx_3, plus Acronym Finder. As described in Table 4.4, each group of participants dealt with the four context levels, we gave each group a set of ten identifiers per context, *i.e.*, a total of 40 different identifiers to split and expand. We randomized the ordering of treatments to mitigate such a threat; however, it is impossible to have all possible orderings (which would be 40!). In



(a) no context



(b) file

Figure 4.1 Example of treatments received by the participants: no-context and file-level context.

principle, we could have given to each participant all identifiers of a given set with a given context, then another set with a different context, and so on. For instance, we could have

Table 4.4 Exp I: Experimental design.

Sequence	**Group**$_1$	**Group**$_2$	**Group**$_3$	**Group**$_4$	**Group**$_5$
1	rnd(ids1)-cx_1	rnd(ids2)-cx_1	rnd(ids1)-cx_1	rnd(ids1)-cx_1	rnd(ids1)-cx_1
2	rnd(ids2)-cx_2	rnd(ids3)-cx_2	rnd(ids3)-cx_2	rnd(ids2)-cx_2	rnd(ids2)-cx_2
3	rnd(ids3)-cx_3	rnd(ids4)-cx_3	rnd(ids4)-cx_3	rnd(ids4)-cx_3	rnd(ids3)-cx_3
4	rnd(ids4)-cx_4	rnd(ids5)-cx_4	rnd(ids5)-cx_4	rnd(ids5)-cx_4	rnd(ids5)-cx_4
5	rnd(ids1)-cx_1	rnd(ids2)-cx_1	rnd(ids1)-cx_1	rnd(ids1)-cx_1	rnd(ids1)-cx_1
6	rnd(ids2)-cx_2	rnd(ids3)-cx_2	rnd(ids3)-cx_2	rnd(ids2)-cx_2	rnd(ids2)-cx_2
7	rnd(ids3)-cx_3	rnd(ids4)-cx_3	rnd(ids4)-cx_3	rnd(ids4)-cx_3	rnd(ids3)-cx_3
8	rnd(ids4)-cx_4	rnd(ids5)-cx_4	rnd(ids5)-cx_4	rnd(ids5)-cx_4	rnd(ids5)-cx_4
...	...	...	...	...	...
37	rnd(ids1)-cx_1	rnd(ids2)-cx_1	rnd(ids1)-cx_1	rnd(ids1)-cx_1	rnd(ids1)-cx_1
38	rnd(ids2)-cx_2	rnd(ids3)-cx_2	rnd(ids3)-cx_2	rnd(ids2)-cx_2	rnd(ids2)-cx_2
39	rnd(ids3)-cx_3	rnd(ids4)-cx_3	rnd(ids4)-cx_3	rnd(ids4)-cx_3	rnd(ids3)-cx_3
40	rnd(ids4)-cx_4	rnd(ids5)-cx_4	rnd(ids5)-cx_4	rnd(ids5)-cx_4	rnd(ids5)-cx_4

given to participants of Group$_1$ all identifiers of $ids1$ with context cx_1, then identifiers of $ids2$ with cx_2, and so on. However, this could have caused learning and boredom effects, because participants of Group$_1$ would have worked with context cx_1 first, then with cx_2, etc. To avoid such effect, we used a slightly different design. Specifically, for Group$_1$ an identifier was randomly extracted from $ids1$—as denoted in Table 4.4 by the function *rnd()*—and provided to participants with context cx_1, then one identifier from $ids2$ with cx_2, one from $ids3$ with cx_3, and one identifier from $ids4$ with cx_4. After that, the sequence restarted using again (without resampling) an identifier from $ids1$, and so on. We believe that randomizing the ordering of treatments mitigates the threat related to the order of the application since it is impossible to have all possible orderings for each group of subjects (which would be 40! in Exp I). In our studies, randomization means that not all identifiers were seen in the same order. We followed the same procedure for the other groups, however not in the same order so that the $idsj$-cx_j association is not always the same instance.

To create the five groups, we considered blocks of participants with a basic, medium, and expert knowledge of C. Participants were also blocked according to their level of English, and their background. Then, we created five groups by randomly assigning participants from blocks in nearly identical proportions. Each of the five groups is of approximately the same size (either eight or nine participants). Then, when assigning identifiers to groups, since each group worked on a subset of 40 out of the 50 identifiers, we made sure that each group worked on a roughly equal proportions of plain English words, abbreviations, and acronyms. Table 4.5 shows the distribution of abbreviations, acronyms and plain English words per group of participants. The total number of plain English words, abbreviations and acronyms in all the 50 identifiers is reported on the legend of Table 4.5. The latter number is different for each category because in this work, we did not explicitly design our studies

to analyze whether the splitting/expansion difficulty is related to the kinds of terms that compose identifiers. We studied the accuracy of splitting/expanding different kinds of terms (*i.e.*, abbreviations, acronyms, English words) and how this accuracy varies with different levels of context. In addition, our design reflects the "real" proportions of each category of terms (abbreviations, acronyms, and English words) that belong to realistic identifiers that we used in our experiments. Hence, our design tries to mimic "reality", when developers must understand real identifiers.

In summary, it is important to point out that the experimental design fully reflects the need for controlling the *Context* factor investigated in **RQ1**. The effect of the independent variable of **RQ2** (terms contained in identifiers) is controlled by means of blocking, *i.e.*, assigning to different groups roughly equal proportions of identifiers containing full English words, abbreviations and acronyms as indicated in Table 4.5, and by means of randomization as explained above. Clearly, the randomization is only partial as it is not possible to have all possible combinations of identifiers and/or of applications to which the identifiers belong. Finally, population variables related to **RQ3** are dealt by means of blocking as explained above.

Table 4.5 Distribution of kinds of identifier terms for Exp I, out of a total of 86 abbreviations, 19 acronyms, and 48 plain English words.

Exp I			
Groups of Participants	Abbreviations	Acronyms	Plain English
Group 1	60	14	42
Group 2	69	15	40
Group 3	67	16	34
Group 4	65	16	37
Group 5	70	14	40

The rationale for making participants deal with identifiers belonging to different domain applications was to (1) maximize the number of data points (observations) so as to increase statistical power, (2) avoid bias from the differences in programs' complexity, and (3) make general conclusions from the obtained results. Detailed information about the specific identifiers and contexts provided to participants is available in our replication package. The number of soft-words and hard-words contained in the identifiers provided to each group of participants is indicated in Table 4.6.

Table 4.6 Distribution of soft-words and hard-words for Exp I, out of a total of 119 soft-words and 79 hard-words (provided in Exp II).

Exp I		
Groups of Participants	Soft-words	Hard-words
Group 1	91	71
Group 2	101	61
Group 3	94	73
Group 4	101	67
Group 5	89	74

Exp II

Exp II was a replication of Exp I, having two main purposes:

1. Exploring the usefulness of additional level of context, namely the availability of the whole source code of an application, with and without the additional support of the Acronym Finder.

2. Corroborating the conclusions of Exp I, by employing more experienced participants, and in particular participants having a better level of English, which plays an important role in splitting/expansion as we observed in Exp I.

3. Allowing each participant to work—in different ordering and with different context levels—on all the 50 identifiers, whereas in Exp I each participant received only a subset of 40 identifiers. This decision helped avoid any blocking issue on **RQ2**, since all participants had to deal with all the 50 identifiers *i.e.*, 48 plain English words, 86 abbreviations, and 19 acronyms.

The experimental design of Exp II is similar to that of Exp I, except that (i) there are five context levels instead of four, and (ii) each participant received all the 50 identifiers instead of 40 of them. Similarly to Exp I, we had five groups of participants, we considered blocks of participants with a basic, medium, and expert knowledge of C. Participants were also blocked according to their level of English and their background. Then, we created five groups by randomly assigning participants from blocks in nearly identical proportions. Each of the five groups is of approximately the same size (either four or five participants). Then, each group worked on the 50 identifiers, *i.e.*, on the total number of plain English words, abbreviations, and acronyms, *i.e.*, 48, 86, and 19 respectively.

The way identifiers were presented to participants is the same as for Exp I. However, for the application-level context, it was not convenient to show the whole application source code in a Web page. Instead, when participants had to work with application or application plus Acronym Finder context levels, they had access to the identifier applications in the Eclipse-JDT IDE, while the Web application only indicated the identifier to be split/expanded and the name of the application where the identifier appears. Since we increased the number of identifiers in Exp II, we also increased the time allocated to the experimental tasks. In fact, we informed the participants before the experiment that the time available for the tasks was 180 minutes in total, and that they were free to leave at any time without incurring any penalty. We also explained to them that the collected data was anonymous and made them aware of the general goal of the study without revealing the exact hypotheses being tested.

Post-experiment Questionnaire

In both Exp I and Exp II, after having completed their task, we gave to participants a post-experiment questionnaire, aimed at gaining insights about the collected data. We asked each participant whether the context was helpful for him/her. A summary of the post-experiment questionnaire is shown in Table 4.7; answers were collected on a five-point Likert scale, plus a free text form where the participants were asked to provide further comments, if any.

Table 4.7 Post-experiment survey questionnaire.

Summary of post-experiment questionnaire.	
ID	Questions
Question 1	How did you find the information provided in the experiment procedure?
Question 2	Was the context helpful when splitting and expanding identifiers? If yes, what was the most helpful type of context (Function, File, File plus Acronym Finder, Application, or Application plus Acronym Finder) for you among the investigated ones?
Question 3	To what extent was your knowledge in Linux helpful when splitting and expanding source code identifiers?
Question 4	Were the comments provided with the source code helpful for you when splitting and expanding the identifiers in question?

4.3 Analysis Method

In the following we describe the statistical procedures used to analyze our results. All of them have been applied using the *R* statistical environment (Team, 2012).

RQ1 concerns the comparison of the precision, recall, and F-measure of identifier splitting/expansion provided by the participants for the studied levels of context. Other than

showing boxplots and descriptive statistics, we tested the null hypothesis H_{01} using the Wilcoxon paired test. We used the latter test to perform a pairwise comparison of the results obtained for each identifier (on which the test is paired) with the different levels of context. Since we applied the Wilcoxon test multiple times, we had to adjust p-values. We used the Holm's correction procedure (Holm, 1979). In addition to the statistical comparison, we computed the effect-size of the difference using Cliff's delta (d) non-parametric effect size measure (Grissom et Kim, 2005). Wilcoxon, Cliff's delta (d), and Holm correction are defined in Chapter 2.

RQ2 concerns the accuracy in splitting/expanding terms composing identifiers and consisting of plain English words, acronyms and abbreviations. We performed the analysis of **RQ2** by pairwise comparing the proportions of correctly split/expanded identifiers belonging to different categories, and using different contexts. Such a comparison is done using the Fisher's exact test (Sheskin, 2007). In addition, we used the Odds Ratio (OR) (Sheskin, 2007) as an effect size measure. The definitions of Fisher's exact test and OR are provided in Chapter 2. For contingency matrices, like our case, the OR is defined as: $OR = (Wrong_1/Corr_1) \,/\, (Wrong_2/Corr_2)$, where $Wrong_1$ and $Corr_1$ are the number of wrongly and correctly split/expanded terms for the first context, respectively, while $Wrong_2$ and $Corr_2$ the number of wrongly and correctly split terms for the second context.

RQ3 analyzes the interaction between the effect of context levels on participants' performances when splitting/expanding identifiers and a set of investigated population variables (experience, Linux expertise, C knowledge, and English proficiency). The analysis of population variables is performed using permutation tests (Baker, 1995) (cf. Chapter 2).

We used an implementation available in the *lmPerm* R package. We have set the number of iterations of the permutation test procedure to 500,000. Since the permutation test samples permutations of combination of factor levels, multiple runs of the test may produce different results. We made sure to choose a high number of iterations such that results did not vary over multiple executions of the procedure.

Wherever the permutation test indicated the presence of significant differences, we identified which pair(s) of factor levels exhibit the difference by using the Tukey's HSD (Honest Significant Differences) test (Sheskin, 2007) (cf. Chapter 2).

4.4 Experiments' Results

In this section, we report the quantitative results of our experiments, with the aim of addressing the research questions formulated in Section 4.1.2.

4.4.1 RQ1: Context Relevance

In Fig. 4.2, we show, for both experiments, the boxplots of F-measure computed for the studied context levels. Boxplots related to precision and recall can be found in Appendix B. Table 4.8 reports descriptive statistics, of precision, recall and F-measure *i.e.*, 1^{st} quartile, median, 3^{rd} quartile, mean and standard deviation.

For Exp I, the boxplots and the table show that the participants achieve the best performances in terms of precision, recall and F-measure when using file plus Acronym Finder and file contexts. The recall (and consequently the F-measure) results are slightly higher for the File plus Acronym Finder context, while the precision is similar for the two contexts. Function-level context yields lower performances (precision, recall, F-measure) than the other context levels, and performances are even lower when no contextual information is provided. These results were expected. In fact, when no context was available, participants were guessing possible split/expansion of identifiers and, thus, the chances of providing correct answers were low, which reflects the low precision, recall, and consequently the F-measure. The function-level context was, in general, not sufficient to know the exact semantics of the code and its identifiers especially when the function is not sufficiently commented. Providing participants with a wider context such as the file (with or without) Acronym Finder is helpful to understand the program. The availability of the Acronym Finder favors the increase of recall because, in general, it helps participants to correctly split/expand some acronyms by selecting the correct expansion among all the possible ones available in the Acronym Finder.

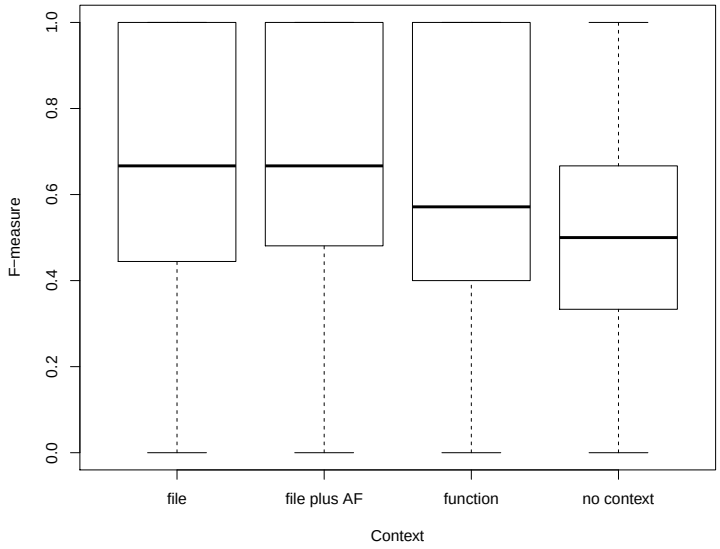
Data also shows that, as the available context increases (*i.e.*, from no context to function-level context, file, and file plus Acronym Finder), the performance variation (in terms of interquartile range and of variance) also increases, especially for what concerns the precision.

For Exp II, and consistently with what we found in Exp I, results indicate that the file-level context help obtain better performances than no context and that there is a slight improvement when the Acronym Finder is added. If using a larger—*i.e.*, application-level—context, performances do not further improve. These results were expected since participants, to avoid being overwhelmed by a large context such as the application-level, focused, in general, on source code files where identifiers appear.

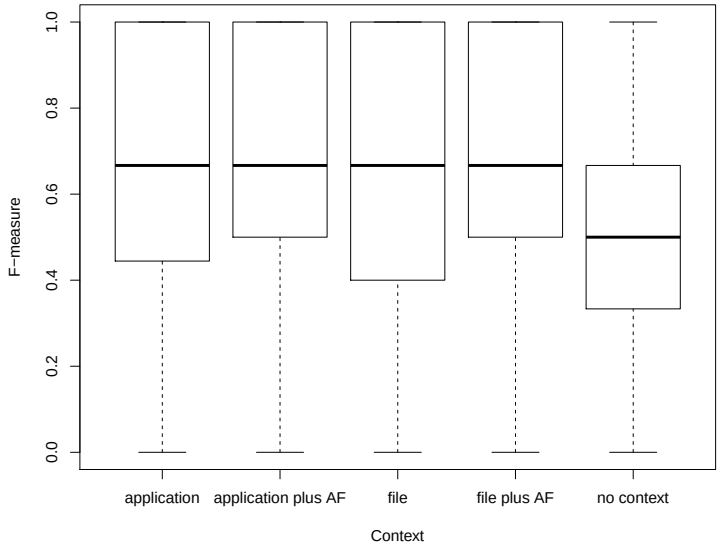
Tables 4.9 and 4.10 report results of the statistical comparison among the different levels of context for Exp I and Exp II respectively. Specifically, it shows the difference in participants' performances when using a context level *Context 1* versus another one *Context 2* (adjusted p-values[5] and Cliff's d effect size are positive when the effect is in favor of *Context 1*).

For Exp I, as indicated by Table 4.9, there is a significant difference in participants' performances, in terms of precision, between file plus Acronym Finder and no context with a

[5]Significant p-values are highlighted in bold face here and in all other tables.



(a) Exp I



(b) Exp II

Figure 4.2 Boxplots of F-measure for the different context levels (AF= Acronym Finder).

medium effect size; file and no context with a *small* effect size; function and no context with a *small* effect size. These differences are significant because participants perform significantly better when they are provided, in general, with some context. However, their performance decreases when no context, *i.e.*, no information about the semantics of the identifiers is

Table 4.8 Precision, recall, and F-measure of identifier splitting and expansion with different contexts.

Exp I						
Metrics	Contexts	1q	Median	3q	Mean	σ
Precision	no context	0.33	0.50	0.67	0.50	0.27
	function	0.50	0.67	1.00	0.63	0.29
	file	0.50	0.67	1.00	0.65	0.30
	file plus Acronym Finder	0.50	0.67	1.00	0.69	0.30
Recall	no context	0.33	0.50	0.67	0.47	0.27
	function	0.33	0.50	1.00	0.59	0.30
	file	0.40	0.67	1.00	0.64	0.31
	file plus Acronym Finder	0.48	0.67	1.00	0.68	0.31
F-measure	no context	0.33	0.50	0.67	0.48	0.27
	function	0.40	0.57	1.00	0.60	0.20
	file	0.44	0.67	1.00	0.64	0.30
	file plus Acronym Finder	0.49	0.67	1.00	0.68	0.30

Exp II						
Metrics	Contexts	1q	Median	3q	Mean	σ
Precision	no context	0.33	0.50	0.67	0.51	0.28
	file	0.50	0.67	1.00	0.68	0.30
	file plus Acronym Finder	0.50	0.67	1.00	0.69	0.29
	application	0.50	0.67	1.00	0.68	0.30
	application plus Acronym Finder	0.50	0.67	1.00	0.69	0.29
Recall	no context	0.33	0.50	0.67	0.48	0.28
	file	0.33	0.67	1.00	0.65	0.32
	file plus Acronym Finder	0.50	0.67	1.00	0.69	0.30
	application	0.40	0.67	1.00	0.66	0.32
	application plus Acronym Finder	0.50	0.67	1.00	0.69	0.30
F-measure	no context	0.33	0.50	0.67	0.49	0.27
	file	0.40	0.67	1.00	0.66	0.31
	file plus Acronym Finder	0.50	0.67	1.00	0.70	0.29
	application	0.44	0.67	1.00	0.67	0.31
	application plus Acronym Finder	0.50	0.67	1.00	0.68	0.30

provided. There is also a significant difference between file plus Acronym Finder and function with a *small* effect size. The latter result can be explained by the fact that the source code file contains more information about the application than the function. Also, adding the Acronym Finder to the file makes the context even more useful than the function level. In all other cases, there is no significant difference. Similar results were obtained for recall. In fact, there is a significant difference in participants' performances between file plus Acronym Finder and no context with a *medium* effect size; file and no context with a *small* effect size; function and no context with a *small* effect size; file plus Acronym Finder and function with a *small* effect size. In addition, there is a difference, in terms of recall, between file and function with a *small* effect size. This difference is explained by the fact that a limited context such as the source code function does not provide helpful information to participants when splitting/expanding identifiers. In all other cases, there is no significant difference. Finally, for the F-measure, differences are significant, as expected, between file and no context with a *small* effect size, between file plus Acronym Finder and no context with a *medium* effect

size, between function and no context with a *small* effect size, and between file plus Acronym Finder and function with a *small* effect size. The latter result is justified by the relevance of information provided by the source code file context and also by the help of the Acronym Finder, compared to the information provided by the function-level context.

Table 4.9 Exp I: precision, recall, and F-measure for different context levels: results of Wilcoxon paired test and Cliff's delta.

Precision			
Context 1	Context 2	Cliff's d	Adj p
file plus Acronym Finder	file	0.0747	0.120
file	function	0.0571	0.140
file	no context	0.2947	**<0.001**
file plus Acronym Finder	function	0.1296	**<0.001**
file plus Acronym Finder	no context	0.3624	**<0.001**
function	no context	0.2446	**<0.001**
Recall			
Context 1	Context 2	Cliff's d	Adj p
file plus Acronym Finder	file	0.0699	0.080
file	function	0.1058	**0.01**
file	no context	0.3293	**<0.001**
file plus Acronym Finder	function	0.1668	**<0.001**
file plus Acronym Finder	no context	0.3871	**<0.001**
function	no context	0.2215	**<0.001**
F-measure			
Context 1	Context 2	Cliff's d	Adj p
file plus Acronym Finder	file	0.0731	0.060
file	function	0.0855	0.060
file	no context	0.3214	**<0.001**
file plus Acronym Finder	function	0.1530	**<0.001**
file plus Acronym Finder	no context	0.3841	**<0.001**
function	no context	0.2427	**<0.001**

Regarding Exp II, results reported in Table 4.10 indicate that all context levels (file and application, with and without Acronym Finder) exhibit, in terms of precision, a significantly higher precision than no context, with a *medium effect size*. The latter result is justified by the fact that contextual information provides hints and clues to the participants to understand the meaning of identifiers and, hence, correctly split/expand them. However, there is no significant difference between any pair of these contexts. That is, beyond the file-level context, the precision does not significantly increase, which means that providing participants with a large context does not help attain better performances since, in general, participants will browse the source code files where the identifiers appear when facing too much information *i.e.*, the application-level context. Consistent results—in terms of the statistical significance and effect size—were obtained for recall and F-measure too between application plus Acronym Finder and no context, as well as between file plus Acronym Finder and no context. The latter results were expected since the level of context increased and also the availability of the Acronym Finder helps improve the participants performance, in terms of recall, and,

thus, F-measure by providing additional information and indications about the semantics of identifiers. Results of recall and F-measure are, however, as expected, significant with a *small effect size* between the application and no context levels, and between file and no context levels because, in general, when facing identifiers made up of acronyms without knowing or having clues about these acronyms, participants show low performances in terms of recall, as they cannot figure out the exact expansion of the acronym even if they are able to split/expand parts of it.

Table 4.10 Exp II: precision, recall, and F-measure for different context levels: results of Wilcoxon paired test and Cliff's delta.

Precision			
Context 1	Context 2	Cliff's d	Adj p
application plus Acronym Finder	application	0.0013	1.000
application plus Acronym Finder	file plus Acronym Finder	0.0495	1.000
application plus Acronym Finder	file	0.0077	1.000
application plus Acronym Finder	no context	0.3345	**<0.001**
application	file plus Acronym Finder	0.0507	1.000
application	file	0.0034	1.00
application	no context	0.3304	**<0.001**
file plus Acronym Finder	file	0.0540	1.000
file plus Acronym Finder	no context	0.3883	**<0.001**
file	no context	0.3834	**<0.001**
Recall			
Context 1	Context 2	Cliff's d	Adj p
application plus Acronym Finder	application	0.0318	1.000
application plus Acronym Finder	file plus Acronym Finder	0.0282	1.000
application plus Acronym Finder	file	0.0453	1.000
application plus Acronym Finder	no context	0.3519	**<0.001**
application	file plus Acronym Finder	0.0604	1.000
application	file	0.0145	1.0000
application	no context	0.3120	**<0.001**
file plus Acronym Finder	file	0.0741	1.000
file plus Acronym Finder	no context	0.3836	**<0.001**
file	no context	0.2916	**<0.001**
F-measure			
application plus Acronym Finder	application	0.0184	1.000
application plus Acronym Finder	file plus Acronym Finder	0.0405	1.000
application plus Acronym Finder	file	0.0277	1.000
application plus Acronym Finder	no context	0.3471	**<0.001**
application	file plus Acronym Finder	0.0548	1.000
application	file	0.0093	1.0000
application	no context	0.3234	**<0.001**
file plus Acronym Finder	file	0.0647	1.000
file plus Acronym Finder	no context	0.3882	**<0.001**
file	no context	0.3095	**<0.001**

In summary, we can conclude that *contextual information significantly increases the participants' performances when splitting and expanding identifiers*, in terms of precision, recall, and F-measure. Different levels of context do not exhibit significant differences, although the file plus Acronym Finder-level context exhibits (in *Exp I*) better performances than the function-level context. An application-level context does not contribute to improve the per-

formance, if compared with a narrower context *i.e.*, file-level.

4.4.2 RQ2: Effect of Kinds of Terms Composing Identifiers

In this section, we study the accuracy of splitting/expanding in terms of different kinds of composing identifiers, and how this accuracy varies with different levels of context. Table 4.11 reports the number of abbreviations, acronyms, and plain English terms that have been correctly expanded (# Matched), the number of those incorrectly expanded (# Unmatched), and the expansion accuracy. Table 4.12 reports the results of the pairwise comparison of accuracies for different kinds of terms; it shows Fisher's exact test adjusted p-values and OR.

For Exp I, as shown by Table 4.12, and for all levels of context, plain English words were significantly easier to expand than acronyms and abbreviations, with an OR between 2 and 2.6. The latter result can be perceived as obvious, because plain English words *per se* do not need to be expanded. However, as discussed in Section 4.1.2, two problems can still arise (i) identifiers composed of two or more English words can lead towards multiple possible splittings, and (ii) people might still "expand" an English word by conjugating verbs, adding/removing plurals, etc. Indeed, for these reasons, as Table 4.11 shows, the accuracy for plain English words is not 100%, but it ranges between 81% for the no context level and 87% for the file- and function-level contexts in Exp I, and between 83% for the no context level and 92% for the file-level context in Exp II.

Results of Exp I do not indicate any significant difference between proportions of correctly expanded abbreviations and acronyms. Indeed, the accuracy is very similar for all context levels, and the OR$\sim$ 1, *i.e.*, abbreviations and acronyms have equal chances to be correctly expanded. As Table 4.11 shows, their accuracy ranges between 63% with no context and 78% with the file plus Acronym Finder context.

Exp II shows slightly different results than Exp I. First, as it can be noticed in Table 4.11, the percentage of plain English terms correctly split/expanded is higher than in Exp I, and it ranges between 83% with no context and 92% with the file-level context. Abbreviations have accuracy in line with—or slightly higher than—Exp I, *i.e.*, ranging between 68% (with no context) and 83% (with the file plus Acronym Finder level). The latter results can be explained by the fact that participants of Exp II have a high proficiency of English that helped them recognize plain English words and expand abbreviations. Results of the statistical comparison—shown in Table 4.12—indicate, as expected, that plain English words always exhibit a significantly higher accuracy than abbreviations. The OR shows that the chances of correctly splitting/expanding plain English words are 2-4 times higher than abbreviations. Interestingly, in Exp II there is no significant difference between splitting/expanding acronyms and plain English words. The latter result suggests that participants of Exp II did not face

Table 4.11 Proportions of kind of identifiers' terms correctly expanded per context level.

Exp I				
Context	Kind of term	# Matched	# Unmatched	Accuracy (%)
file plus Acronym Finder	abbreviation	523	169	75.58
	acronym	112	31	78.32
	plain	336	50	87.05
file	abbreviation	542	164	76.77
	acronym	94	32	74.60
	plain	346	50	87.37
function	abbreviation	582	161	78.33
	acronym	97	36	72.93
	plain	374	52	87.79
no context	abbreviation	467	248	65.31
	acronym	82	47	63.57
	plain	326	75	81.30
OVERALL	abbreviation	2114	742	74.02
	acronym	385	146	72.50
	plain	1382	227	85.89
Exp II				
Context	Kind of term	# Matched	# Unmatched	Accuracy (%)
application plus Acronym Finder	abbreviation	274	69	79.88
	acronym	57	13	81.43
	plain	181	17	91.41
application	abbreviation	266	87	75.35
	acronym	57	12	82.61
	plain	180	19	90.45
file plus Acronym Finder	abbreviation	295	61	82.87
	acronym	63	10	86.30
	plain	176	16	91.67
file	abbreviation	272	84	76.40
	acronym	57	10	85.07
	plain	162	13	92.57
no context	abbreviation	242	114	67.98
	acronym	51	16	76.12
	plain	162	31	83.94
OVERALL	abbreviation	1349	415	76.47
	acronym	285	61	82.37
	plain	861	96	89.97

difficulties when splitting/expanding acronyms. This can be justified by their knowledge of the domain, and thus, of its acronyms that play a important role besides their proficiency in English. In summary, such a finding suggests that—at least for Exp II–the major issue for participants was to split/expand abbreviations, and not acronyms.

In summary, findings of **RQ2** indicate that:

- As expected, in both experiments, plain English words are handled better than abbreviations and acronyms.

Table 4.12 Participants' performances on different kind of identifiers' terms per context level: Fisher exact test results.

Exp I				
Context	Kind 1	Kind 2	OR	Adj. p-value
file	acronym	abbreviation	0.889	0.649
	plain	abbreviation	2.093	**<0.001**
	plain	acronym	2.351	**0.002**
file plus Acronym Finder	acronym	abbreviation	1.167	0.520
	plain	abbreviation	2.170	**<0.001**
	plain	acronym	1.858	**0.040**
function	acronym	abbreviation	0.746	0.177
	plain	abbreviation	1.989	**<0.001**
	plain	acronym	2.664	**<0.001**
no context	acronym	abbreviation	0.927	0.690
	plain	abbreviation	2.307	**<0.001**
	plain	acronym	2.486	**<0.001**
Exp II				
Context	Kind 1	Kind 2	OR	Adj. p-value
application plus Acronym Finder	acronym	abbreviation	1.104	0.870
	plain	abbreviation	2.677	**0.001**
	plain	acronym	2.419	0.057
application	acronym	abbreviation	1.552	0.217
	plain	abbreviation	3.093	**<0.001**
	plain	acronym	1.989	0.170
file plus Acronym Finder	acronym	abbreviation	1.302	0.604
	plain	abbreviation	2.271	**0.013**
	plain	acronym	1.742	0.493
file	acronym	abbreviation	1.758	0.176
	plain	abbreviation	3.840	**<0.001**
	plain	acronym	2.178	0.176
no context	acronym	abbreviation	1.500	0.392
	plain	abbreviation	2.458	**<0.001**
	plain	acronym	1.636	0.392

- In one case (Exp II), participants were able to handle acronyms with performances not significantly different from plain English words. Intuitively, one could probably be tempted to consider abbreviations easier to handle than acronyms, because they are often obtained by dropping some letters (often vowels) from the original words. However, it turns out that this is not the case. Our interpretation is that most of the acronyms (at least within a specific domain) lead towards a unique expansion, and many of them are well known by participants especially those knowledgable in the

domain *i.e.*, Linux in our case. Also, additional context levels such as Acronym Finder, or the availability of the entire application source code can sometimes provide a useful support for expanding acronyms. Instead, many abbreviations can lead to multiple possible expansions, hence causing imprecisions, *e.g.*, *cntr* can be expanded to *counter* or *control*.

- The high performances obtained by participants of Exp II for acronyms rather than abbreviations can be explained by their high level of English as shown in Table 4.1. The effect of such a population variable will be further investigated in **RQ3**.

In summary, we can conclude that plain English words are handled/recognized better than abbreviations and acronyms, that there is, in general, no significant difference in splitting and expanding acronyms and abbreviations, although in some case (Exp II) acronyms are easy to split/expand than abbreviations.

4.4.3 RQ3: Effect of Population Variables

This subsection reports results concerning the interaction of population variables—*i.e.*, knowledge of Linux (domain knowledge), knowledge of the C programming language, knowledge of English, and participants' background.

Table 4.13 F-measure: two-way permutation test by context & knowledge of Linux.

Exp I				
	Df	R Sum Sq	R Mean Sq	Pr(Prob)
Context	3.000	9.802	3.267	< **0.001**
Linux	2.000	0.031	0.015	0.841
Context:Linux	6.000	0.587	0.098	0.309
Residuals	1668.000	142.012	0.085	
Exp II				
	Df	R Sum Sq	R Mean Sq	Pr(Prob)
Context	4.000	5.594	1.398	< **0.001**
Linux	2.000	0.586	0.293	**0.037**
Context:Linux	8.000	0.150	0.019	0.988
Residuals	1035.000	91.541	0.088	

Table 4.13 shows, for both experiments, the two-way permutation test of F-measure by context (Context row) and Linux knowledge (Linux row). As the table shows, for Exp I the Linux knowledge has no effect on the F-measure. Also, there is no significant interaction (Context:Linux row) between the two factors. Thus, there is no evidence of a correlation

Table 4.14 Knowledge of Linux (Exp II): results of the Tukey's HSD test.

Pair of Linux knowledge levels	diff	lwr	upr	Adj. p-value
2: basic-1: occasional	-0.021	-0.072	0.029	0.587
3: knowledgeable-1: occasional	-0.059	-0.113	-0.005	**0.027**
3: knowledgeable-2: basic	-0.037	-0.097	0.021	0.297

between the accuracy (*i.e.*, F-measure) of identifier splitting/expansion and the knowledge of Linux.

In Exp II, however, the Linux knowledge has a significant effect, although it does not interact with the context. As reported in Table 4.14, results of the Tukey's HSD test show a significant difference between participants having a "knowledgeable" level of Linux and participants having an "occasional" one.

Table 4.15 shows results of the two-way permutation test of F-measure by context (Context row) and C experience (CExp row). As the table shows, for both experiments C experience has no effect on the F-measure. Also, there is no significant interaction (Context:CExp row) between the two factors.

We concluded that there is no evidence of a correlation between identifier splitting/expansion and C experience. The C experience does not play an important role because, in our understanding, when preforming identifier splitting/expanding tasks, most of the participants do not try to perform a thorough source code understanding. Rather, they try to get an idea about the context by reading comments or other identifiers parts of the same context for example.

Table 4.16 shows results of the two-way permutation test of F-measure by context (Context row) and program of studies (Program row). As the table shows, the program of studies has no effect on the F-measure. Also, there is no significant interaction (Context:Program row) between the two factors. We concluded that there is no evidence of a correlation between identifier splitting/expansion and the program of studies.

Finally, Table 4.17 reports the two-way permutation test of F-measure by context (Context row) and English proficiency (English row). For Exp I, as the table shows, not only does the English proficiency have a significant effect on the F-measure (p-value=0.032), but there is also a marginal interaction (Context:English row, p-value=0.54) between the context and the English knowledge. The level of English knowledge also plays a significant effect in Exp II, although in this case the permutation test does not show any significant interaction.

In Table 4.18, we report results of the post-hoc analysis using the Tukey's HSD test. As

Table 4.15 F-measure: two-way permutation test by context & knowledge of C.

Exp I				
	Df	R Sum Sq	R Mean Sq	Pr(Prob)
Context	3.000	7.596	2.532	< **0.001**
CExp	2.000	0.289	0.144	0.194
Context:CExp	6.000	0.300	0.050	0.741
Residuals	1668.000	142.040	0.085	
Exp II				
	Df	R Sum Sq	R Mean Sq	Pr(Prob)
Context	4.000	5.346	1.337	< **0.001**
CExp	2.000	0.032	0.016	1.000
Context:CExp	8.000	0.484	0.061	0.703
Residuals	1035.000	91.760	0.089	

the table shows, in Exp I subjects having a good English proficiency perform significantly better than those having a bad proficiency. In Exp II, subjects having a good or very good English proficiency significantly outperform those having a bad proficiency.

In summary, we can conclude that *the English proficiency significantly influences the ability of participants to split/expand identifiers*. This conclusion reveals that the English proficiency is used besides the domain knowledge that developers have about the programs they are dealing with, to understand the source code, and hence disambiguate the concepts conveyed by source code identifiers.

4.5 Qualitative Analysis

In this section, we report a qualitative analysis based on (i) the post-experiment questionnaires, and (ii) observations obtained by monitoring participants during the two experiments. Also, we provide some illustrative examples, discussing cases in which the splitting/expansion of identifiers was (not) correctly performed.

4.5.1 Exp I - Post Experiment Questionnaire Results

As it can be noticed from Fig. 4.3, participants agreed on the usefulness of the information provided in the experiment procedures: 16 participants found this information very helpful and 26 participants found it helpful. In summary, **Q1** indicates that, overall, participants correctly understood the experimental procedure and did not experience major problems in performing the tasks.

Table 4.16 F-measure: two-way permutation test by context & program of studies.

Exp I				
	Df	R Sum Sq	R Mean Sq	Pr(Prob)
Context	3.000	2.860	0.953	< **0.001**
Program	3.000	0.396	0.132	0.199
Context:Program	9.000	0.170	0.019	0.992
Residuals	1664.000	142.059	0.085	
Exp II				
	Df	R Sum Sq	R Mean Sq	Pr(Prob)
Context	4.000	4.633	1.158	< **0.001**
Program	3.000	0.093	0.031	0.799
Context:Program	12.000	0.644	0.054	0.863
Residuals	1030.000	96.278	0.093	

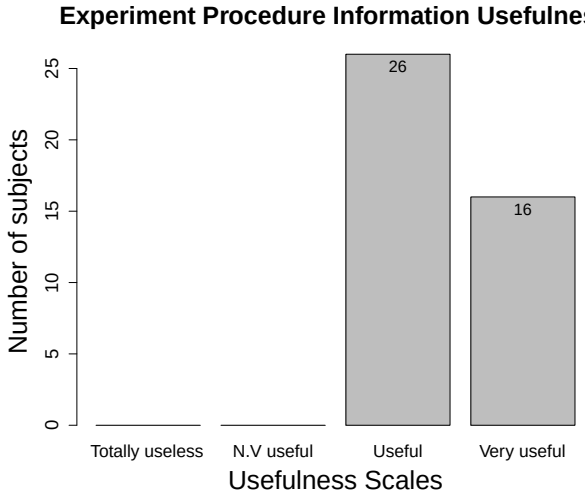


Figure 4.3 Exp I - Post-experiment questionnaire: usefulness of experiment procedure.

Regarding **Q2**, which concerns the relevance of contextual information (cf. Fig. 4.4(a)), 30 participants agreed that the most helpful context for them was the file context level. Eight participants found that the function level was the most useful for them. Yet, only four participants found that the file plus Acronym Finder level was the most helpful during the experiment. This confirms the quantitative results of RQ1 (Context Relevance), *i.e.*, the file level is the most relevant context level, and the Acronym Finder does not introduce significant additional benefits.

Concerning **Q3**, *i.e.*, usefulness of C knowledge, Fig. 4.4(b) shows that eight participants

Table 4.17 F-measure: two-way permutation test by context & English proficiency.

Exp I				
	Df	R Sum Sq	R Mean Sq	Pr(Prob)
Context	3.000	8.224	2.741	< **0.001**
English	3.000	0.728	0.243	**0.032**
Context:English	9.000	1.438	0.160	0.054
Residuals	1664.000	140.491	0.084	
Exp II				
	Df	R Sum Sq	R Mean Sq	Pr(Prob)
Context	4.000	3.172	0.793	< **0.001**
English	3.000	0.714	0.238	**0.044**
Context:English	12.000	0.799	0.067	0.698
Residuals	1030.000	90.763	0.088	

found that their experience in C programming was totally useless when splitting and expanding source code identifiers. However, 14 participants found that their C experience was not very useful when performing such a task, and 17 participants found it a bit useful. Only three participants indicated that their C experience was helpful. In summary, participants perceived the knowledge of C programming language not particularly useful for the task. The latter observation confirms the quantitative results of RQ3 (Effect of Population Variable) summarized in Table 4.15.

Regarding **Q4** that deals with the knowledge of Linux utilities. As Fig. 4.4(c) shows, only three participants found that their Linux knowledge was very useful for them when splitting and expanding the identifiers given to them, and five participants found it a bit useful. Yet, ten participants found that their Linux knowledge was not very useful, and 24 found it totally useless. In summary, most of the participants perceived the Linux knowledge not particularly useful when performing identifier splitting and expansion. The latter observation confirms the quantitative results of RQ3 (Effect of Population Variable) summarized in Table 4.13 and 4.14.

As Fig. 4.4(d) indicates, five participants found the source comments very useful, 33 found them useful, and one participant found it a bit useful when performing identifier splitting/expansion tasks. Only three participants found the comments not very useful. Hence, most of the participants agreed on the usefulness of source code comments when splitting/expanding source code identifiers.

Our observations also reveal the importance of having a good level of English proficiency when performing identifier splitting and expansion tasks. In fact, English native speaker

Table 4.18 English proficiency: results of the Tukey's HSD test.

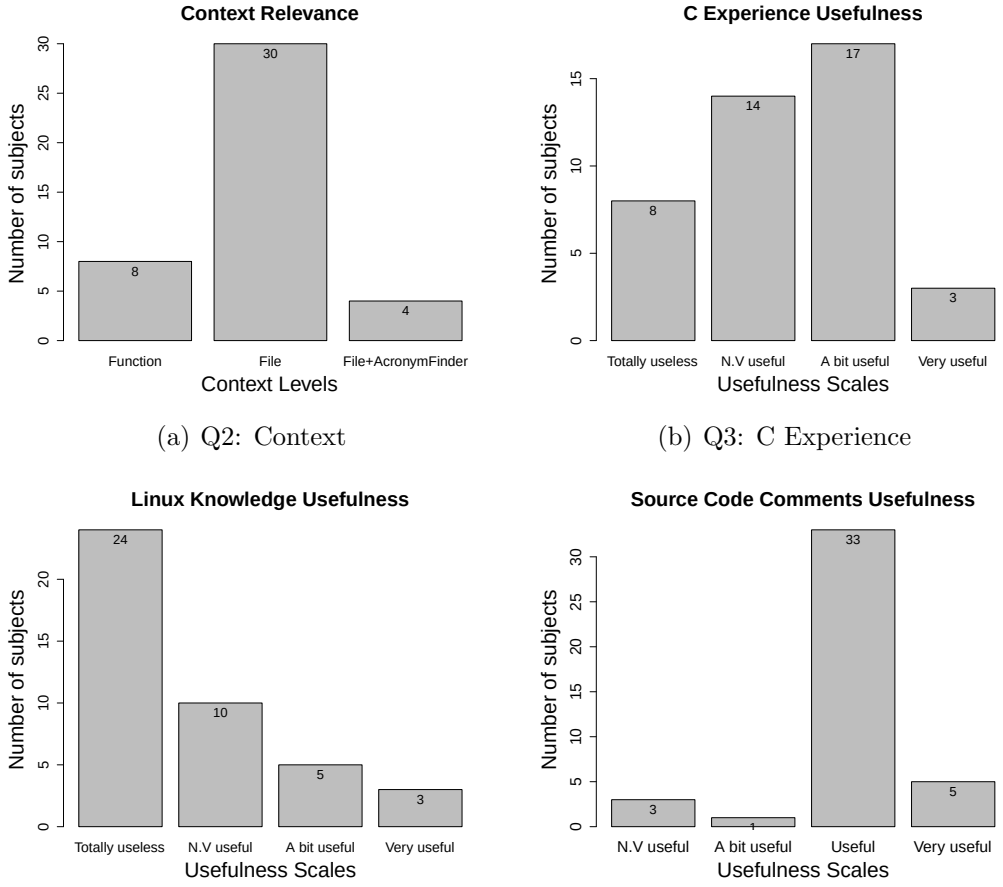
Exp I				
Pair of English knowledge levels	diff	lwr	upr	Adj. *p*-value
2: good-1: bad	-0.063	-0.122	-0.004	**0.030**
3: very good-1: bad	-0.035	-0.086	0.014	0.253
4: excellent-1: bad	-0.051	-0.110	0.007	0.110
3: very good-2: good	0.027	-0.022	0.077	0.500
4: excellent-2: good	0.011	-0.047	0.070	0.957
4: excellent-3: very good	-0.015	-0.065	0.034	0.853
Exp II				
Pair of English knowledge levels	diff	lwr	upr	Adj. *p*-value
2: good-1: bad	-0.094	-0.219	0.030	0.209
3: very good-1: bad	-0.120	-0.237	-0.003	**0.040**
4: excellent-1: bad	-0.118	-0.231	-0.005	**0.035**
3: very good-2: good	-0.025	-0.102	0.050	0.818
4: excellent-2: good	-0.024	-0.094	0.046	0.813
4: excellent-3: very good	0.001	-0.053	0.056	0.999

participants (seven of them), or those having a good/excellent English proficiency, did not face difficulty in recognizing English words in identifiers, even in presence of abbreviations and acronyms. The latter observation confirms the quantitative results of RQ3 (Effect of Population Variable) summarized in Table 4.17 and 4.18.

4.5.2 Exp II - Post Experiment Questionnaire Results

Similarly to Exp I, we collected and analyzed data gained from the post-experiment questionnaire that we gave to the participants at the end of the experiment and where the main questions are summarized in Table 4.7.

Regarding **Q1** (usefulness of information provided by the experiment material), as shown in Fig. 4.5, all participants agreed that this information was helpful for them, which means that they did not face any problem when performing the tasks. For what concerns **Q2** (relevance of context), Fig. 4.6(a) shows that 10 participants found that the application-level context was helpful for them, five agreed that the application plus Acronym Finder was the most helpful for them, four found the file-level context more important when performing the identifier splitting and expansion tasks, and only two participants claim that the file plus Acronym Finder was the most helpful level for them. The positive feedbacks about the usefulness of the application level (partially) contrasts the quantitative results obtained in



(a) Q2: Context

(b) Q3: C Experience

(c) Q4: Linux Knowledge

(d) Q5: Source code comments

Figure 4.4 Exp I - Post-experiment questionnaire: context and participants' background relevance.

RQ1, which indicated that application level context did not introduce significant additional benefits with respect to file level context. Nevertheless, some participants indicated that they found particularly useful the possibility to browse header files to better understand identifiers used in C files. The latter observation highlights that for languages like C/C++ header files are a very important source of information, because very often functions, types, data structures, or classes (C++) are defined there.

For **Q3**, we notice from Fig. 4.6(b) that 12 participants agreed that their experience in C was not very helpful when performing the given tasks, and only nine participants claim that their C expertise was a bit helpful for such experimental tasks, and this is in accordance with the quantitative results of Exp II obtained for RQ3 and summarized in Table 4.15. Regarding **Q4**, Fig. 4.6(c) reveals that one participant found that his knowledge in Linux was totally useless. Yet, nine participants agreed that their knowledge in Linux was not very useful,

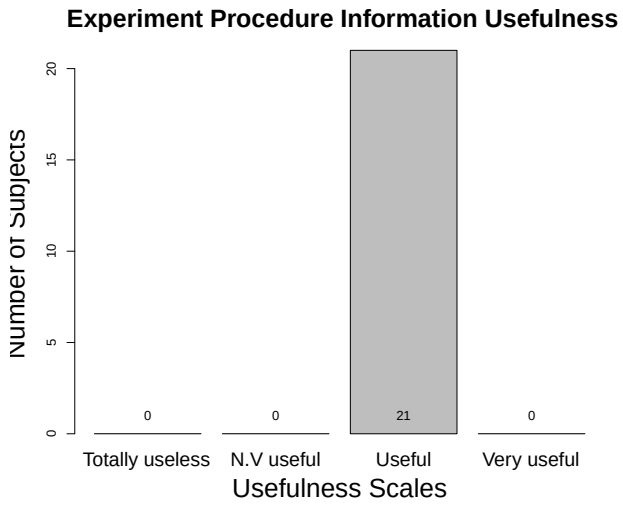
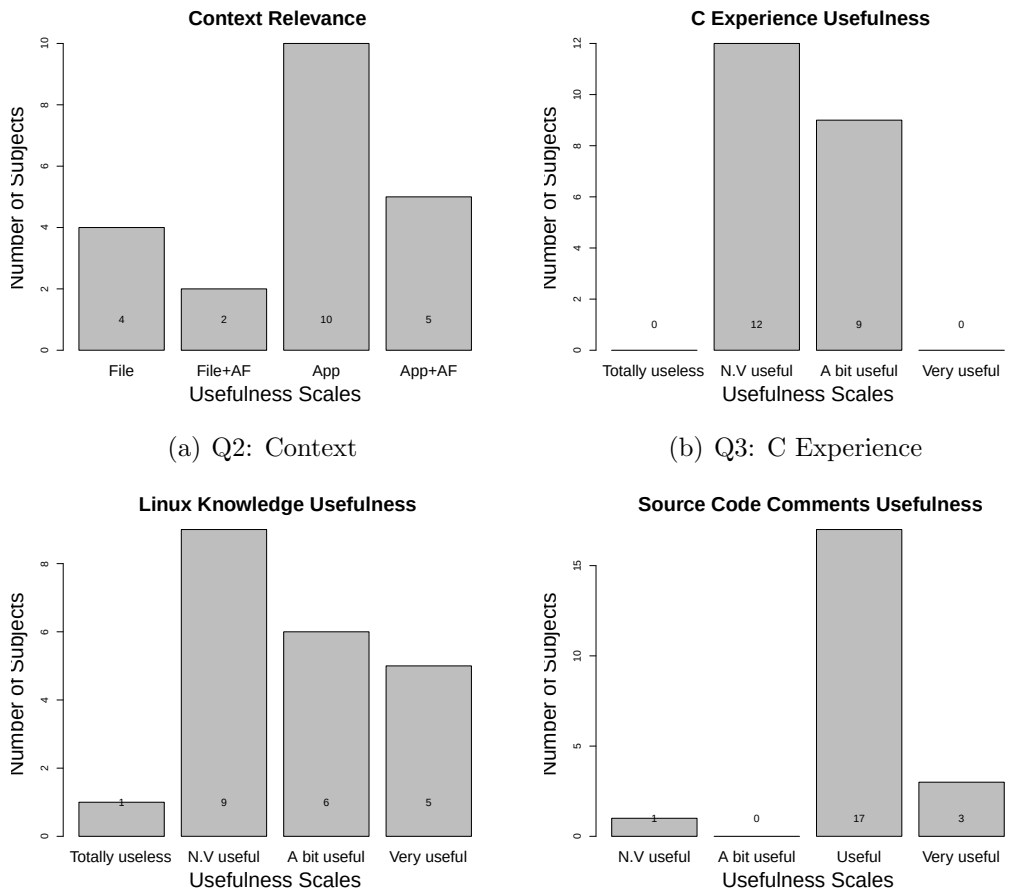


Figure 4.5 Exp II - Post-experiment questionnaire: usefulness of experiment procedure.

six mentioned that it was a bit useful when splitting and expanding the given C identifiers, and five claim that the Linux knowledge was very helpful for them. The latter observation confirms the quantitative results of Exp II obtained for RQ3 and summarized in Tables 4.13 and 4.14. The significant results obtained for Exp II can be explained by the fact that most of the participants in Exp II know (according to our interviews with them) the applications from where we sampled our identifiers, thus, they benefited from their knowledge of the domain (*i.e.*, Linux) when performing the experimental tasks of Exp II. Such homogeneity does not apply to the participants of Exp I. As Fig. 4.6(d) shows, only one participant found the source code comments not very useful, 17 participants found them useful, and three found the source code comments very useful.

As shown by Table 4.1, participants in Exp II are homogenous in terms of C expertise and the English proficiency. In fact, most of the participants have a good to excellent English proficiency (six are English native speakers), having such a knowledge of English, participants of Exp II did not face difficulties when performing the experimental tasks. Also, the participants have a medium to expert knowledge in C programming. Such homogeneity does not apply to Exp I participants.

Overall, the post-experiments questionnaires results confirm the quantitative results presented in Section 4.2, *i.e.*, the increase of the context help participants split/expand identifiers. The application and file levels seem to be the most helpful context levels (even though the difference between the two levels is not statistically significant). The external information (*i.e.*, the Acronym Finder) is not always useful, especially when the acronyms are specific to the application in question. However, the source code comments were found helpful by almost



(a) Q2: Context

(b) Q3: C Experience

(c) Q4: Linux Knowledge

(d) Q5: Source code comments

Figure 4.6 Exp II - Post-experiment questionnaire: context and participants' background relevance.

all the participants. Results of this section confirms the results reported in Section 4.2 for what concerns the participants' characteristics and background. In fact, the C programming expertise was found not useful for the identifier splitting/expansion task. Yet, the Linux knowledge was helpful for some participants, confirming that for Exp II it had a significant effect.

4.5.3 Illustrative examples from the data exploration

To provide insights about identifier splitting/expansion difficulties faced by the participants of the two experiment, Table 4.19 reports examples of wrong splits/expansions with brief explanations about the cause of the failure in expanding these identifiers. As it can be noticed from the table, wrong splits/expansions were mainly due to single letters (*e.g.*, *p* in *mempcpy*, which some participants did not recognize), and abbreviations composed of only

two letters (*e.g.*, *rw* in *case_data_rw_idx*). Wrong answers were also due to incorrect splits (*e.g.*, *dupok* that was incorrectly split into *directory*, *up*, and *ok*), or to the inability to recognize terms composing an identifier, as in the case of *lf_isset* that was kept as it is, instead of being split into *is* and *set*. Difficulties were also faced when expanding few acronyms such as (*e.g.*, *dbsm* in *dbsm_start* and *arm* in *arm_reg_parse*). Other wrong expansions were noticed when a term part of an identifier appeared to be an English word, but indeed it required an expansion which participants did not apply. For example, *dumbterm* was split into *dumb* and *term* instead of *dumb* and *terminal*. Then, we investigated how participants expanded acronyms, and to what extent the Acronym Finder was helpful for that. To this aim, we classified acronyms into:

Table 4.19 Examples of wrong splits and expansions.

Identifier	Wrong Expansion	Identifier Oracle	Type of Mistake
mempcpy	memory **p** copy	memory pointer copy	correct split but inability to expand the letter *p* into *pointer*
case_data_rw_idx	case data **read write** index	case data row index	correct split but incorrect expansion of *rw* into *read write* instead of *row*
dupok	**dupok**	duplicate ok	inability to correctly split *dupok* into *dup ok* and expand it into *duplicate ok*
fpcw_new_offset	floating point control word new offset	**f p** control word new offset	correct split and inability to expand the letters *f* and *p* in *fpcw* into *floating* and *point*
pendulist	**pend** user list	pending user list	correct split and incorrect expansion by keeping *pend* as it is instead of *pending*
pmat	**private** matrix	partitioned matrix	incorrect expansion of *p* to *private* instead of *partitioned*
rm_so	remove **socket**	remove shared object	incorrect splitting and expansion of *rm_so* into *remove socket* instead of *remove shared object*
ipfrag	internet protocol **fragmentation**	internet protocol fragment	correct split and incorrect expansion of *frag* to *fragmentation* instead of *fragment*
assoc	**associate**	association	incorrect expansion of *assoc* to *associate* instead of *association*
dupok	**directory up** ok	duplicate ok	incorrect splitting and expansion of *dupok* into *directory up ok* instead of *duplicate ok*
dbsm_start	**data base state machine** start	decibel per square meter start	incorrect expansion of the acronym *dbsm*
internal_auxent	internal auxiliary **entry**	internal auxiliary entities	incorrect expansion of the abbreviation *ent* to *entry* instead of *entities*
extcase	**external** case	**extended** case	incorrect expansion of the abbreviation *ext* to *external* instead of *extended*
pl_stm_tbl	prolog **statement** table	prolog stream table	incorrect expansion of the abbreviation *stm* to *statement* instead of *stream*
dumbterm	dumb **term**	dumb terminal	keeping the abbreviation *term* as it is instead of expanding it to *terminal* towards a wrong expansion of *dumbterm*

1. $AF \wedge ORA$: acronyms that have an expansion proposed by the Acronym Finder that matches the one of our oracle, 50% of acronyms fall into this category;

Table 4.20 Proportions of correctly expanded acronyms with the file plus Acronym Finder context.

Identifiers	$AF \wedge ORA$	$AF \wedge not(ORA)$	$notAF$	Matched – Exp I (%)	Unmatched – Exp II (%)
API_SIGNON	√			100	100
argv_to_**scm**	√			0	20
arm_reg_parse		√		0	0
bfd_abs_section_ptr	√			78	100
blk_queue_**io**_stat	√			100	100
dbsm_start	√			0	0
dfp		√		67	100
esi	√			25	25
FFEBAD_severityFATAL		√		37.5	50
FFI_Ok	√			100	75
fpcw_new_offset			√	25	20
gmon_**io**_write	√			100	100
GMP_NUMB_MAX	√			75	40
hol_cluster_base		√		50	66.7
ipfrag	√			100	100
pBt		√		12.5	50
PNG_INFO_PLTE	√			62.5	50
rrt_prev		√		12.5	0
scm_set_smob_print	√			25	0

2. $AF \wedge not(ORA)$: acronyms that have an expansion proposed by the Acronym Finder that does not matches the one of our oracle, 38% of acronyms fall into this category;

3. *not AF*: acronyms that are not in the Acronym Finder, 12% of acronyms fall into this category.

We computed the proportions of correctly expanded acronyms for the file plus Acronym Finder context, *i.e.*, the common context level between Exp I and Exp II that uses the Acronym Finder as an external source of information. The proportions of correctly expanded acronyms (Matched) are reported in Table 4.20.

For what concerns the category $AF \wedge ORA$, participants were able to expand 100% of well-known acronyms, *i.e.*, *API* in the identifier *API_SIGNON* *ip* in *ipfrag*, and *io* in the identifiers *gmon_io_write* and *blk_queue_io_stat*. In addition, some acronyms were expanded in high proportions by participants of Exp II while others were easily handled by participants of Exp I. Examples of acronyms that were expanded in high proportions by participants of Exp II are *bfd* in the identifier *bfd_abs_section_ptr* (with a percentage of 100% versus 75% in Exp I), *scm* in the identifier argv_to_scm (20% of cases, versus no correct expansion in Exp I).

Acronyms that were better expanded by the participants of Exp I are *GMP* in the identifier GMP_NUMB_MAX (75% in Exp I versus 40% in Exp II), *scm* in *scm_set_smob_print* (25% in Exp I versus 0% in Exp II), *FFI* in *FFI_Ok* (100% in Exp I and 75% in Exp II), *PNG* in *PNG_INFO_PLTE* (62.5% in Exp I versus 50% in Exp II).

The acronym *esi* was correctly expanded with a proportion of 25% in both Exp I and Exp II. The acronym *dbsm* was not correctly expanded by any participant.

Overall, only well-known acronyms were expanded in 100% of cases, some were expanded in different proportions less that 100% by participants of Exp I and II, and others were not correctly expanded by any participant (*e.g.*, *dbsm*) even if their expansions exist among the expansions provided by the Acronym Finder. Thus, we concluded that the availability of the Acronym Finder is not particularly useful, and that it could even be misleading by providing several choices of expansions that are sometimes too generic, and do not fit with the exact context/domain of the applications participants are dealing with.

For what concerns the second category of acronyms ($AF \wedge not(ORA)$), participants of Exp II expanded a high proportion of almost all acronyms, and they were able to achieve an accuracy between 50% and 100%. These further remarks show that the internal (source code and comments) context is more useful than the external context (Acronym Finder). The high proportion of acronym expansions provided by the participants of Exp II can be justified by their high English proficiency as indicated in Table 4.1 and also by their knowledge of the domain (Linux) as reported in Table 4.13.

Finally, we had only one acronym that does not exist in the Acronym Finder (*not AF*), *i.e.*, *fpcw* in the identifier *fpcw_new_offset*. Participants were still able to correctly expand it with a proportion of 25% in Exp I and 20% in Exp II.

Overall, we can conclude from these examples that, on the one hand, adding an external source of information to the participants is not often helpful if this does not properly reflect the application context and domain. On the other hand, the participants' domain knowledge can, sometimes, play an important role, and their English proficiency turns out to be very useful.

In summary, we can conclude from both qualitative and quantitative analyses that the increase of the context help participants split/expand identifiers. The application and file levels are the most helpful context levels (even though the difference between the two levels is not statistically significant). The external information (*i.e.*, the Acronym Finder) is not always useful if this does not properly reflect the application context and domain of the application. Regarding participants' characteristics and background, C programming expertise was not useful for the identifier splitting/expansion task. Yet, the knowledge of the domain (Linux) was helpful for some participants (Exp II), and their English proficiency turns out to be very useful.

4.6 Threats to Validity

Threats to **construct validity** concern the relation between the theory and the observation. In our study, this threat is mainly due to possible mistakes in the oracle, which we cannot exclude a priori. To limit such a threat, the oracle was produced using a consensus approach, *i.e.*, two authors produced independent oracles, and then discussed cases of disagreement. We proceeded in such a way to minimize the bias and the risk of producing erroneous results. This decision was motivated by the complexity of identifiers, which capture developers' domain and solution knowledge, personal preferences, etc.

Threats to **internal validity** concern any confounding factor that could influence our results. One case of an internal validity threat is the one related to learning and fatigue effect. This threat is addressed in our experiment by (i) using different treatments—including different ordering in which identifiers were shown to the participants—for each group of participants in the experimental design (see Table 4.4), and (ii) having a task and lab duration of a reasonable size, *i.e.*, 40 identifiers (Exp I) and 50 identifiers (Exp II) for each participant, and 120 minute laboratory session.

Another threat could be the one related to the subjectivity in the answers provided in the pre-experiment questionnaire. Since we asked subjects to rate themselves for what concerns their knowledge of Linux, C programming language, and English, we are aware that the collected information can contain over-positive and over-negative assessments. To limit threats related to variation of participants' performance in the experimental tasks, we divided participants into groups ("blocks") of participants having, roughly, the same level of skills/experience, and then performed a stratified sampling to make sure that skill/expertise is uniformly distributed across groups in the experimental designs of Table 4.4.

A further internal validity threat is the diffusion or imitation of treatments. This threat is also limited by preventing access to the experiment material outside the experiment hours by other groups' members. Also, although participants were aware of the laboratory objectives—*i.e.*, splitting and expanding identifiers using any source of information available—they did not know exactly the experimental hypotheses.

Threats to **conclusion validity** are concerned with issues that affect the ability to draw the correct conclusions about relations between the treatment and the outcome of the experiment. We used non-parametric tests—*e.g.*, Wilcoxon and permutation tests—which do not make any assumption on the underlying distributions of the data set. Also, whenever multiple Wilcoxon tests are performed, we adjusted p-values using the Holm's correction. Finally, other than the presence of significant differences, we also analyzed the magnitude of the detected differences using a non-parametric effect-size measure, *i.e.*, Cliff's delta.

Threats to **external validity** concern the possibility of generalizing our results. This relates to (i) the choice of object programs from which identifiers to split/expand have been sampled and (ii) the choice of the experiment participants. To make our results as generalizable as possible, we randomly selected our sample of identifiers from a set of open-source project. We only considered C programs rather than Java programs because previous studies have already shown that in most cases Java identifiers can be split/expanded trivially, *e.g.*, using a simple CamelCase heuristic (Madani *et al.*, 2010; Guerrouj *et al.*, 2012). This is because Java identifiers are usually built using the CamelCase convention and, quite often, are composed of full English words rather than abbreviations and/or acronyms. Instead, the usage of a more complex splitting/expansion is particularly useful for programming languages that use short identifiers (*e.g.*, C, C++, and COBOL).

The participants that performed the experiments belong to a population of Canadian students (Bachelor, Master, Ph.D.) and post-docs. Many of them already had previous industrial experience. Nevertheless, we are aware that the context in which our experiment was performed is still an academic one; therefore, replications in industrial settings are highly desirable.

4.7 Chapter Summary

This work is a family of two controlled experiments investigating the effect of context in one of the practical tasks in software maintenance and evolution, that is the splitting and expansion of source code identifiers. More specifically, we investigated the extent to which a source-code context could be helpful when splitting/expanding source code identifiers, and the extent to which other factors related to identifiers' characteristics and to developers skill/experience could influence participants' performances or interact with the effect of context.

The experiments involved 63 participants, students (Bachelor, Master, Ph.D.) and post-docs from the École Polytechnique de Montréal, and used, as objects, a set of 50 identifiers randomly sampled from a corpus of C open-source programs. Exp I involved 42 participants, and investigated four different context levels: (i) splitting/expanding identifiers without any contextual information, (ii) function-level context, (iii) file-level context, and (iv) file plus the availability of an external context, *i.e.*, the Acronym Finder. Exp II involved 21 participants and, in addition to the Exp I contexts (excluding the function context only), considered the application level context, with and without the Acronym Finder.

The experimental results provided evidence on the usefulness of context for identifier splitting and expansion. In particular, results indicated that a wider context—*i.e.*, file-

level—is more helpful than a limited one—*i.e.*, function level. However, a wider context, *i.e.*, application level, does not introduce further improvements. In general, the presence of an Acronym Finder did not introduce significant benefits, likely because the acronyms contained in the identifiers are not domain specific.

We found no significant difference in the accuracy of splitting and expanding terms consisting of abbreviations from those consisting of acronyms. However, while abbreviations are always significantly more difficult to split/expand than English words, in Exp II this is not the case for acronyms. This result highlights that, counterintuitively, developers would need support to split/expand abbreviations, while acronyms are not always a big issue. Thus, identifier splitting/expansion tools that exploit contextual information (Enslen *et al.*, 2009; Lawrie *et al.*, 2010; Lawrie et Binkley, 2011; Corazza *et al.*, 2012) are particularly useful.

Results also show that, in both experiments, the participants' level of English plays a significant role in the identifier splitting and expansion performance. In particular, participants are able to better exploit contextual information if they have a very good knowledge of English. Instead, C experience and participants' background (program of studies) do not have a significant effect. The knowledge of Linux revealed to have a significant effect only in Exp II.

In general, the obtained results can be used to provide useful insights to practitioners and researchers, confirming the belief about the relevance of contextual information in program comprehension. Such information is helpful not only to humans when performing program comprehension tasks, but also to automatic tools that rely on source code lexicon to perform various kinds of tasks, including feature location (Dit *et al.*, 2011).

CHAPTER 5

Context-Aware Source Code Vocabulary Normalization Approaches

Prior works on normalization have focused on identifier splitting (Hill *et al.*, 2011). The widely used techniques rely on CamelCase naming conventions. The CamelCase convention is the practice of creating identifiers by concatenating terms with capitalized first letter, giving identifiers a Camel-like looking with flats and humps, *e.g.*, *drawRectangle*. It leads to the development of a family of algorithms to split identifiers into component terms. These algorithms have in common the assumption that the CamelCase convention and–or an explicit separator are systematically used to create identifiers. Samurai can be thought as a clever CamelCase that mines terms frequencies in programs under analysis and in a large corpus of programs to perform the identifier splitting. The main weakness of Samurai is its reliance on frequency tables which may lead to over-splits (Enslen *et al.*, 2009). To overcome the latter shortcomings, we suggest two novel context-aware approaches that both split and expand source code identifiers even in the absence of naming conventions and–or presence of abbreviations. We use context because our experimental results show that it is relevant for vocabulary normalization (cf. Chapter 4).

In this chapter, we will present our contributions to source code vocabulary normalization, *i.e.*, TIDIER and TRIS. TRIS is inspired by TIDIER. However, it deals with the identifier splitting/expansion problem differently and uses a tree-based representation that considerably reduces its complexity and makes it fast.

5.1 TIDIER

When writing source code, in particular when naming source-code identifiers, developers make use of concepts from high-level documentation and from the program domain. Also, they encode identifiers using implicit and explicit coding conventions and–or past experience. The goal of TIDIER is to split program identifiers using high-level and domain concepts by associating identifier terms to domain-specific words or to words belonging to some generic English dictionaries.

First, TIDIER assumes that it is possible to model developers creating an identifier with a set of transformation rules on terms/words. For example, to create an identifier for a variable that stores a number of customers, the two words *number* and *customers* can be concatenated with or without an underscore, *e.g.*, *customer_number* or *customernumber* or following the

CamelCase convention, *e.g.*, *customerNumber*. Contractions of one or both words are also possible, leading to identifiers such as *customerNbr*, *nbr_customer*, or *cstmr_nbr*.

Second, TIDIER assumes that it is possible to define a distance between dictionary words and identifier terms to quantify how close are words, representing concepts, to such terms and, thus, to provide a measure of the likelihood that the terms refer to some words. Although there are several ways—none of which are the best—to compute a distance between two terms and words, string-based distances have been used in the past for various purposes, such as code differencing (Canfora *et al.*, 2009) or clone tracking (Canfora *et al.*, 2010), with good results. Therefore, TIDIER use the string-edit distance between terms and words as a proxy for the distance between the terms and the concepts they represent.

In a nutshell, TIDIER relies on a set of input dictionaries and a distance function to split (if necessary) simple and composed identifiers and associate the resulting terms with words in the dictionaries, even if the terms are truncated/abbreviated, *e.g.*, *objectPtr*, *cntr*, or *drawrect*. Dictionaries may include English words and–or technical words, *e.g.*, *microprocessor* and *database* (in the computer domain), or known acronyms, *e.g.*, *afaik* (in the Internet jargon). The distance function measures how close a given identifier term is to a dictionary word and, thus, how well the concepts associated to the dictionary words are conveyed by the identifier.

Developers of C programs sometimes use word abbreviations to compose identifiers, which is likely a heritage of the past when certain operating systems and compilers limited the maximum length of identifiers. For example, a developer may use the term *dir* instead of the word *directory*, *ptr* or *pntr* instead of *pointer*, or *net* instead of *network*. TIDIER aims to segment identifiers into terms and recover the original non-abbreviated words. Thus, TIDIER uses a thesaurus rather than English and–or domain dictionaries. A thesaurus entry, a word, in TIDIER is the original word followed by the list of abbreviated terms, *i.e.*, word synonyms; if TIDIER finds the term *ptr* in an identifier, then it knows that this term is actually an abbreviation of *pointer*. In the following, wherever there is no risk of confusion, the two terms dictionary and thesaurus will be used interchangeably to indicate a list of words; each word possibly associated with a list of abbreviations.

Some abbreviations are well-known and can, thus, be part of the thesaurus we built. In such case, each row of the thesaurus contains a word and its possible synonyms, *e.g.*, *dir* for *directory* or *direction*. Some other abbreviations may not appear in the thesaurus because they are too domain and–or developer specific. To cope with such abbreviations, TIDIER is the first approach that finds the best splitting using a string-edit distance and a greedy search. If the edit distance between a term and a word is not zero, TIDIER tries to reduce the distance by transforming the word into some possible abbreviated forms, *e.g.*, by removing all vowels *pointer* is mapped into *pntr*. Then, TIDIER recomputes the edit distance and

adds the abbreviated forms as possible synonyms of the word if the distance between the abbreviated form and the term is smaller than the previous distance between the word and the term. A hill-climbing algorithm iterates over all words and all transformation rules to obtain the best split—*i.e.*, a zero distance—or until a termination criterion is reached.

Thus, the current implementation of TIDIER takes as input an identifier and a thesaurus, and uses a simple string-edit distance to split, whenever possible, the identifier into a number of terms that have a small (or zero) distance with dictionary words. TIDIER is not able to deal with missing information or to generate abbreviations in all cases. If the identifiers use terms belonging to a specific domain, whose words are not present in the thesaurus, TIDIER cannot split and associate these terms with words. Similarly, TIDIER cannot identify the words composing acronyms, *e.g.*, *afaik*, *cpu*, *ssl*, or *imho*, because it cannot associate a single letter from the acronym with the corresponding word: for any letter, there exist thousands of words with the same string-edit distance, *e.g.*, the *c* of *cpu* has the same distance with *central* and with any other word starting with *c*.

We now detail the main components of TIDIER.

5.1.1 String Edit Distance

The string-edit distance between two given strings, also known as Levenshtein distance (Levenshtein, 1966), is the number of operations required to transform one string into another. The most common setting considers the following edit operations: character deletion, insertion, and substitution. Specifically, these settings assume that each insertion and each deletion increase the distance between the two strings by one, whereas a substitution (*i.e.*, a deletion followed by one insertion) increases it by two (Cormen *et al.*, 1990). An exact match is just a special case of substitution; it has a zero cost since both characters are the same.

Let us assume that we must compute the edit distance between the strings *pointer* and *pntr*. Their edit distance is three, as the characters *o*, *i*, and *e* must be removed from *pointer* or, alternatively, added to *pntr*. The main problem in computing the string-edit distance is

8	r	∞	6	5	4	3
7	e	∞	5	4	3	4
6	t	∞	4	3	2	3
5	n	∞	3	2	5	6
4	i	∞	2	3	4	5
3	o	∞	1	2	3	4
2	p	∞	0	1	2	3
1		0	∞	∞	∞	∞
			p	n	t	r
		1	**2**	**2**	**2**	**5**

Figure 5.1 Single Word Edit Distance Example.

the algorithm efficiency. A naive implementation is typically exponential in the string length. A quadratic complexity implementation can be easily written using dynamic programming and the algorithm is then often referred to as the Levenshtein algorithm. The Levenshtein algorithm computes the distance between a string s of length N and a string w of length M as follows.

First, a distance matrix D of $(N+1) \times (M+1)$ cells is allocated; in our example, 8×5, *i.e.*, the lengths of *pointer* and *pntr* plus one. The cells in the first column and first row are initialized to a very high value but for cell $(1,1)$, which is initialized to zero. (This allocation and initialization strategy simplifies the algorithm implementation). Matrix D can be seen as a Cartesian plane, and strings s and w, *i.e.*, *pointer* and *pntr*, as places along the plane axes starting from the second cells, as shown in Figure 5.1.

The computation proceeds column by column starting from cell $(1,1)$. The distance in cell $D(i,j)$ is computed as a function of the previously computed (or initialized) distances in cells $D(i-1,j)$, $D(i-1,j-1)$, and $D(i,j-1)$. At the end of the process, the cell $(N+1, M+1)$ contains $D(N+1, M+1)$, which is the minimum edit distance.

$$
\begin{aligned}
c(i,j) = \quad & \begin{cases} 1 & \text{if s[i]} \neq \text{w[j]} \\ 0 & \text{if s[i]} = \text{w[j]} \end{cases} \\
D(i,j) = \quad & min[D(i-1,j) + c(i,j), \quad // \; insertion \\
& \qquad D(i,j-1) + c(i,j), \quad // \; deletion \\
& D(i-1,j-1) + 2 * c(i,j)] \quad // \; substitution
\end{aligned}
$$

Unfortunately, the Levenshtein algorithm is not suitable to split identifiers because it only computes the distance between two given strings, not between sub-strings in a string (*i.e.*, identifier terms) and some other strings (*i.e.*, dictionary words).

In the early '80s, Ney proposed (Ney, 1984) an adaptation to continuous speech recognition of the dynamic programming alignment algorithm, known as Dynamic Time Warping (DTW) (Sakoe et Chiba, 1978), originally conceived for isolated word recognition. Ney's adaptation considers that a word can begin and end at any point in an utterance, similarly as a term can begin and end at any point in an identifier. It thus does not assume a-priori knowledge of where a word is located in an utterance, *i.e.*, where a term begins or ends in an identifier. The details of Ney's algorithm are available elsewhere (Ney, 1984). TIDIER implements an extension of the Levenshtein algorithm based on Ney's adaptation. This ex-

			Columns								
			1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Rows	**4**	r	∞	2	3	2	1	3	3	4	3
	3	t	∞	1	2	1	2	3	3	3	4
	2	p	∞	0	1	2	3	2	3	4	3
	1		0	∞	∞	∞	∞	∞	∞	∞	∞
				p	n	t	r	c	n	t	r
Rows	**5**	r	∞	4	4	3	2	3	3	2	1
	4	t	∞	3	3	2	3	3	2	1	2
	3	n	∞	2	2	3	3	2	1	2	3
	2	c	∞	1	2	3	2	1	2	3	3
	1		0	∞	∞	∞	∞	∞	∞	∞	∞
				p	n	t	r	c	n	t	r
Minimal Distance			∞	2	3	2	1	2	3	1	1

Figure 5.2 Multiple Words Edit Distance Example.

tension requires a dictionary (or a thesaurus) of known words (referred to as *speech template* in (Sakoe et Chiba, 1978; Ney, 1984)).

Let us suppose that we have the identifier *pntrcntr* and that our dictionary contains only the two words *ptr* and *cntr*, abbreviations of *pointer* and *counter*, respectively. The algorithm is initialized as described above for the Levenshtein algorithm, except that it creates one matrix for each word in the dictionary, as shown in Figure 5.2. The algorithm then proceeds by computing one column at a time, going from Row 2, to Row $N+1$. Row 1 and Column 1 just contain initialization values used to simplify the DTW and, thus, the actual computation goes from cell $(2,2)$ to cell $(N+1, N+1)$.

Once Column 2 is computed for all words in the dictionary as in the Levenshtein algorithm, a decision is taken on the minimum distance contained in cell $(2,4)$ for *ptr* and $(2,5)$ for *cntr*. This minimum distance is equal to 2, as shown in Figure 5.2, and the corresponding best term, *i.e.*, *ptr*, is then recorded. The minimum distance is then copied into the cell $(1,3)$ of the matrices, which corresponds to assuming that the word with lower cost ends at Column 2.

At the beginning of Column 3 (*i.e.*, to calculate $(2,3)$), the algorithm checks if it is less costly to move from one of the cells $(1,2)$ and $(2,2)$ or, instead, if it is cheaper to assume that a string was matched at Column two (previous column) with the distance cost recorded in the minimum distance array (*i.e.*, two) and copied into $(1,3)$. In the example, for both dictionary words, the algorithm decides to insert a character, *i.e.*, move to the next column (along the x axis), as previous values are lower, *i.e.*, zero for *ptr* and one for *cntr*.

When the column of the character c of *pntrcntr* is computed (Column 6), the minimum distance recorded for dictionary terms at Column 5 is one, as *ptr* just needs one character insertion to match *pntr*. Thus, the computation propagates the minimum distance in Column

5 for *ptr*, *i.e.*, *ptr* matches *pntr* with distance one, and the algorithm detects that the word *ptr* ends at Column 5. Because the character *c* is matched in *cntr*, the distance of one is propagated to cell $(6, 2)$. The last part of the identifier *pntrcntr* matches *cntr*. Thus, when all columns are computed, the lowest distance is one. Distance matrices and the minimum distance array allow one to compute the minimum string-edit distance between the terms in the identifier and the two words, and thus split the identifier.

The algorithm uses back-pointers matrices for improved performance, one for each dictionary word. For a given term, in each cell (i, j), the back-pointer matrix records the decision taken and thus words, words matching distances, as well as the beginning and end of each word is recovered.

5.1.2 Thesaurus of Words and Abbreviations

The thesaurus used by TIDIER plays an important role for the quality of its results. In the thesaurus, a word may be followed by a list of equivalent words or abbreviations. For example, the words *network* and *net* are considered equivalent and form a single row as well as the terms *pointer*, *pntr*, and *ptr*. Thus, if *pntr* is matched, TIDIER expands it into the dictionary word *pointer*.

One possibility to build such a thesaurus would be to merge different specific or generic dictionaries, such as those of spell checkers, *e.g.*, i-spell[1], which contains about 35,000 words, or of upper ontologies, *e.g.*, WordNet[2], which contains about 90,000 entries.

Yet, it would be desirable, if possible, to build smaller dictionaries, *e.g.*, dictionaries containing the most frequently-used English words only as well as specialized dictionaries containing acronyms and known abbreviations to reduce the computation time. In the following, we used five different kinds of dictionaries.

1. *Small English dictionary (referred to as "English Dictionary")*: an English dictionary built from the 1,000 most frequent English words, the 250 most frequent technical words (from the Oxford Dictionary), and 275 most frequent business words (from the Oxford Dictionary), plus words from a glossary found on the Internet[3]. Overall, this dictionary includes 2,774 words.

2. *Small English dictionary, plus specialized knowledge*: this dictionary consists of the English Dictionary plus: (i) a set of 105 acronyms used in computer science (*e.g.*, *ansi*, *dom*, *inode*, *ssl*, *url*), (ii) a set of 164 abbreviations collected among the authors used

[1] `http://www.gnu.org/software/ispell/ispell.html`
[2] `http://wordnet.princeton.edu/wordnet/`
[3] `http://www.matisse.net/files/glossary.html`

when programming in C (*e.g.*, *bool* for *boolean*, *buff* for *buffer*, *wrd* for *word*), and (iii) a set of 492 C library functions (*e.g.*, *malloc*, *printf*, *waitpid*, *access*). This dictionary includes the union of the 2,774 English words plus 761 abbreviations and C functions, for a total of 3,535 distinct words.

3. *Complete English dictionary (referred to as "WordNet")*: a complete English dictionary extracted from the WordNet upper-ontology database and from the GNU i-spell spell-checker. This dictionary includes 175,225 words.

4. *Context-aware dictionaries*: dictionaries containing function-level, source code file-level, and program-level identifiers. We built these dictionaries using dictionary words appearing in the context where the identifiers are located.

5. *Application dictionary, plus specialized knowledge*: a dictionary based on the program-level dictionary—described in the previous step—augmented with domain knowledge (abbreviations, acronyms, and C library functions).

The abbreviations used to describe specialized knowledge were collected with no prior knowledge about the identifiers to be split. The rationale of including abbreviations is to identify terms not contained in the English Dictionary but that are likely to be contained in identifiers and that could not be expanded into English words because their distance from the words that they represent is too large. For example, the identifier *ipconfig* contains the term *ip*, which means "internet protocol". It would be impossible for any algorithm to guess that *i* stands for *internet* and *p* for *protocol*. Widely used abbreviations are introduced to make the search faster as it would be useless and time consuming to generate well-known abbreviations. C library terms are introduced because, often, they correspond to jargon or domain-specific words, and C program identifiers contain these terms. For example, functions wrapping known C functions often contain terms such as *printf*, *socket*, *flush*, and so on, as in the Linux identifiers *threads_fprintf*, *seq_printf*, or, in the Apache Web server, *snprintf_flush* or *apr_socket_create*.

The context-aware dictionaries are built by tokenizing source code, extracting identifiers and comment terms, and saving them into specialized context-aware dictionaries at the level of functions, files, or programs. These lists of terms need to be pruned of strings not corresponding to English words or technical terms before being considered as usable dictionaries; in TIDIER, the filtering is done by string comparison with the WordNet dictionary.

TIDIER dictionaries must be carefully validated as its results depend on them. Building and validating dictionaries is a non-trivial activity. We used two ways to validate a dictionary.

Manual validation for small dictionaries or highly-specific dictionaries, such as the abbreviations, and automatic filtering using a trusted reference dictionary, among others WordNet, for large dictionaries.

Typically, we created a dictionary for expanding identifiers as follows:

- First, we created a dictionary containing words from the English language (assuming that identifiers are in English) using already-available dictionaries, such as GNU i-spell or WordNet.
- Second, we built context-aware dictionaries by filtering WordNet/i-spell words that appear in a given source-code context. We used source code tokenization and pattern matching to automatically perform the filtering.
- Finally, we complemented the previously-built dictionaries with domain-specific words (not contained in the original dictionaries) and acronyms (together with their expansions), which is the most critical task.

Overall, TIDIER dictionaries requires one day to be produced, populated with abbreviations and acronyms typical for Unix utilities (*i.e.*, the domain of our empirical study). Documenting the C library functions required four to five days of manual verification.

5.1.3 Word Transformation Rules

Some identifier terms might not be part of the thesaurus and must be generated from existing words and, possibly, added to the thesaurus. Let us consider the identifier *fileLen* and suppose that the thesaurus contains the words *length*, *file*, *lender*, and *ladder*, and no abbreviations. Clearly, the word *file* matches with a zero string-edit distance with the first four characters of *fileLen*, while both *length* and *lender* have a distance of three from *len* because their last three characters could be deleted. The distance of *ladder* to *len* is higher than that of other words, because only *l* matches. Thus, both *length* and *lender* should be preferred over *ladder* to be associated with the term *len*. Clearly TIDIER performs at the character level and does not take in to account the software application semantic.

We defined and used the following transformation rules in TIDIER:

- *Delete a random character:* one randomly-chosen character is deleted from the word, *e.g.*, *pointer* becomes *poiner*;
- *Delete a random vowel:* one randomly-chosen vowel from the word is deleted, *e.g.*, *number* becomes *numbr*;

- *Delete all vowels:* all the vowels in a word are deleted, *e.g.*, *pointer* becomes *pntr*;
- *Delete suffix:* the suffix of the word, such as *ing*, *tion*, *ed*, *ment*, *able*, is deleted, *e.g.*, *improvement* becomes *improve*;
- *Keep the first n characters only:* the word is transformed by keeping the first n characters only, *e.g.*, *rectangle* becomes *rect*, with $n = 4$.

Constraints exist between transformation rules. For example, it is impossible to delete a random vowel once all vowels have been deleted; a suffix can be removed if and only if it is part of the word.

To choose the most suitable word to be transformed, TIDIER uses the following simple heuristic. It selects the closest words to the term to be matched, *i.e.*, the smallest non-zero distances, and repeatedly transforms these words using randomly-chosen transformation rules. This process continues until a transformed word matches the term or the transformed words reach a length shorter than or equal to three characters. We choose three characters as a lower limit because too many words would have the same abbreviation with two or less characters. If the transformed word matches the term, then this abbreviation is added in the thesaurus, else the algorithm tries to transform the next closest words to either find an abbreviation or report a failure to match the term with any word/abbreviation.

Putting It All Together

We now describe a typical run of TIDIER. First, wherever possible, identifiers are simply split using explicit separators, namely special characters, *e.g.*, "_", ".", "$", "->", and the CamelCase convention. Then, TIDIER applies transformations and computes the distance between the identifier terms and the thesaurus words by using a hill climbing search. For a given identifier and a given dictionary, the string-edit distance assigns a distance to each thesaurus word as well as the positions where it begins and ends in the identifier. In Fig. 5.3, we summarize the overall hill climbing procedure and its steps explained below.

The edit distance is the fitness function guiding the hill-climbing search as follows:

1. Based on the thesaurus, TIDIER (i) splits the identifier using the edit distance, (ii) computes the global minimum distance between the input identifier and all words in the thesaurus, (iii) associates a fitness value based on the distance computed in step (ii) to each thesaurus word. If the minimum global distance in step (ii) is zero, the process terminates successfully; else
2. From the thesaurus words with non-zero distance obtained at Step 1, TIDIER randomly selects one word having a minimum distance and:

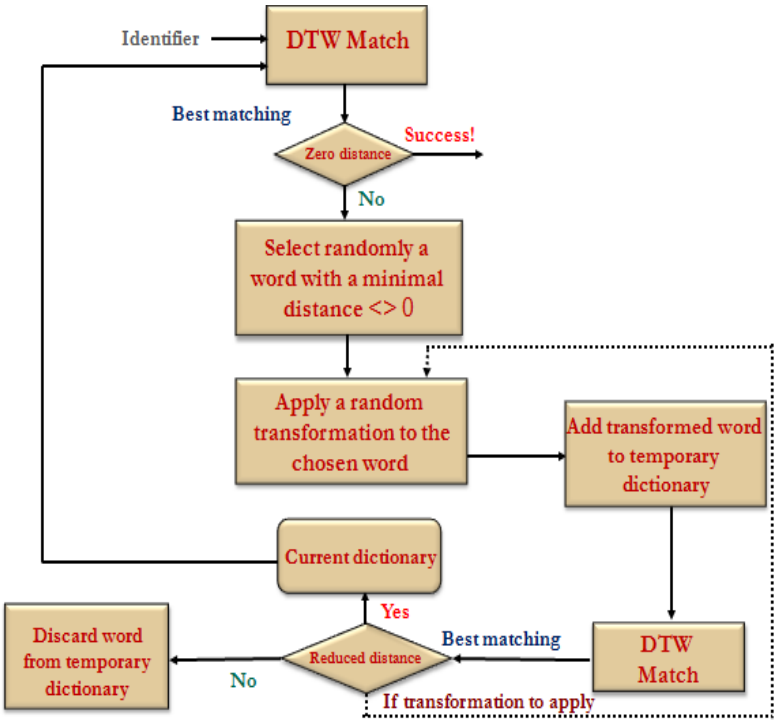


Figure 5.3 Overall Identifier Mappping (Hill Climbing) Procedure

(a) TIDIER randomly selects one transformation not violating transformation constraints, applies it to the word, and adds the transformed word to a temporary thesaurus;

(b) TIDIER splits the identifier using the temporary thesaurus and computes a new minimum global distance. If the added transformed word reduces the previous global distance, then TIDIER adds it to the current thesaurus and go to Step (a); else

(c) If there are still applicable transformations, and the string produced in Step (a) is longer than three characters, TIDIER goes to Step (a);

3. If the global distance is non-zero and the iteration limit was not reached, then, TIDIER goes back to Step 1, otherwise it terminates with a failure.

The above steps describe a hill-climbing algorithm, in which a transformed term is added to the thesaurus if and only if it reduces the global distance. Briefly, a hill-climbing algorithm (Michalewicz et Fogel, 2004) searches for a (near) optimal solution of a problem by moving from the current solution to a randomly chosen, nearby solution and accepts this solution only if it improves the global fitness. The algorithm terminates when there is no move to

nearby solutions improving the fitness. Differently from traditional hill-climbing algorithms, in Steps 1 and 2, TIDIER attempts to explore as many neighboring solutions as possible by performing word transformations. Different neighbors are explored depending on the order of the transformations.

Currently, the implementation of TIDIER uses a naive strategy to select a transformation. However, in our experience, even such a strategy performs well with small-to-medium size dictionaries (up to 5,000 words). For dictionaries larger than 20,000 words, the computation time to obtain meaningful results can become excessive. For example, with a dictionary of about 100,000 words and an upper computation limit of 20,000 attempts to improve distance, the computation can take up to 30 minutes or one hour depending on the input identifier.

5.2 TRIS

The cubic complexity of TIDIER makes it computationally demanding especially when the size of the used dictionaries increases as is the case for large software systems. To overcome this limitation, we suggested a fast solution, namely TRIS, which is inspired by TIDIER, but dealing differently with the vocabulary normalization problem. Similarly to TIDIER, TRIS assumes that programmers often build source code identifiers by applying a set of transformation rules to words, such as dropping all vowels (*e.g.*, *pointer* becomes *pntr*), dropping one or more characters, or dropping a suffix (*e.g.*, *allocation* becomes *alloc*) (Guerrouj *et al.*, 2013a). However, TRIS treats the identifier splitting and expansion as an optimization problem, in the following referred as Optimal Splitting-Expansion (OSE) problem. The search space of the OSE problem contains a set of solutions that represent potential splitting-expansions of the input identifier. Once a cost is assigned to each solution, the OSE problem consists in finding a solution with minimal cost, which, hopefully, corresponds to the correct splitting-expansion of the input identifier.

To efficiently resolve the OSE problem, TRIS applies a two-phase strategy. The first phase—named "building dictionary transformations"—builds a set of legal transformations based on an input dictionary using a family of transformation types. The obtained set of transformations is then compressed and represented as an arborescence *i.e.*, a directed rooted tree. The second phase is named "identifier processing". Its goal is to determine an optimal splitting-expansion of a given input identifier. Note that the second phase uses the directed rooted tree (*i.e.*, the arborescence) built during the first phase.

In practice, we generally wanted to find the splitting-expansion of a set of identifiers originating from the same source code—instead of a single one. It is very important to note that building the dictionary transformations (phase 1) has to be performed only once. Then,

identifier processing (phase 2) will be applied to each identifier to be split/expanded.

As we will show in Subsection 5.2.3, the identifier processing algorithm boils down to finding a shortest path in an identifier graph. Its complexity is $\mathcal{O}(n^2)$, where n represents the size of the input identifier, this is to say quadratic in the length of the identifier to split. As a result, producing the splitting-expansion of a given input identifier is very fast—thousands of identifiers can be split within a few seconds oppositely to TIDIER that requires hours in such case. On the other hand, the creation of dictionary transformations (which is performed only once) can take a few seconds (*e.g.*, 35 milliseconds for a dictionary of 2,953 words on a machine having a processor running at 2.10 GHz and memory (RAM) of 6.00 GB).

5.2.1 TRIS Formalization of the Optimal Splitting-Expansion Problem

The input of the OSE problem consists of: (i) an identifier to be split/expanded; (ii) a dictionary (iii) source code of the system that uses the identifier. In the following, we define the transformations, the search space, and the cost function of the OSE problem.

Transformations

Similarly to TIDIER, we used a set of transformation rules that we believe are the most used by software developers when creating identifiers. The current implementation of TRIS uses four types of transformation rules:

1. *Null transformation:* keep the word as it is;
2. *Vowels removal:* remove all vowels contained in the dictionary word (*e.g.*, *pointer* $\rightarrow$ *pntr*) but the first one if it is the first character of the identifier;
3. *Suffix removal:* suffixes such as *ing*, *tion*, *ed*, *ment*, and *able* are removed from the dictionary word (*e.g.*, *improvement* $\rightarrow$ *improve*);
4. *First n characters*: keeps only the first n characters of a word with n $\in \{3,4,5\}$, *e.g.*, *rectangle* $\rightarrow$ *rect* for $n = 4$.

In the following, we denote by type(.) the type of a given transformation.

Search Space

A potential solution (*i.e.*, an element of the search space) corresponds to a splitting-expansion. Such a solution is composed of a series of transformations. For example, a potential splitting-expansion of *drawimagrect* is (*draw*$\rightarrow$ *draw*)/(*image*$\rightarrow$ *imag*)/(*rectangle*$\rightarrow$ *rect*) as:

- the concatenation of *draw*, *imag*, and *rect* produces *draw.imag.rect=drawimagrect*;
- the words *draw*, *image*, and *rectangle* belong to a given dictionary;
- the transformations (*draw*→ *draw*), (*image*→ *imag*), and (*rectangle*→ *rect*) are legal transformations (whose types are respectively 1, 4, and 4).

Cost Function

Recall that a solution (splitting-expansion) is composed of a series of transformations. The cost of a solution is simply the sum of the costs of the transformations it is composed of. Each transformation has an associated cost C($wOrig \rightarrow w$) defined as the sum of two terms:

$$\text{C}(wOrig \rightarrow w) = \alpha * \text{Freq}(wOrig) + \text{C}(\text{type}(wOrig \rightarrow w))$$

The first term Freq($wOrig$) is the relative frequency of the dictionary word $wOrig$ in the source code, multiplied by a parameter α. The frequency is simply the number of occurrences of a dictionary word in the source code, divided by the sum of all occurrences of dictionary words in source code. TRIS uses the relative frequency as a local context to determine the most likely identifier splitting-expansion. We use the parameter α to favor transformations derived from original words having a high frequency. To minimize our cost function which is the sum of the two components explained above while favoring the local context (high frequencies), we multiply the frequency component (*i.e.*, Freq($wOrig$)) by a parameter α. The value of the parameter α is negative: as a result, a transformation ($wOrig \rightarrow w$) such that $wOrig$ has a low frequency will be in fact penalized.

The second component C(type($wOrig \rightarrow w$)) corresponds to the cost of the transformation type. The cost of the four different transformation types are algorithm parameters whose values will be reported in the empirical evaluation of TRIS (cf. Section 6.3 of Chapter 6); the general idea is to assign a low cost to a transformation type that is believed to be more natural and more often used by developers.

5.2.2 Building Dictionary Transformations Algorithm

The goal of this phase is to build the set of transformations and to represent it as an arborescence. It consists of the three following successive steps:
(1.1) Computation of the frequency of dictionary words;
(1.2) Construction of the set of transformations;
(1.3) Construction of the arborescence of transformations.
Each of these steps is detailed in the following. In this context, a dictionary is just a collection

of strings representing application-level concepts (*e.g.*, socket), known acronyms (*e.g.*, cpu), and plain English words (*e.g.*, a set of WordNet entries).

Computation of the frequency of dictionary words (step 1.1)

Input: (1) a dictionary; (2) code of the application.

Output: frequencies of dictionary words.

During this step, the source code is scanned. For each string found in the source code, if this string corresponds to a dictionary word, we increment the number of occurrences of this word. Finally, this procedure returns the frequency of each dictionary word.

Construction of the set of transformations (step 1.2)

Input: (1) a dictionary; (2) frequencies of dictionary words.

Output: set of transformation triples.

For each dictionary word $wOrig$ and each type T of transformation (T=1..4), we determine each word w that can be derived from $wOrig$ according to T. For each transformed word w, we add the triplet ($wOrig$, w, C($wOrig$, w)) to the set of transformations.

Construction of the arborescence (step 1.3)

Input: set of transformation triples.

Output: arborescence of transformations.

The goal of this step is to represent the set of transformations (built during step 1.2) under the form of an arborescence. The rationale is that, in the following, it will dramatically decreases the complexity of the construction of the auxiliary graph (step 2.1).

In this arborescence, each node (except the root) is labeled with a letter of the alphabet. Each transformation triple ($wOrig$, w, *cost*) is represented by a path that starts from the root and whose nodes are labeled by the letters of w. The last node X contains a pointer towards the considered transformation triple. In fact, as several transformations ($wOrig_1$, w_1, $cost_1$), ($wOrig_2$, w_2, $cost_2$), etc. may produce the same string w, we only take into account one of those whose cost is minimum. An interesting property of this arborescence is that, given a string w, we can determine in $\mathcal{O}(|w|)$ if there exists at least one transformation ($wOrig$, w, *cost*) and, if it is the case, which is the transformation of minimal cost. In Table 5.1, we provide a simplified example of a (small) dictionary (D) used to split/expand the identifier *callableint* along with dictionary words frequencies and the resulting transformation triples. As shown in Table 5.1, the dictionary D contains four words (*i.e.*, d_1=*"able"*, d_2=*"call"*, d_3=*"callable"*, and d_4=*"interface"*). We computed for each word its relative frequency in the source code, and then we applied the set of the four transformation rules on it. Thus, for each dictionary word, we had all possible (legal) transformations corresponding to it. For

example, after applying our four transformation rules to the dictionary word d_1=*"able"*, we got two legal transformations. The transformation t_1 resulting from the application of the fourth transformation rule (*i.e.*, keeping the three first characters) and the transformation t_2 resulting from the application of the first transformation rule (*i.e.*, null transformation). All the remaining transformation rules applied on d_1=*"able"*lead to transformed words having less than three characters. Since we did not consider transformed words of two characters and less, we only kept the transformations t_1 and t_2 in our set of transformations. The transformations t_1 and t_2 are triplets of the dictionary word d_1 = "*able*", the transformed words resulting from a transformation rule (*i.e.*, *abl* and *able*), and the cost computed according to the cost function we previously detailed. The same procedure was followed to generate the transformations corresponding to the other dictionary words.

The arborescence of the transformations corresponding to the dictionary D is shown in Fig. 5.4.

Let N be the sum of the sizes of w such that (*wOrig*, w, cost) belongs to the set of transformations. The arborescence construction algorithm complexity is $\mathcal{O}(N)$. Therefore, with respect to worst-case complexity, there is no extra-cost to transform the set of transformation triples into an arborescence.

5.2.3 Identifier Processing Algorithm

The goal of identifier processing is to determine an optimal splitting-expansion of a given input identifier *Idtf*.

It consists of the two following steps:
(2.1) Construction of the auxiliary graph associated to *Idtf*;
(2.2) Search for a shortest path in the auxiliary graph, corresponding to an optimal splitting-expansion of *Idtf*.

Construction of the auxiliary graph (step 2.1)
Input: (1) arborescence of transformations; (2) input identifier
Output: identifier auxiliary graph
Let *Idtf*`[i;j]` be the substring of *Idtf* between characters at position i and j. The auxiliary graph of *Idtf* is defined as follows:

- The graph has |Idtf|+1 vertices denoted by v_0,...,$v_{|\text{Idtf}|}$;
- For a transformation triple (*wOrig*, w, *cost*) such that w=Idtf`[i;j]`, there is an edge between the vertices v_i and v_j and the weight of this edge equals *cost*.

Table 5.1 Dictionary Transformations Building Information Example

Dictionary Transformations Building Phase Information: Identifier *callableint*		
Dictionary Words (D)	**Words Frequencies**	**Transformations Set**
d_1="*able*"	$f_1 = 0.1$	$t_1 = (d_1, abl, 0.55)$ $t_2 = (d_1, able, -0.2)$
d_2="*call*"	$f_2 = 0.2$	$t_3 = (d_2, cal, 0.35)$ $t_4 = (d_2, call, -0.4)$ $t_5 = (d_2, cll, 0.6)$
d_3="*callable*"	$f_3 = 0.6$	$t_6 = (d_3, calla, -0.95)$ $t_7 = (d_3, callable, -1.2)$ $t_8 = (d_3, cllbl, -0.2)$
d_4="*interface*"	$f_4 = 0.1$	$t_9 = (d_4, int, 0.55)$ $t_{10} = (d_4, inte, 0.3)$ $t_{11} = (d_4, inter, -0.05)$ $t_{12} = (d_4, interface, -0.2)$ $t_{13} = (d_4, intrfc, 0.8)$

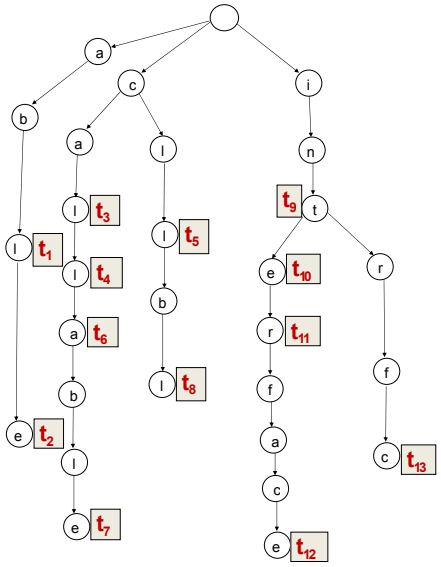


Figure 5.4 Arborescence of Transformations for the Dictionary D.

We can notice that a path in the auxiliary graph corresponds to a splitting-expansion of *Idtf*, and that the weight of this path corresponds to the cost of the corresponding splitting-expansion. Therefore, a shortest path in the graph corresponds to an optimal splitting-expansion.

The auxiliary graph is built as follows. For every position p in the identifier *Idtf*, $p = 0...|Idtf|$, we go from the root of the arborescence of transformations and down following the path labeled by *Idtf*`[p;n]` where $n = |Idtf|$. For each node X on this path, if $X.transfPtr$ is not null and points toward a transformation ($wOrig$, w, $cost$), we insert into the graph an edge between v_p and $v_{p+|w|}$ and assign to this edge a weight equals to $cost$. The complexity of this procedure is $\mathcal{O}(|n|^2)$, thus it is quadratic in the identifier length and as identifiers are usually short this step is very fast.

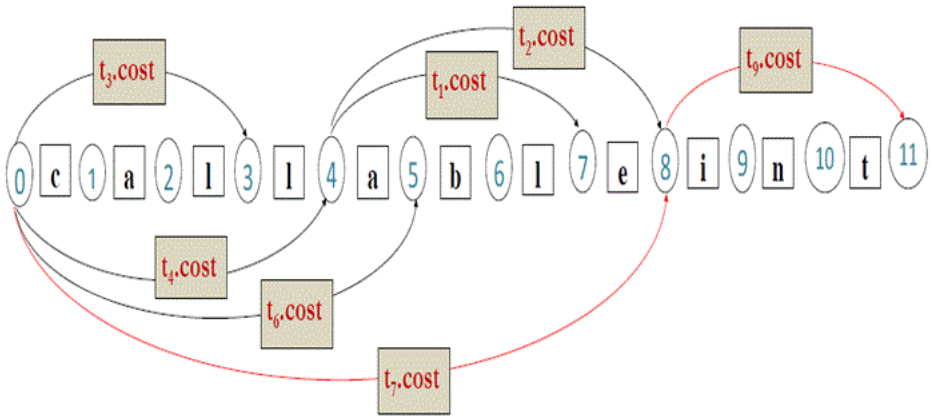


Figure 5.5 Auxiliary Graph for the Identifier *callableint*.

In Fig. 5.5, we show the auxiliary graph corresponding to the identifier *callableint*, built using the arborescence shown in Fig. 5.4. On this example, we have two possible splits (based on the set of transformations shown in Table 5.1). The first split is: *call-able-int*. The second is *callable-int*. Their corresponding expansions (pointing to their original dictionary words) are respectively *call-able-interface* (as *int* is derived from *interface* in the example), and *callable-interface*. According to the cost of transformations indicated in the last column of Table 5.1, denoted (for simplification of Fig. 5.5) by t_i.cost with i $\in$ {1,...,13} (computed based on words frequencies shown in the second column of the same table, plus costs of used transformation types), the minimum cost is the one corresponding to the split *callable-int* and hence to the expansion *callable-interface*.

Search for an optimal splitting-expansion (step 2.2)

Input: (1) *Idtf* auxiliary graph

Output: an optimal splitting-expansion of *Idtf*

The auxiliary graph is acyclic. Therefore, although some edges may have negative weights (remember the α multiplier), it makes sense to talk about a shortest path in this graph. The shortest path found in the auxiliary graph provides us with an optimal splitting-expansion of *Idtf*. The complexity of this procedure is at worst $\mathcal{O}(|n|^2)$, where $n = |Idtf|$.

CHAPTER 6

TIDIER and TRIS: Evaluation, Results, Discussion, and Threats to Validity

To evaluate TIDIER and TRIS normalization accuracy, we conduct two empirical studies. The first evaluates TIDIER on identifiers randomly-extracted from the source code of 340 C open-source projects while the second evaluates TRIS on this data set in addition to data sets used in previous studies and belonging to different programming languages (*i.e.*, Java, C, and C++).

We also compare TIDIER and TRIS, in terms of their correctness, precision, recall, and F-measure with previous approaches (*i.e.*, CamelCase, Samurai, or GenTest). For comparison reasons, we perform convenient statistical tests plus appropriate effect-size measures.

This chapter thoroughly describes TIDIER and TRIS empirical evaluations, it also shows their results and limitations. Then, it explains the main threats to validity related to our studies. Finally, the chapter summarizes our main findings and observations.

6.1 TIDIER Empirical Evaluation

The *goal* of this study is to analyze TIDIER with the *purpose* of evaluating its ability to adequately identify dictionary words composing identifiers, even in presence of abbreviations and–or acronyms. The *quality focus* is the precision and recall of the approach when identifying words composing identifiers with respect to a manually-built oracle and to alternative normalization approaches. The *perspective* is of researchers, who want to understand if TIDIER can be used as a means to assess the quality of source-code identifiers, *i.e.*, the extent to which they would refer to domain terms or, in general, to meaningful words, *e.g.*, words belonging to a dictionary.

The *context* consists of a set of 1,026 composed identifiers randomly-sampled from the source code of 337 GNU[1] projects, the Linux Kernel[2] 2.6.31.6, FreeBSD[3] 8.0.0, and the Apache Web server[4] 2.2.14. The GNU project was launched in 1984 with the ultimate goal to provide a free, open-source operating system and environment. GNU projects include well-known tools, such as the GCC compiler, parser generators, shells, editors, libraries, and textual utilities, just to name a few. Most code of the GNU project is written in C, with a

[1] `http://www.gnu.org/`
[2] `http://www.kernel.org/`
[3] `http://www.freebsd.org/`
[4] `http://www.apache.org/`

Table 6.1 Main characteristics of the 340 projects for the sampled identifiers.

GNU Projects (337 Projects)				
	C	C++	.h	Java
Files	57,268	13,445	39,257	14,811
Size (KLOCs)	25,442	2,846	6,062	3,414
Terms	26,824	–	17,563	–
Identifiers	1,154,280	–	619,652	–
Oracle Identifiers	927	–	26	–
Linux Kernel				
	C	C++	.h	Java
Files	12,581	–	11,166	–
Size (KLOCs)	8,474	–	1,994	–
Terms	19,512	–	13,006	–
Identifiers	845,335	–	352,850	–
Oracle Identifiers	73	–	4	–
FreeBSD				
	C	C++	.h	Java
Files	13,726	128	7,846	15
Size (KLOCs)	1,800	128	8,016	4
Terms	21,357	–	12,496	–
Identifiers	634,902	–	278,659	–
Oracle Identifiers	20	–	0	–
Apache Web Server				
	C	C++	.h	Java
Files	559	–	254	–
Size (KLOCs)	293	–	44	–
Terms	6,446	–	3,550	–
Identifiers	33,062	–	11,549	–
Oracle Identifiers	11	–	0	–

few C++ program (*e.g.*, *groff*). Linux is the well-known operating system widely adopted on servers and, in recent years, used as a desktop alternative to proprietary operating systems. The Linux Kernel is entirely written in C with additional utilities written mostly in scripting languages, such as Bash or TCL/TK. FreeBSD is another freely available operating system; as the name suggests it derives from the BSD branch of the Unix tree. The Apache Web server is a free and open-source Web server; it is adopted by public and private organizations for its robustness, speed, and security as well as its large community of developers. It is entirely developed in C. The main characteristics of these programs are listed in Table 6.1.

Table 6.2 Descriptive statistics of the contextual dictionaries.

Context	Min	1Q	Median	3Q	Max	Avg	σ
Application	29	900	1,797	3,028	22,190	2,320	2,374
File	1	40	79	175	4,088	131	148
Function	1	3	6	21	1,625	16	29

TIDIER aims at splitting identifiers by trying to match their terms with words contained in a thesaurus. We used the different kinds of dictionaries introduced in Subsection 5.1.2 (cf. Chapter 5). In Table 6.2, we report descriptive statistics of the context-aware dictionaries, built from all programs from which we sampled identifiers.

Research Questions

The study reported in this section addresses the following research questions:

1. **RQ1:** *How does TIDIER compare with alternative approaches, CamelCase splitting and Samurai, when C identifiers must be split?* This research question analyzes the performance of TIDIER and compares it with alternative approaches, a CamelCase splitter and an implementation of Samurai.

2. **RQ2:** *How sensitive are the performances of TIDIER to the use of contextual information and specialized knowledge in different dictionaries?* This research question analyzes the performances of TIDIER in function of different dictionaries.

3. **RQ3:** *What percentage of identifiers containing word abbreviations is TIDIER able to map to dictionary words?* This research question evaluates the ability of TIDIER to map identifier terms with dictionary words when these terms represent abbreviations of dictionary words.

6.1.1 Variable Selection and Study Design

The main independent variable of our study is the kind of splitting algorithm being used. There are three different values for this factor:

1. CamelCase

2. Samurai

3. TIDIER

The second independent variable is the used dictionary (or a set of dictionaries) among those defined in Subsection 5.1.2 (cf. Chapter 5). Thus, we have a number of possible treatments equal to the number of different dictionaries plus two, *i.e.*, the two alternative approaches: CamelCase and Samurai.

The first dependent variable considered in our study is the *correctness* of the splitting/mapping to dictionary words produced by TIDIER with respect to the oracle (cf. Subsection 2.8.1 of Chapter 2).

To provide a more insightful evaluation, we computed the precision and recall measures, plus F-measure, *i.e.*, the aggregated and overall measure of precision and recall (cf. Subsection 2.8.2 of Chapter 2).

6.1.2 Analysis Method

RQ1 and **RQ2** concern the comparison of the correctness, precision, recall, and F-measure of the different approaches and of variations of TIDIER when using different dictionaries. Thus, the analysis methods are the same for both research questions and their results are presented together in the next section.

We tested the differences among different approaches using the Fisher's exact test because correctness is a categorical measure. We tested the following null hypothesis H_0*: the proportion of correct splits,* p_1 *and* p_2*, between two approaches do not significantly change.*

To quantify the effect size of the difference between any two approaches, we computed the odds ratio (OR) (Sheskin, 2007).

Precision, recall, and F-measure are compared using the Wilcoxon paired test. We quantified the effect size of the difference using the Cohen d effect size for dependent variables.

As both the Fisher's exact test and the Wilcoxon paired test are executed multiple times to compare the various approaches and dictionaries, significant p-values was corrected using Holm correction (Holm, 1979).

Fisher's exact test, odds ratio, Wilcoxon paired test, Cohen d, and Holm correction are defined in Chapter 2 of this thesis.

For **RQ3**, we identified a set of abbreviations used in the sampled identifiers and computed the percentage of these abbreviations that were correctly mapped to dictionary words by TIDIER. We identified the set of likely abbreviations in our sample as follows:

1. For each identifier, *e.g.*, *counterPtr*, we considered the split performed using the CamelCase, *i.e.*, *counter ptr*, and the oracle, *i.e.*, *counter pointer*;

2. Then, we compared each term in the split with the term appearing in the same position in the oracle, *e.g.*, *counter* is compared with *counter* and *ptr* with *pointer*;

3. For all cases where (i) the term in the splitting did not match the one in the oracle, (ii) both terms started with the same letter, (iii) the term in the splitting did not appear in the English dictionary of 2,774 words, and (iv) the term in the oracle appeared in the English dictionary, we considered the term in the splitting as an abbreviation of the term in the oracle: *ptr* is an abbreviation of *pointer*.

The set of 73 abbreviations obtained with the above process has been manually validated to remove false positive. Then, we applied each approach, considering the English dictionary with domain knowledge, and count the percentage of abbreviations correctly mapped to dictionary words. We also computed the set of abbreviations that are not correctly mapped, but with a distance of one from the oracle, *i.e.*, the mapping failed for a single character

only. Thus, we identified and discussed cases where the approach almost found the correct solution, even though it failed to correctly split the identifier.

6.2 TIDIER Experimental Results

We now present and discuss the results of our study to answer the research questions formulated in Section 6.1. Raw data of our study are available on-line[5] for replication purposes. For what concerns the comparison with Samurai, we generated the local frequency table of each application by mining the source code terms frequencies in the application under analysis and we used as a global frequency list, a table generated by mining terms frequencies in the corpus of the 340 GNU analyzed projects.

RQ1 and RQ2

First, we evaluated the correctness of TIDIER when using different dictionaries and compare it with that of the two alternative approaches, *i.e.*, CamelCase and Samurai. We reported the percentages of correctly split/mapped identifiers in Figure 6.1.

The two bars at the bottom of the figure show the performances of the CamelCase splitter and Samurai, respectively, while the other bars show the performances of TIDIER using different dictionaries.

Table 6.3, we report results of the Fisher's exact test (with corrected p-values, significant p-values are shown in bold face) when performing a pair-wise comparison among approaches of the percentages of correctly split identifiers. The table also reports the ORs. ORs greater than one indicate results in favor of Approach 1 and vice versa.

Figure 6.1 and Table 6.3 show that:

- In the extracted sample, CamelCase performs nearly as well as Samurai and there are no statistically significant differences among them.
- When using only the simple English dictionary, TIDIER performs worse than CamelCase and Samurai. The percentage of correctly-split identifiers is only 23.82%, while CamelCase exhibits a performance of 30.08% and Samurai of 31.14%. The OR for TIDIER is 0.73 and 0.69 with respect to the two alternatives.
- When using a larger dictionary, *i.e.*, the WordNet dictionary, TIDIER does not perform significantly better (nor worse) than when using the simple English dictionary.
- When domain knowledge is added to the English dictionary, TIDIER significantly outperforms the alternative approaches. The percentage of correctly-split identifiers is nearly 40% with ORs of about 1.5 in favor of TIDIER wrt. CamelCase and Samurai.

[5]http://web.soccerlab.polymtl.ca/ser-repos/public/TIDIER-rawdata.tgz

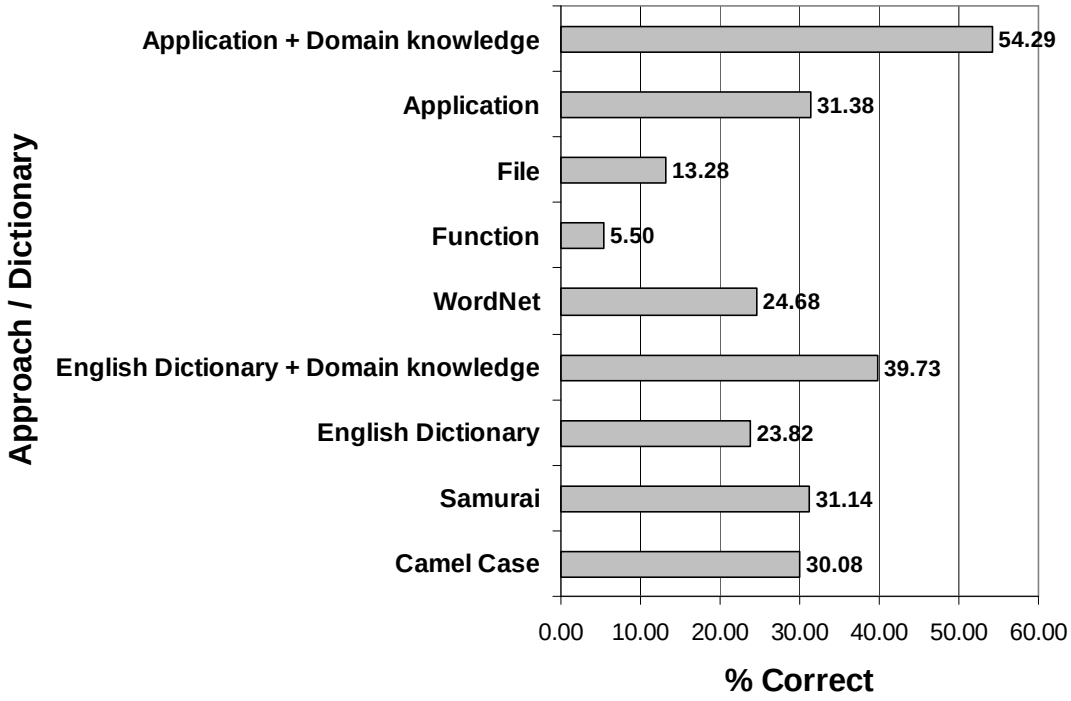


Figure 6.1 Percentages of correctly-split identifier.

Table 6.3 Comparison among approaches: results of Fisher's exact test and odds ratios.

Approach 1	Approach 2	p-values	ORs
CamelCase	Samurai	0.63	0.95
English dictionary	CamelCase	**0.01**	0.73
English dictionary	Samurai	**0.01**	0.69
English dictionary	WordNet	1.00	0.95
English dictionary + domain kn.	CamelCase	<**0.001**	1.53
English dictionary + domain kn.	Samurai	<**0.001**	1.46
English dictionary + domain kn.	English dictionary	<**0.001**	2.13
Application	CamelCase	1.00	1.06
Application	Samurai	1.00	1.01
Application	English dictionary + domain kn.	<**0.001**	0.69
Application	File	<**0.001**	2.98
Application	Function	<**0.001**	7.86
File	Function	<**0.001**	2.63
Application + Domain kn.	Application	<**0.001**	2.56
Application + Domain kn.	English dictionary	<**0.001**	3.80
Application + Domain kn.	English dictionary + domain kn.	<**0.001**	1.80
Application + Domain kn.	CamelCase	<**0.001**	2.76
Application + Domain kn.	Samurai	<**0.001**	2.62

- When using a contextual, program-level dictionary, TIDIER performs slightly (but not significantly) better (31.38%) than the alternative approaches but worse than when

using the English dictionary with domain knowledge. Contextual dictionaries at file or function levels do not seem particularly useful because of their limited size and, thus, the number of terms that they capture.

- When adding domain knowledge to the program-level dictionary, TIDIER shows its best performance, *i.e.*, 54.29% of correct splits. This percentage is significantly higher than those of the alternative approaches and than the one attained when using the English dictionary. *OR*s are 2.76 and 2.62 times in favor of TIDIER wrt. CamelCase and Samurai, respectively, and 1.80 wrt. using the English dictionary with domain knowledge.

Table 6.4 Descriptive statistics of F-measure.

Method	Dictionary	1Q	Median	3Q	Mean	σ
CamelCase		0.00	0.40	1.00	0.44	0.43
Samurai		0.00	0.50	1.00	0.49	0.42
TIDIER	English dictionary	0.00	0.29	0.67	0.38	0.41
	English dict. + domain kn.	0.29	0.67	1.00	0.60	0.39
	WordNet	0.00	0.40	0.80	0.43	0.40
	Function	0.00	0.00	0.00	0.13	0.27
	File	0.00	0.00	0.57	0.30	0.37
	Application	0.00	0.50	1.00	0.52	0.40
	Application + domain kn.	0.50	1.00	1.00	0.72	0.36

Table 6.5 Comparison among approaches: results of Wilcoxon paired test and Cohen d effect size.

Approach 1	Approach 2	p-value	ORs
CamelCase	Samurai	**<0.001**	−0.15
English dictionary	CamelCase	**<0.001**	−0.12
English dictionary	Samurai	**<0.001**	−0.19
English dictionary	WordNet	**<0.001**	−0.11
English dictionary + domain kn.	CamelCase	**<0.001**	0.29
English dictionary + domain kn.	Samurai	**<0.001**	0.22
English dictionary + domain kn.	English dictionary	**<0.001**	0.61
Application	CamelCase	**<0.001**	0.18
Application	Samurai	**0.01**	0.10
Application	English dictionary + domain kn.	**<0.001**	−0.16
Application	File	**<0.001**	0.46
Application	Function	**<0.001**	0.85
File	Function	**<0.001**	0.54
Application + Domain kn.	Application	**<0.001**	0.52
Application + Domain kn.	English dictionary	**<0.001**	0.81
Application + Domain kn.	English dictionary + domain kn.	**<0.001**	0.38
Application + Domain kn.	CamelCase	**<0.001**	0.58
Application + Domain kn.	Samurai	**<0.001**	−0.51

In Table 6.4, we show the descriptive statistics (first quartile, median, third quartile, mean, and standard deviation) of the F-measure. The aim is to evaluate the capability of the

approaches to correctly and completely identify the terms part of the identifiers. We do not show results of precision and recall separately (cf. Appendix A) because they are consistent with the F-measure, *i.e.*, there are no cases for which an approach exhibits a high precision and a low recall or vice versa.

In Table 6.5, we report corrected results of the paired Wilcoxon test and the Cohen d effect size (positive values of Cohen d are in favor of Approach 1, negative values are in favor of Approach 2). Overall, these results are consistent with those obtained when measuring correctness. They show that:

- TIDIER, with the English dictionary, performs significantly worse than the other approaches with a very small effect size, $d < 0.2$.
- When using the English dictionary with domain knowledge, TIDIER performs significantly better than CamelCase ($d = 0.29$) and Samurai ($d = 0.22$).
- When using the program-level dictionary, TIDIER performs significantly better than the alternative approaches, although the effect size is very small ($d < 0.2$).
- When using the program-level dictionary augmented with domain knowledge, TIDIER again performs significantly better than the alternative approaches, with a medium effect size ($d = 0.58$ for CamelCase and $d = 0.51$ for Samurai).

We can summarize the results for **RQ1** as follows: with the simple English dictionary, TIDIER performs worse than the alternative approaches. However, TIDIER outperforms other approaches when the simple English dictionary is augmented with domain knowledge or, with even better results, when it uses a program-level contextual dictionary augmented with domain knowledge.

Regarding **RQ2**, we concluded that there are two factors contributing to the increase of performance of TIDIER: augmenting the dictionary with domain knowledge, using a program-level contextual dictionary, or, to obtain the best performances, augmenting a program-level dictionary with domain knowledge.

RQ3

To answer **RQ3**, we ran TIDIER five times on the 73 abbreviations using the English dictionary of 2,774 words. Out of the 73 abbreviations that TIDIER could potentially map to dictionary words, TIDIER produced a correct mapping for 35 of them, achieving an accuracy of 48%.

Table 6.6 Examples of correct and wrong abbreviations.

Match with the Oracle			
Abbreviation	Oracle	Expansion 1	Expansion 2
arr	array	array	arrow
clr	clear	clear	color
curr	current	current –	
dev	device	device	–
div	division	dividend	divided
intern	internal	internal	–
len	length	length	lender
lng	long	long	language
mov	move	move	–
sec	security	security	secret
snd	sound	sound	sand
spec	specify	specify	specialize
str	string	string	strict
wrd	word	word	–
Wrong Expansions			
Abbreviation	Oracle	Expansion 1	Expansion 2
auth	authenticate	author	
comm	communication	comment	command
del	delete	deal	delay
dest	destination	destroy	
disp	display	dispatch	
exp	expresion	expansion	expire
mem	memory	membrane	memo
procs	process	protocol css	prototype css
vol	volume	voltage	voluntary
Distance > 0			
Abbreviation	Oracle	Expansion 1	Expansion 2
acct	accounting	act (1.0)	
addr	address	add (1.0)	
arch	architecture	march (1.0)	
elt	element	felt (1.0)	
lang	language	long (2.0)	
num	number	enum (1.0)	
paren	parenthesis	green (3.0)	

The first block of Table 6.6 shows examples of abbreviations that were correctly mapped by TIDIER to dictionary words. The table reports: (i) the abbreviations, (ii) the oracle, and (iii) the different mappings produced by TIDIER. The second block of Table 6.6 shows examples of wrong mappings, such as those of *auth* into *author* while the correct mapping was *authenticate*) or of *dest* into *destroy* while the correct expansion was *destination*. Wrong expansions were due to the fact that TIDIER does not use semantic information and, thus, can generate expansions that are different from those in our oracle even though with a zero distance. Consistently with insights gained from **RQ1** and **RQ2**, wrong expansions suggest that domain-specific dictionaries can be useful to better support source code vocabulary normalization.

Out of the $73 - 35 = 38$ abbreviations not correctly expanded by TIDIER, there are 16 identifiers wrongly expanded and 22 identifiers for which TIDIER was not able to produce an expansion with a zero distance. Some of these cases are shown in the third block of Table

6.6, where the numbers in parentheses report the achieved minimum distances. For example, *addr* was mapped to *add* instead of *address* with distance two (trailing *r* removed), *arch* into *march* instead of *architecture* (leading *m* added) with distance one, and *def* into *prefix* instead of *define* with distance two (leading *p* and *r* added).

In conclusion, **RQ3** suggested that TIDIER is, indeed, able to deal with the abbreviations used to build identifiers and can map them into dictionary words in 48% of the abbreviations considered in our sample.

6.3 TRIS Empirical Evaluation

The *goal* of this study is to analyze TRIS, with the *purpose* of evaluating its ability to correctly split and expand compound identifiers. The *quality focus* is the accuracy (*i.e.*, precision, recall, and F-measure) of TRIS when splitting identifiers and expanding abbreviated terms (resulting from the splitting) with respect to oracles, and compared with other state-of-the-art approaches, namely CamelCase, Samurai, TIDIER, and GenTest. The *perspective* is of researchers interested in developing an approach for identifier splitting and expansion, with the aim of easing program comprehension and maintenance tasks. The *context* consists of a set of identifiers extracted from Java, C and C++ programs. Specifically, we used (i) 974 identifiers extracted from the source code of JHotDraw, (ii) 3,085 identifiers from Lynx, (iii) 489 identifiers extracted from the source code of 340 C GNU Linux utilities, and (iii) a mixed set of Java, C, and C++ identifiers used in a study by Lawrie *et al.* (Lawrie et Binkley, 2011) and made available on-line[6]. We used the latter data for replication purposes as we wanted to compare TRIS (in terms of splitting accuracy) with GenTest[7].

JHotDraw[8] is a Java framework for drawing 2D graphics. The project started in October 2000 with the main purpose of illustrating the use of design patterns in a real context.

Lynx[9] is a free, open-source, text-only Web browser and Gopher client. Lynx is entirely written in C. Its development began in 1992 and it is now available on several platforms, including Linux, Unix, and Windows. In Table 6.7, we report the main characteristics of Lynx and JHotDraw analyzed releases.

The benchmark of 340 C/C++ programs from which we sampled 489 identifiers is the one we used to evaluate TIDIER and where the main characteristics are summarized in Table 6.1 of this chapter.

The sample of data used by Lawrie *et al.* was randomly drawn from a source base that

[6] *www.cs.loyola.edu/ binkley/ludiso*
[7] *http://splitit.cs.loyola.edu*
[8] *http://www.jhotdraw.org*
[9] *http://lynx.isc.org/*

Table 6.7 Main characteristics of JHotDraw and Lynx

JHotDraw and Lynx Systems		
	JHotDraw	Lynx
Analyzed Releases	5.1	2.8.5
Files	155	247
Size (KLOCs)	16	174
Identifiers (> 2 chars)	2,348	12,194

includes 186 programs, for a total of 26 MLOC of C, 15 MLOC of C++, and 7 MLOC of Java. Raw and oracle data sets are available on-line[10]. Details about the empirical evaluation can be found in a previous paper by Lawrie *et al.* (Lawrie et Binkley, 2011).

For what concerns the comparison with Samurai, we used the local and global frequency lists provided by the authors for Java systems (*i.e.*, JHotDraw) and already used in their evaluation to Samurai (Enslen *et al.*, 2009). Regarding C systems (*i.e.*, Lynx and the sample of 489 C identifiers), we generated the local frequency table of each application by mining the source code terms frequencies in the application under analysis and we used as a global frequency list, a table generated by mining terms frequencies in a large corpus of GNU projects.

As explained in Section 5.2, the costs assigned to the introduced transformation types (second component of our cost function) are algorithm parameters. In our empirical study, we assigned 0 as a cost to the *null transformation*, (0.75, 0.5, and 0.25) as costs to the three transformations *keeping the n first characters* with n $\in$ {3,4,5} respectively. For the transformations *removing vowels and suffix removal*, we respectively assigned 1 and 1.5 as costs. Also, we assigned to the parameter α, -2 as a value. To determine the values of the parameters, we run TRIS multiple times with different transformations' costs and alpha values.

The study reported in this section aims at addressing the following research question:

RQ: *What is the accuracy of the TRIS identifier splitting and expansion approach compared with alternative state-of-the art approaches?*

To address this research question, we measured the performance of TRIS in terms of correctness, precision, recall, and F-measure of the identifier splits and expansions provided wrt. to the oracles. In addition, we compared its performance to the one shown by CamelCase, Samurai, TIDIER, and GenTest.

[10] *http://www.cs.loyola.edu/ binkley/ludiso/*

6.3.1 Variable Selection and Study Design

The main independent variable of our study is the splitting algorithm used. This factor has five possible levels: (1) TRIS (which is our experimental group), and four control groups, *i.e.*, (2) CamelCase, (3) Samurai, (4) TIDIER, (5) GenTest.

The dependent variables considered in this study are those we considered when evaluating TIDIER, *i.e.*, precision, recall, and F-measure, and the identifier splitting/expansion correctness (cf. Chapter 2).

In this work, we generally measured and compared the performance of the various splitting/expansion algorithms in terms of precision, recall, and F-measure. The only case in which we used the identifier splitting correctness is when comparing TRIS with GenTest on the data and oracle sets used by Lawrie *et al.*, for which we only had identifier splitting correctness data (Lawrie et Binkley, 2011).

6.3.2 Analysis Method

Our research question aim is to understand if TRIS helps in splitting and expanding identifiers, thus, easing program comprehension and supporting IR-based software maintenance tasks. Similarly to TIDIER, we assumed that, given an identifier, there exists an exact subdivision of it into words that, possibly after being transformed and once concatenated, form the identifier.

To apply TRIS, we built an application-level dictionary for each program part of our study, *i.e.*, for JHotDraw, Lynx, and for each one of the 340 programs from which we sampled the C identifiers. In addition, we enriched these dictionaries by the use of domain knowledge (*i.e.*, common abbreviations and acronyms, library functions, etc) as TIDIER results showed that a dictionary containing application-level terms, English dictionary words, and common abbreviations and acronyms, allows one to obtain the best performances. Details about the construction of application-level dictionaries and the used domain knowledge are provided in Chapter 5 of this thesis (cf. Subsection 5.1.2). More precisely we used: (i) a set of 105 acronyms used in computer science (*e.g.*, ansi, dom, inode, ssl, url), (ii) a set of 164 abbreviations collected among the authors used when programming in C (*e.g.*, bool for Boolean, buff for buffer, wrd for word), and (iii) a set of 492 C library functions (*e.g.*, malloc, printf, waitpid, access). The application-level dictionaries for JHotDraw and Lynx contain 2,289 and 2,953 dictionary words respectively, while descriptive statistics about the size of application-level dictionaries for the 340 GNU utilities are reported in Table 6.8.

We filtered identifiers containing short (up to two letters) prefixes such as *f* in *fname* or *ly* in *lynx*. This is because such prefixes can lead to any dictionary word containing the

Table 6.8 Descriptive statistics of the used program-level dictionaries for the 340 GNU utilities

Dictionary level	Min	1Q	Median	3Q	Max	Avg	σ
Application	29	900	1,797	3,028	22,190	2,320	2,374

character *f* or the string *ly*. We needed such a filtering in a very few cases (less than 1% of the identifiers for the Lynx system in general).

To compare TRIS with other algorithms (except GenTest), we used the Wilcoxon paired test. In addition, we computed the effect-size of the difference using Cliff's delta (Grissom et Kim, 2005).

Since we executed the Wilcoxon paired test multiple times to compare the various approaches, we must correct significant p-values. We adjusted the obtained p-values using the Holm correction (Holm, 1979).

For what concerns the comparison with GenTest, since the comparison is performed in terms of correctness (which is a categorical variable), we used Fisher's exact test (Sheskin, 2007) which compares proportion of correct and non correct splittings provided by TRIS and GenTest. To quantify the effect size of the difference between the two approaches, we also computed the odds ratio (OR) (Sheskin, 2007) indicating the ratio of the percentage of identifiers correctly split by TRIS (experimental group) and the percentage of identifiers correctly split by GenTest (control group).

All the tests, effect-size measures, and p-values corrections used in this study are defined in Chapter 2.

6.4 TRIS Experimental Results

This section reports the results of the empirical study. In Table 6.9, we report descriptive statistics (1^{st} quartile, median, 3^{rd} quartile, mean, standard deviation) of the accuracy of TRIS and the ones of CamelCase, Samurai, and TIDIER. Results of the statistical tests for JHotDraw are reported in Table 6.10. Similarly, descriptive statistics and statistical test results for Lynx are reported in Tables 6.11 and 6.12 respectively.

Results indicated that, for JHotDraw, TRIS achieved 93.28% of F-measure while CamelCase and Samurai attained 92.17% and 93.25% of F-measure respectively, and TIDIER exhibited an F-measure of 92.33%. Not surprisingly, CamelCase and Samurai worked well enough on JHotDraw, because JHotDraw developers carefully adhered to coding standards and identifier creation rules. Also, TIDIER performs almost similarly to them, even if its approach does not necessarily reward the use of coding standards as for instance CamelCase does. Statistical comparisons reported in Table 6.10 show that (i) there is no significant difference between TRIS, CamelCase, and Samurai on JHotDraw; and (ii) TRIS performs significantly

Table 6.9 Precision, Recall, and F-measure of TRIS, CamelCase, Samurai, and TIDIER on JHotDraw

Metric	Approach	1Q	Median	Mean	3Q	σ
Precision	CamelCase	1.0000	1.0000	0.9244	1.0000	0.2424
	Samurai	1.0000	1.0000	0.9316	1.0000	0.2244
	TIDIER	1.0000	1.0000	0.9716	1.0000	0.1472
	TRIS	1.0000	1.0000	0.9804	1.0000	0.2025
Recall	CamelCase	1.0000	1.0000	0.9203	1.0000	0.2502
	Samurai	1.0000	1.0000	0.9367	1.0000	0.2129
	TIDIER	1.0000	1.0000	0.8984	1.0000	0.2158
	TRIS	1.0000	1.0000	0.9084	1.0000	0.1213
F-measure	CamelCase	1.0000	1.0000	0.9217	1.0000	0.2476
	Samurai	1.0000	1.0000	0.9325	1.0000	0.2200
	TIDIER	1.0000	1.0000	0.9233	1.0000	0.1791
	TRIS	1.0000	1.0000	0.9328	1.0000	0.1614

Table 6.10 Comparison among approaches: results of Wilcoxon paired test and Cliff's delta effect size on JHotDraw.

Approach 1	Approach 2	adj p-value	Cliff's delta
TRIS	CamelCase	0.431	0.041
TRIS	Samurai	0.894	0.001
TRIS	TIDIER	**0.024**	0.043

Table 6.11 Precision, Recall, and F-measure of TRIS, CamelCase, Samurai, and TIDIER on Lynx.

Metric	Approach	1Q	Median	Mean	3Q	σ
Precision	CamelCase	0.0000	0.5000	0.4065	0.7500	0.4147
	Samurai	0.0000	0.5000	0.4767	1.0000	0.4089
	TIDIER	0.8000	1.0000	0.8609	1.0000	0.2674
	TRIS	1.0000	1.0000	0.9344	1.0000	0.1369
Recall	CamelCase	0.0000	0.3333	0.3705	0.6667	0.4066
	Samurai	0.0000	0.3333	0.4569	1.0000	0.4101
	TIDIER	0.7500	1.0000	0.8499	1.0000	0.2684
	TRIS	1.0000	1.0000	0.9138	1.0000	0.2060
F-measure	CamelCase	0.0000	0.4000	0.3851	0.7273	0.4086
	Samurai	0.0000	0.4000	0.4634	1.0000	0.4084
	TIDIER	0.6667	1.0000	0.8525	1.0000	0.2664
	TRIS	1.0000	1.0000	0.9206	1.0000	0.2055

Table 6.12 Comparison among approaches: results of Wilcoxon paired test and Cliff's delta effect size on Lynx.

Approach 1	Approach 2	adj p-value	Cliff's delta
TRIS	CamelCase	**<0.001**	0.743
TRIS	Samurai	**<0.001**	0.684
TRIS	TIDIER	**<0.001**	0.204

better than TIDIER with a very small effect size, $d < 0.148$. On Lynx, in terms of F-measure, TRIS significantly outperforms (92.06%) CamelCase (38.51%), Samurai (46.34%), and TIDIER (85.25%). More precisely, the statistical comparisons shown in Table 6.12 indicate that, on Lynx (i) TRIS significantly outperforms the CamelCase splitter ($d = 0.743$) and Samurai ($d = 0.684$), and (ii) TRIS performs significantly better than TIDIER with a small effect size ($d = 0.204$).

Table 6.13 Precision, Recall, and F-measure of TRIS and TIDIER on the 489 C sampled identifiers.

Metric	Approach	1Q	Median	Mean	3Q	σ
Precision	TIDIER	0.4000	0.6667	0.6368	1.000	0.3681
	TRIS	1.0000	1.0000	0.8933	1.0000	0.2471
Recall	TIDIER	0.5000	0.6667	0.6496	1.000	0.3654
	TRIS	1.0000	1.0000	0.872	1.0000	0.2606
F-measure	TIDIER	0.4000	0.6667	0.6409	1.0000	0.3650
	TRIS	1.0000	1.0000	0.879	1.0000	0.2524

Table 6.13 reports the performance of TRIS and TIDIER on the sample of 489 C identifiers. On such data set, we did not report performances of CamelCase and Samurai, since it is known from (Guerrouj *et al.*, 2013a) that TIDIER outperforms CamelCase and Samurai on C systems when using application-level dictionaries augmented with domain knowledge. Hence, we were only interested in comparing TRIS with the approach performing better on this data set *i.e.*, TIDIER. Results showed that, in terms of F-measure, TRIS performs better (87.9%) than TIDIER (64.09%) for this set also. The statistical comparison through Wilcoxon test indicated that the difference is statistically significant (p-value < 0.001), and that the Cliff's delta effect size is medium ($d = 0.456$).

In Table 6.14, we report the results of TRIS, in terms of precision, recall, and F-measure, on the data set from Lawrie *et al.* (Lawrie et Binkley, 2011). As it can be noticed, performances are very high, with a median of 100% and a mean precision of 98%, recall of 94% and F-measure of 96%.

Table 6.14 Precision, Recall, and F-measure of TRIS on the Data Set from Lawrie *et al.*.

Metric	Approach	1Q	Median	Mean	3Q	σ
Precision	TRIS	1.0000	1.0000	0.9763	1.0000	0.1184
Recall	TRIS	1.0000	1.0000	0.9439	1.0000	0.1565
F-measure	TRIS	1.0000	1.0000	0.9559	1.0000	0.1358

In Table 6.15, we report the accuracy of TRIS in terms of percentage of correct splittings, compared with the performances of GenTest and Samurai as reported by Lawrie *et al.* (Lawrie

et Binkley, 2011). As it can be noticed, TRIS correctly splits identifiers in 86% of the cases, while GenTest does it in 82% of the cases, and Samurai in 70% of the cases.

Table 6.15 Correctness of the splitting provided using the Data Set from Lawrie *et al.*.

Approach	Identifier Splitting Correctness
Samurai	70%
GenTest	82%
TRIS	86%

When comparing the correctness of TRIS with the one of GenTest, Fisher's exact test did not indicate a significant difference (p-value=0.5), even though the achieved correctness is higher for TRIS. We believe the comparison would be insightful if precision, recall or F-measure were provided because splitting correctness, in our case, is a Boolean variable that returns (true) if the split is correct and (false) if not. Thus, when the splitting is almost correct, *i.e.*, most of the terms are correctly identified, the correctness would still be false. Unfortunately, this was the case for identifiers such as the ones prefixed with letters (*e.g.*, *mEnvironmentalistNb*, *sOS_DriveDirectory*, *xGetJobStaus*, *xgetAutomaticFocus*, *xgetColumnWidth*, etc.) and that we filtered as the letters can be generated by any dictionary word prefixed with them. Also, even though the difference in the strict correctness measure is not high (86%) against (82%) for GenTest, the F-measure of our approach attains 96%. The latter measure clearly shows that the novel approach performs well on the overall data of Lawrie *et al.*.

6.5 TIDIER and TRIS Discussion

TIDIER uses techniques inspired from dynamic programming and string-edit distance to split and expand identifiers into meaningful words.

Although a distance-based identifier splitting and expansion approach is promising, it does not consider, *per se*, semantics. For example, with *fileLen*, *length* should be preferred over *lender*. However, the string-edit distance cannot be used to choose between *lender* or *length*. In addition, it is not possible to disambiguate complex identifiers that actually have an optimal non-zero distance splitting/expansion, because the algorithm always favors zero-distance splitting/expansion. For example, *imagEdges* contains the words *image* and *edges*. However, *image* and *edges* match the identifier with a distance of one because character *E* is shared by both terms in the identifier. Clearly, in this example, developers would use syntax and semantics as well as contextual and specialized knowledge: even if *imag* is not an English word, they would correctly split *imagEdges* into *image* and *edges*.

Finally, the string-edit distance used by TIDIER has a cubic complexity in the number of characters in the identifier (say M), words in the dictionary (say T), and maximum number of characters composing dictionary words (say N). For each word in the dictionary, we must compute as many distances as there are cells to fill in the distance matrix, with a complexity of $\mathcal{O}(M \times N)$. Because there are T dictionary words, the overall complexity is $\mathcal{O}(T \times M \times N)$. The latter limitation makes TIDIER computationally demanding in comparison with CamelCase and Samurai. Indeed, TIDIER has a cubic distance evaluation cost plus the search time, while its prior works, *i.e.*, CamelCase and Samurai have linear and quadratic complexities respectively. Also, the performance of TIDIER is highly dependent on the dictionary size and quality. In an extreme case, if each identifier is composed of dictionary words and split/expanded with an exact match, the complexity of TIDIER would be quadratic. TIDIER, even with the largest dictionary among those considered, took a few hours to split and expand the 1,026 identifiers of our study. Clearly, if hundreds of thousands of identifiers must be processed, the current implementation of TIDIER is not suitable and heuristics must be used to reduce computation time. For example, identifiers consisting of single words contained in the dictionary and neither composed of multiple words nor containing abbreviations, could be filtered in linear time.

TRIS is accurate and fast, its wrong splittings were mainly due to identifiers containing acronyms or short abbreviations. For example, we believe that it is impossible to correctly split and expand acronyms such as *afaik* or *imho*. We also believe that even if we consider the context (*i.e.*, the frequency of dictionary words in the source code) in TRIS, it is impossible to find the exact expansion of identifiers prefixed with letters such as *f* in the identifier *fsize* (appearing in JHotDraw) because the mapping could vary from *file size* to *figure size* depending on the JHotDraw code region where *fsize* appears.

Overall, the results showed that the novel approach performs more accurately than previous ones on the overall studied systems. In addition to splitting and expansion performances, TRIS has the advantage of performing reasonably fast: it takes 0.049 seconds to compile the JHotDraw dictionary (of 2,289 words) and 3.709 seconds to split/expand the 974 JHotDraw identifiers, while it takes 0.053 seconds to compile the Lynx dictionary (of 2,953 words) and 16.940 seconds to process the 3085 Lynx identifiers. In fact, TIDIER computation time increases with the increase of the dictionary size due to its cubic distance evaluation cost plus the search time. CamelCase and Samurai performs fast their computations. Yet, they are not accurate when naming conventions are not used. With the above timing performance, TRIS showed an improvement of 4% (not statistically significant) in terms of identifier splitting correctness over GenTest.

In summary, we can conclude that for Java programs properly following coding standards,

a simple CamelCase is enough. For C and C++ programs, TRIS outperforms CamelCase, Samurai and TIDIER. Also, TRIS performs slightly better than GenTest in terms of identifier splitting correctness, although the difference is not statistically significant.

6.6 TIDIER and TRIS Threats to Validity

Threats to **construct validity** are mainly due to mistakes in the oracle. We cannot guarantee that no errors are present in the oracle. As the intent of the oracle is to explain identifiers semantics, we cannot exclude that some identifiers could have been split in different ways by the developers that originally created them. This problem is related to guessing the developers' intent and we can only hope that, given the program domain, the class, file, method, or function containing the identifiers (and the general information that can be extracted from the source code and documentation), it will be possible to infer the developers' *likely* intent. To limit this threat, different sources of information, such as comments, context, and online documentation were used when producing the oracle. Another threat could be the fact that a given string can be derived from several dictionary words, *e.g.*, the string *imag* can be derived from *image* and *imagination* by applying word transformations. We mitigated such a threat by considering the identifier context, *i.e.*, function, file or application in the case of TIDIER and the frequency of source code strings in the case of TRIS.

Threats to **internal validity** are due to the subjectivity in the manual building of the oracle and to the possible biases introduced by manually splitting identifiers. To limit this threat, the oracle was produced following the consensus approach we previously explained (cf. Chapter 2), *i.e.*, the oracle was created by two of the authors independently and inconsistencies in splitting/expanding identifier terms to dictionary words were discussed.

Threats to **Conclusion validity** concern the relations between the treatment and the outcome. Proper tests were performed to statistically reject the null hypotheses. In particular, we used non-parametric tests, which do not make any assumption on the underlying distributions of the data, and, specifically, a test appropriate for categorical data (the Fisher's exact test) and one for paired, ranked data (the Wilcoxon paired test). Also, we based our conclusions not only on the presence of significant differences but also on the presence of a practically relevant difference, estimated by means of an effect-size measure. Last, but not least, we dealt with problems related to performing multiple Fisher and Wilcoxon tests using the Holm's correction procedure.

Threats to **external validity** concern the possibility of generalizing our results. To make our results as generalizable as possible, we evaluated TIDIER on a sample of identifiers that we extracted from a very large set of open-source projects. The size of our sample (1,026

composed identifiers) is comparable to the one used by Enslen *et al.* in their work (Enslen *et al.*, 2009).

To make our results as generalizable as possible, we analyzed C open-source projects in TIDIER, and Java, C, and C++ projects in TRIS. We believe the number of the analyzed systems is sufficient enough to generalize our results. However, we cannot be sure that our findings will be valid for other domains, applications, or programming languages.

6.7 Chapter Summary

We proposed two source code vocabulary normalization approaches, namely, TIDIER and TRIS. TIDIER is inspired by speech recognition, it uses Dynamic Time Warping, a string-edit distance, and a hill-climbing search technique. TIDIER results show that, with program-level dictionaries augmented with domain knowledge, *i.e.*, common acronyms, abbreviations, and C library functions, TIDIER significantly outperforms previous approaches. Specifically, TIDIER achieved with the program-level dictionary complemented with domain knowledge 54% of correct splits, compared to 30% for CamelCase and 31% for Samurai. Moreover, TIDIER was also able to map identifiers terms to dictionary words with a precision of 48% for a set of 73 abbreviations. The only two main limitations of TIDIER are its pure lexical-level matching and cubic complexity.

TRIS is a two-phases approach that we suggested as a fast solution for vocabulary normalization, it deals with the identifier splitting and expansion problem as a graph optimization (minimization) problem to find the optimal path (*i.e.*, the optimal splitting-expansion) in an acyclic weighted identifier graph. TRIS has been applied on several Java, C, and C++ systems, and compared to four techniques, *i.e.*, CamelCase, Samurai (Enslen *et al.*, 2009), TIDIER (Guerrouj *et al.*, 2013a), and GenTest (Lawrie et Binkley, 2011). TRIS results indicated that while for Java systems following appropriate naming conventions—such as JHotDraw—simple splitting approaches such as CamelCase are just enough, on C systems, TRIS significantly outperformed CamelCase, Samurai, and TIDIER with a medium to large effect size. In addition, TRIS performs slightly better than GenTest in terms of identifier splitting correctness, it shows a small improvement, not statistically significant, of 4% on a data set from Lawrie *et al.* (Lawrie et Binkley, 2011) consisting of Java, C, and C++ identifiers. TRIS uses a tree-based representation that makes it—in addition to being more accurate than other approaches—efficient in terms of computation time. Thus, TRIS produced one optimal split and expansion fast using an identifier processing algorithm having a quadratic complexity in the length of the identifier to normalize.

CHAPTER 7

Impact of Identifier Splitting on Feature Location

Source code identifier splitting and expansion is one of the essential ingredients in any feature location or traceability recovery technique (Antoniol *et al.*, 2002; Marcus *et al.*, 2005; Liu *et al.*, 2007; Poshyvanyk *et al.*, 2007; Eaddy *et al.*, 2008a; Revelle *et al.*, 2010). ,

Several identifier splitting/expansion approaches have been suggested (Enslen *et al.*, 2009; Lawrie et Binkley, 2011; Guerrouj *et al.*, 2013a) as a solution for the vocabulary mismatch problem that exists in IR between the natural language found in source code and that used in other software artifacts, making source code vocabulary more appropriate for use with IR-based tools. To date there has been little empirical evidence on the impact of identifier splitting/expansion on IR-based techniques.

In this chapter, we describe our empirical study on the impact of vocabulary normalization on feature location. Specifically, we investigate the effect of three identifier splitting techniques: CamelCase, Samurai and manually built splitting (*i.e.*, Oracle) on two FLTs for locating bugs and features. The first FLT is based on IR while the second uses both IR and dynamic information (IRDyn).

7.1 Empirical Study Design

The *goal* of this study is to compare accuracy of two FLTs (*i.e.*, IR and IRDyn), when utilizing three identifier splitting algorithms: CamelCase, Samurai and Oracle (*i.e.*, manual splitting of identifiers). This study is done from the *perspective* of researchers who want to understand if existing approaches for splitting identifiers can improve accuracy of FLTs under different scenarios and settings, including best possible scenario where splitting is done by experts. In addition, we are interested to know if an advanced splitting algorithm would be still useful for enhancing the accuracy of feature location when execution information is used. The *context* consists of two Java applications: Rhino and jEdit, their main characteristics are described in Subsection 8.1.3.

7.1.1 Variable Selection and Study Design

The main independent variable of our study is the type of splitting algorithm used: CamelCase, Samurai or Oracle (*i.e.*, manually split identifiers). The second independent variable is the use of dynamic information. Thus, we have two FLTs, and each has three configurations,

which depend on the identifier splitting technique (cf. Table 7.1). For example, $\text{IR}_{CamelCase}$, $\text{IR}_{Samurai}$, and IR_{Oracle} are the IR-based FLTs that use LSI to compute similarities between queries and methods, after applying the CamelCase, Samurai and Oracle splitting algorithms on the identifiers from the methods and queries. Similarly, $\text{IR}_{CamelCase}$Dyn, $\text{IR}_{Samurai}$Dyn and IR_{Oracle}Dyn are the FLTs that use IR and dynamic information after applying the latter splitting algorithms respectively. In order to compare which configuration of the FLTs is more accurate than another (*i.e.*, $\text{IR}_{CamelCase}$ vs. $\text{IR}_{Samurai}$), we considered their effectiveness measure (Liu *et al.*, 2007).

Table 7.1 The configurations of the two FLTs (*i.e.*, IR and IRDyn) based on the splitting algorithm.

Splitting Algorithm	IR FLT	IRDyn FLT
CamelCase	$\text{IR}_{CamelCase}$	$\text{IR}_{CamelCase}$Dyn
Samurai	$\text{IR}_{Samurai}$	$\text{IR}_{Samurai}$Dyn
Oracle (Manual Split)	IR_{Oracle}	IR_{Oracle}Dyn

As indicated in Chapter 2, the effectiveness measure is the best rank (*i.e.*, lowest rank) among all the methods from the gold set for a specific feature. Intuitively, the effectiveness measure quantifies the number of methods a developer has to examine from a list of ranked methods returned by the feature location technique, before she is able to locate a relevant method pertaining to the feature. Obviously, a technique that consistently places relevant methods towards the top of the ranked list (*i.e.*, lower ranks) is more effective than a technique that contains relevant methods towards the middle or the bottom of the ranked list (*i.e.*, higher ranks). In this analysis, we focused on the scenario of finding just one relevant method, as opposed to finding all relevant methods from the gold set. The latter decision was made for two reasons. First, our focus was on concept location, rather than impact analysis. Second, once a relevant is found, it becomes easier to find other related methods by following program dependencies from the relevant method, or by using other heuristics.

In literature, the identifiers that are split using CamelCase are referred as "hard-words", whereas the identifiers split using Samurai or TIDIER are called "soft-words" (cf. Chapter 2). During our analysis, we treated the hard and soft words in the same way and we refereed to them as split identifiers.

The dependent variable considered in our study is the effectiveness measure of the FLTs.

We aimed at answering the following overarching question: *if we had a perfect technique for splitting identifiers, would it still help improve accuracy of FLTs?* We answered this question by examining these more specific research questions (RQs):

1. **RQ1**: Does $IR_{Samurai}$ outperform $IR_{CamelCase}$ in terms of effectiveness?

2. **RQ2**: Does $IR_{Samurai}$Dyn outperform $IR_{CamelCase}$Dyn in terms of effectiveness?

3. **RQ3**: Does IR_{Oracle} outperform $IR_{CamelCase}$ in terms of effectiveness?

4. **RQ4**: Does IR_{Oracle}Dyn outperform $IR_{CamelCase}$Dyn in terms of effectiveness?

Previous work (Guerrouj *et al.*, 2013a) compared the CamelCase, Samurai and TIDIER splitting algorithms in terms of their accuracy for correctly splitting identifiers. However, in this study we were addressing the impact that splitting algorithms have on feature location.

7.1.2 Simplifying Oracle - "Perfect Splitter"- Building

In this study, we tried to simplify the oracle building process explained in Chapter 2. The reason is that manual verification and split can be a tedious and error prone task due to the huge number of words contained in application dictionaries, collected identifiers and terms from comments. We simplified this phase by applying a multi-step strategy aiming at minimizing the manual effort. In the following subsections we report details of each step.

Step one - building software application dictionary

We parsed and extracted identifiers and comments from both Rhino and jEdit and created a dictionary for each system. During this step we also built an application specific identifier (or term) frequency table for Samurai. Following this preliminary step, we filtered some dictionary entries to reduce manual validation effort.

Step two - filtering concordant identifier split

For each dictionary entry we ran the CamelCase, Samurai and TIDIER to locate the identifiers for which these three splitting algorithms were in agreement. TIDIER was configured with WordNet[1] dictionary, as well as with acronyms and abbreviations known to the authors. We used the Samurai global frequency table made available by Samurai authors (Enslen *et al.*, 2009), as well as a local frequency table estimated from the software application under analysis (see Step 1). Whenever the three splitting algorithms agreed on the identifier term subdivision, we considered this as a strong indication that the resulting split was actually correct. This assumption divided the dictionary into two sub-dictionaries: one on which the algorithms disagree and one where there is agreement among them. The sub-dictionary where the tools agreed was then manually inspected to make sure that no errors were present. For example, out of about 6,000 dictionary entries (or words) for Rhino, about 2,500 words were split in this phase with a minimum manual effort.

[1] `http://wordnet.princeton.edu`

Step three - filtering discordant identifier split

We manually inspected the identifiers for which the three splitting algorithms did not agree, in order to provide the best splitting. Examples of identifiers from the Rhino dictionary are words such as *DToA*, *DCMPG* or *impdep2*. Most of identifiers were manually split in this step (including careful inspection of the source code to understand the exact context of those identifiers), but there was a reduced set where it was unfeasible to assign any evident meaning even after inspecting the source code. For example, about 120 Rhino dictionary entries fell into this category. Examples of such identifiers include short strings (*e.g.*, *DT*, *i3* or *m5*) and cryptic identifiers (*e.g.*, *P754*, *u00A0* or *zzz*).

During the oracle building process, we validated the split identifiers following the consensus approach described in Chapter 2, *i.e.*, we proposed an identifier split, which was then verified and validated by two other Ph.D. Students who already worked with the analyzed systems. In a few cases, we discussed disagreements. We adopted this approach in order to minimize the bias and the risk of producing erroneous results. This decision was motivated by the complexity of identifiers, which capture developers' domain and solution knowledge, experience, personal preference, etc.

7.1.3 Analyzed Systems

We conducted our evaluation on two open source Java systems, Rhino and jEdit, and constructed four datasets from these two systems. The first system considered is Rhino[2], an open-source implementation of JavaScript written in Java. Rhino version 1.6R5 has 138 classes, 1,870 methods and 32K lines of code. Rhino implements the specifications of the European Computer Manufacturers Association (ECMA) Script[3]. We constructed two datasets from Rhino. The first dataset is $\text{Rhino}_{Features}$ and contains 241 features extracted from the specifications. Each feature has a textual description that was used as a query in the evaluation. These descriptions correspond to sections of the ECMAScript specifications. Each feature also has a set of methods which are associated with the features (*i.e.*, gold set). The gold sets were constructed using the mappings between the source code and the features, which were made available by Eaddy *et al.* (Eaddy *et al.*, 2008a). These mappings were produced by considering the sections of the ECMAScript specification as features, and associating them with software artifacts using the following prune dependency rule, created by Eaddy *et al.* (Eaddy *et al.*, 2008b): "A program element is relevant to a concern if it should be removed, or otherwise altered, when the concern is pruned". These mappings were used in other research works, such as (Eaddy *et al.*, 2008a,b; Revelle *et al.*, 2010). Rhino is dis-

[2] `http://www.mozilla.org/rhino/`

[3] `http://www.ecmascript.org/`

tributed with a suite of test cases, and each test case has a correspondence in the ECMAScript specification. We used these test cases to collect full traces for each of the features.

The second dataset collected is Rhino$_{Bugs}$ and contains 143 issue reports (*i.e.*, bugs) that were collected from Bugzilla, the issue tracking system of Rhino[4]. Each bug from Bugzilla has a title and a description, and we used this information as queries in the evaluation. As in the Rhino$_{Features}$ dataset, we used the information made available by Eaddy et al. (Eaddy *et al.*, 2008b) to associate each bug with a set of methods from Rhino, which are responsible for the bug (*i.e.*, the gold set). Eaddy *et al.* extracted the mappings between bugs and source code by analyzing CVS commits. However, there was no association between the 143 issue reports and the test cases, hence, we did not collect any execution traces for this dataset.

The second system considered is jEdit[5], a popular open-source text editor written in Java. jEdit version 4.3 has 483 classes, 6.4K methods and 109K lines of code. We constructed two datasets from this system. The first dataset is jEdit$_{Features}$ and consists of 64 issues (34 features and 30 patches) extracted from jEdit's issue tracking system[6]. The second dataset is jEdit$_{Bugs}$ and consist of 86 bug reports.

We now describe some steps used for collecting additional information for these two datasets. We used the changes associated with the SVN commits between releases 4.2 and 4.3 to construct the gold sets. In addition, the SVN logs were parsed for issue identifiers which were matched against the issues from the tracking system. Similarly to the Rhino$_{Bugs}$ dataset, the title and description of these issues were used in the evaluation as queries. We used a tracer to generate marked traces, by executing jEdit and following the steps to reproduce from the issue description. Details about the process of generating this dataset, and the complete dataset, which includes queries and execution traces can be found in our online appendix[7].

The four datasets, extracted from Rhino and jEdit, which were used in the evaluation, are summarized in Table 7.3. We also present additional information about the datasets used in the evaluation in Table 7.2.

First, we present details about the number of methods from the gold sets of each dataset. Each data point (*i.e.*, a feature or a bug) from the Rhino$_{Features}$ dataset has on average 12 methods, whereas the Rhino$_{Bugs}$ dataset has only two methods on average. For jEdit there are on average four to six methods associated with each issue. The features from the Rhino$_{Features}$ dataset have many gold set methods in common, hence the total number of methods is much higher than for the other datasets.

[4] `https://bugzilla.mozilla.org/`
[5] `https://www.jedit.org/`
[6] `https://www.jedit.org/`
[7] `http://www.cs.wm.edu/semeru/data/icpc11-identifier-splitting/`

Table 7.2 Descriptive statistics from datasets: number of methods in the gold set (#GS_Methods), number of methods in traces (#TR_Methods), and number of identifiers from corpora (#CR_Identifiers).

# of	Measure	$\text{Rhino}_{Features}$	Rhino_{Bugs}	$\text{jEdit}_{Features}$	jEdit_{Bugs}
GS_Methods	min	1	1	1	1
	median	4	1	5	2
	average	12.82	2.24	6.3	4.01
	max	280	15	19	41
	st. dev	28.8	2.39	5.33	5.63
	total	3,089	320	403	345
TR_Methods	min	777	N/A	227	227
	median	917	N/A	1.1K	1.1K
	average	912	N/A	1.1K	1.1K
	max	1.1K	N/A	1.9K	1.9K
	st. dev	54	N/A	310	310
CR_Identifiers (with queries)	split by CamelCase	3,318 (4,154)	3,318 (4,223)	4,227 (4,361)	4,227 (4,596)
	split by Samurai	2,642 (3,416)	2,642 (3,411)	3,439 (3,552)	3,439 (3,751)
	Split by Oracle	2,030 (2,921)	2,030 (2,718)	2,758 (2,852)	2,758 (3,051)

Table 7.3 Summary of the four datasets used in the evaluation: name (number of features/issues), source of the queries and gold sets, and the type of execution information.

Dataset(Size)	Queries	Gold Sets	Execution Information
$\text{Rhino}_{Features}$ (241)	Sections of ECMAScript	Eaddy et al.	Full Execution
Rhino_{Bugs} (143)	Bug title and description	Eaddy et al.	(CVS) N/A
$\text{jEdit}_{Features}$ (64)	Feature (or Patch) title and description	SVN	Marked Execution Traces
jEdit_{Bugs} (86)	Feature (or Patch) title and description	SVN	Marked Execution Traces

Second, we present information about the number of methods extracted from the traces. For both systems, the average number of unique methods extracted from each trace was about one thousand. Third, we present information about the size of the corpora in terms of the number of identifiers, after applying the CamelCase, Samurai and Oracle. As expected, the more accurately we split the identifiers, the more we reduced the number of unique soft-words. For example, the corpus for $\text{Rhino}_{Features}$ has 3,318 soft-words after applying CamelCase, and has only 2,030 soft-words after using the Oracle. This is explained by the fact that identifiers that could not be split by CamelCase formed unique soft-words, whereas the Oracle split the identifier into two or more (common) terms that already appear in the

corpus, hence reducing the number of unique soft-words.

7.1.4 Analysis Method

For each dataset, every FLT will produce a list of ranks (*i.e.*, effectiveness measures) that has the size of the number of features in the dataset. For example, the dataset $\text{Rhino}_{Features}$ produced 241 ranks for $\text{IR}_{CamelCase}$, 241 ranks for $\text{IR}_{Samurai}$ and 241 ranks for IR_{Oracle}, and each of those ranks represents the best position (*i.e.*, lowest rank) of a method from the gold set associated with that feature. These lists of ranks are used as an input for the following comparison techniques: descriptive statistics, side by side comparisons, and statistical tests.

First, we compared the ranks using descriptive statistics, such as minimum, first quartile, median, third quartile, maximum, and average. We presented all these descriptive statistics graphically, using box plots (*i.e.*, whisker charts). Although this technique provides a quick and intuitive view of the data, it only presents a high level perspective. The second comparison technique examines the data in more details and works as follows. Given two lists of ranks produced by two different FLTs, we compared the ranks side by side and counted the number of cases the first technique produces lower ranks than the other, as well as the number of cases the second technique produces lower ranks (*i.e.*, better results) than the other. We reported these values as percentages.

The third comparison of the ranks is a statistical analysis. We used the Wilcoxon signed-rank test (Conover, 1998) to test whether the difference in terms of effectiveness for two measures is statistically significant or not. This test is non-parametric, and it takes as an input two lists of ranks produced by two different feature techniques. In the test we used a significance level $\alpha = 0.05$, and the output of the test is a p-value, which can be interpreted as follows. If the p-value is less than α, then the difference in ranks produced by one feature location technique is statistically significantly lower than the ranks produced by the other technique. Otherwise, if the p-value is larger than α, then we concluded that the two techniques produced almost equivalent results.

7.1.5 Hypotheses

We formulated several null hypotheses in order to test whether an improved splitting algorithm has a higher effectiveness measure than a simple splitting algorithm. For example:

1. $\text{H}_{0,IRSamurai}$: There is no statistical significant difference in terms of effectiveness between $\text{IR}_{Samurai}$ and $\text{IR}_{CamelCase}$.

2. $\text{H}_{0,IRSamurai}$Dyn: There is no statistical significant difference in terms of effectiveness between $\text{IR}_{Samurai}$Dyn and $\text{IR}_{CamelCase}$Dyn.

We also defined several alternative hypotheses for the case when a null hypothesis is rejected with high confidence. These alternative hypotheses state that an improved identifier splitting technique (*e.g.*, Samurai or Oracle) would produce higher effectiveness than the baseline splitting technique (*i.e.*, CamelCase). The following alternative hypotheses correspond to the null hypotheses defined above.

1. $H_{a,IRSamurai}$: $IR_{Samurai}$ has statistically significantly higher effectiveness than $IR_{CamelCase}$.

2. $H_{a,IRSamurai}$Dyn: $IR_{Samurai}$Dyn has statistically significantly higher effectiveness than $IR_{CamelCase}$Dyn

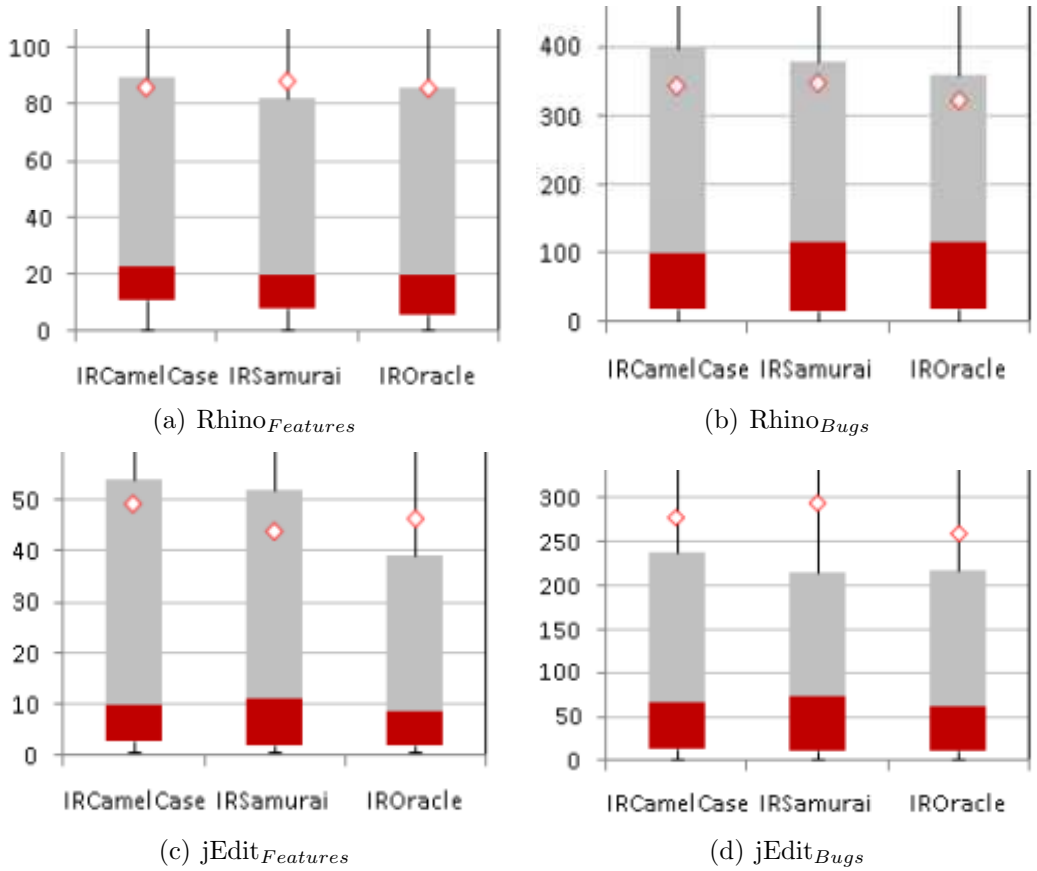
   The corresponding null and alternative hypotheses for the Oracle splitting technique were defined analogously.

7.2 Results and Discussion

This section presents the effectiveness measures of the FLTs presented in Table 7.4, which were applied on the four datasets (cf. Table 7.6) extracted from Rhino and jEdit.

In Fig. 7.1, we present the box plots of the effectiveness measures of the three IR-based FLTs applied on the four datasets. For each dataset, all the instances of the IR FLT produced very similar results in terms of lower quartile, median, mean, upper quartile, etc. For example, Fig. 7.1(a) shows that for the $\text{Rhino}_{Features}$ dataset, using CamelCase $IR_{CamelCase}$, we obtained a median of 23 and an average of 86, and if we used the Oracle splitting IR_{Oracle}, we obtained a median of 20 and an average of 86. The same small differences between the descriptive statistics measures are observed among all the IR instances, and in all the four datasets.

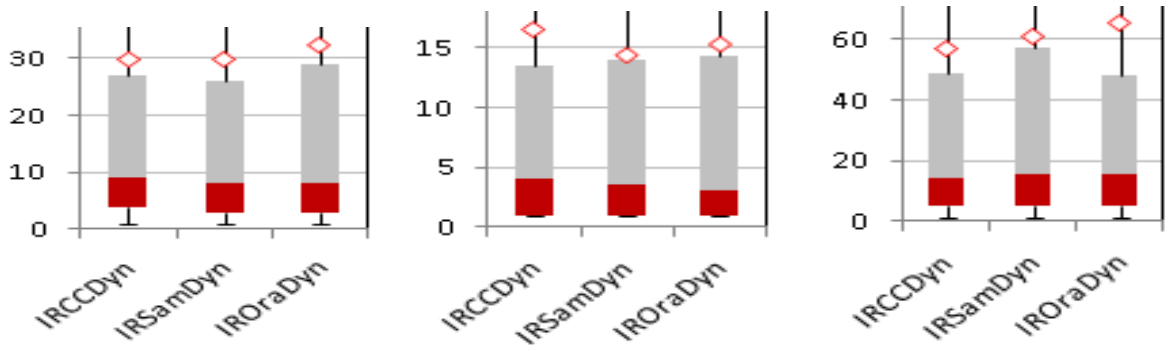
Similarly to Fig. 7.1, Fig. 7.2 presents the box plots of the effectiveness measure of the three IRDyn FLTs which were applied on the following three datasets: $\text{Rhino}_{Features}$ (cf. Fig. 7.1(a)), Rhino_{Bugs} (cf. Fig. 7.1(b)), and $\text{jEdit}_{Features}$ (cf. Fig. 7.1(c)). For all the datasets, the three FLTs produced almost identical results, regardless of the technique used for splitting the identifiers. For example, Fig. 7.1(a) shows that for the $\text{Rhino}_{Features}$ dataset, using CamelCase splitting $IR_{CamelCase}$Dyn, the median and average are 9 and 30 respectively, whereas for Oracle splitting IR_{Oracle}Dyn the median and average are 8 and 32 respectively. The small differences observed on the IR based instances are also observed here. Even more so, for the other datasets, when incorporating dynamic information the differences produced by the feature location techniques seem to be less noticeable than the differences produced by IR-based FLTs. This fact may suggest that dynamic information has some influence and the splitting techniques used for identifiers may not be as important. It is



(a) Rhino$_{Features}$ (b) Rhino$_{Bugs}$

(c) jEdit$_{Features}$ (d) jEdit$_{Bugs}$

Figure 7.1 Box plots of the effectiveness measure of the three IR-based FLTs (IR$_{CamelCase}$, IR$_{Samurai}$ and IR$_{Oracle}$) for the four datasets: Rhino$_{Features}$, Rhino$_{Bugs}$, jEdit$_{Features}$, jEdit$_{Bugs}$.



(a) Rhino$_{Features}$ (b) Rhino$_{Bugs}$ (c) jEdit$_{Features}$

Figure 7.2 Box plots of the effectiveness measure of the three FLTs (IR$_{CamelCase}$Dyn (IR$_{CC}$Dyn), IR$_{Samurai}$Dyn (IR$_{Sam}$Dyn) and IR$_{Oracle}$Dyn (IR$_{Ora}$Dyn) for the three datasets: a) Rhino$_{Features}$, b) jEdit$_{Features}$, and c) jEdit$_{Bugs}$.

also interesting to observe that feature location techniques applied on the datasets that use features as queries (*i.e.*, Rhino$_{Features}$ and jEdit$_{Features}$) have lower effectiveness measures than the feature location techniques applied on the datasets that use bug descriptions as queries. For example, for Rhino, the median effectiveness when using feature descriptions as queries is about 21 (cf. Fig. 7.1(a)), whereas the median effectiveness when using bug descriptions as queries is about 110 (cf. Fig. 7.1(b)). The same observation is valid for the jEdit when only textual information is used (cf. Fig. 7.1(c) and 7.1(d)) as well as when textual and execution information are combined (cf. Fig. 7.1(a) and 7.1(b)).

Table 7.4 Percentages of times the effectiveness of the FLT from the row is higher than IR$_{CamelCase}$.

FLT	RHINO$_{Features}$ (%)	RHINO$_{Bugs}$ (%)	JEDIT$_{Features}$ (%)	JEDIT$_{Bugs}$ (%)
IR$_{Samurai}$	9	36	33	41
IR$_{Oracle}$	49	45	44	40

Table 7.5 Percentages of times the effectiveness of the IR$_{CamelCase}$ is higher than the FLT from the row.

FLT	RHINO$_{Features}$ (%)	RHINO$_{Bugs}$ (%)	JEDIT$_{Features}$ (%)	JEDIT$_{Bugs}$ (%)
IR$_{Samurai}$	40	48	36	41
IR$_{Oracle}$	33	48	38	55

The results illustrated in Fig. 7.1(a) and Fig. 7.1(b) provide only a high level picture of the effectiveness measure. We now present results from a case by case comparison of the effectiveness measure. In Table 7.4, we present the percentage of times an instance of the IR-based FLT produced lower ranks than another instance of the IR-based FLT. The first cell value represents the percentage of times the FLT from the corresponding row produced lower ranks than IR$_{CamelCase}$, whereas the number in parenthesis represents the percentage of times IR$_{CamelCase}$ produced lower ranks than the technique from the row (in the remaining percentages, the two techniques produce identical ranks). In this case, a higher percentage denotes a more effective technique. Similarly, Table 7.6 shows the percentage of times the FLT from the row produced better results than IR$_{CamelCase}$Dyn.

We observe from Tables 7.4 and 7.5 that comparing the effectiveness measures of IR$_{Oracle}$ and IR$_{CamelCase}$ side by side, IR$_{Oracle}$ produced lower ranks in 49% of cases, whereas IR$_{CamelCase}$

Table 7.6 Percentages of times the effectiveness of the FLT from the row is higher than $\text{IR}_{CamelCase}$Dyn.

FLT	$\text{Rhino}_{Features}$ (%)	Rhino_{Bugs} (%)	$\text{jEdit}_{Features}$ (%)	jEdit_{Bugs} (%)
$\text{IR}_{Samurai}$Dyn	33	N/A	27	28
IR_{Oracle}Dyn	42	N/A	34	35

Table 7.7 Percentages of times the effectiveness of the $\text{IR}_{CamelCase}$Dyn is higher than the FLT from the row.

FLT	$\text{Rhino}_{Features}$ (%)	Rhino_{Bugs} (%)	$\text{jEdit}_{Features}$ (%)	jEdit_{Bugs} (%)
$\text{IR}_{Samurai}$Dyn	36	N/A	22	41
IR_{Oracle}Dyn	35	N/A	22	50

produced better results in 33% of cases. In the remaining 18% of cases (*i.e.*, 100%-49%-33%) the two techniques produced identical ranks. Similarly, from Tables 7.6 and 7.7 we observe that when dynamic information is taken into account, for the $\text{Rhino}_{Features}$ dataset, IR_{Oracle}Dyn produced lower ranks (*i.e.*, better results) in 42% of cases, whereas $\text{IR}_{CamelCase}$Dyn produced better results in 35% of cases. In the remaining 23% of cases (*i.e.*, 100%- 42%-35%) the techniques produced the same results. It is interesting to observe that for both systems, IR_{Oracle} and $\text{IR}_{OracleDyn}$ produced a higher percentage of good results than $\text{IR}_{CamelCase}$ and $\text{IR}_{CamelCase}$Dyn respectively, when these techniques are applied on the datasets that use features as queries (columns two and four of the last rows of Tables 7.4 and 7.6). However, when these techniques are applied on the datasets that use bug description as queries, the opposite phenomenon is observed. In other words, $\text{IR}_{CamelCase}$ and $\text{IR}_{CamelCase}$Dyn produced higher percentage of good results than IR_{Oracle} and IR_{Oracle}Dyn respectively. The effectiveness measures presented as box plots and percentages are statistically analyzed using the Wilcoxon signed-rank test.

In Table 7.8, we present the p-values of the Wilcoxon signed-rank test for all the instances of the IR-based FLTs. The results that are statistically significant (*i.e.*, the p-value is lower than $\alpha = 0.05$) are highlighted in bold. The table shows that there is only one instance when the Oracle splitting technique (*i.e.*, IR_{Oracle}) produced results that are statistically significantly better than the technique that uses CamelCase splitting (*i.e.*, $\text{IR}_{CamelCase}$). This is for the $\text{Rhino}_{Feature}$ dataset and the p-value is equal to 0.005. We performed the same

Table 7.8 The p-values of the Wilcoxon signed-rank test for the FLT from the row compared with $\text{IR}_{CamelCase}$ (stat. significance values are in bold).

FLT	$\text{RHINO}_{Features}$	RHINO_{Bugs}	$\text{JEDIT}_{Features}$	JEDIT_{Bugs}
$\text{IR}_{Samurai}$	0.692	0.890	0.742	0.479
IR_{Oracle}	**0.005**	0.497	0.202	0.785

analysis between IR_{Oracle} and $\text{IR}_{Samurai}$ and the results show that only for the $\text{Rhino}_{Features}$ dataset IR_{Oracle} produced results that are statistically significantly better than $\text{IR}_{Samurai}$ (p-value=0.009).

Table 7.9 The p-values of the Wilcoxon signed-rank test for the FLTs from the row compared with $\text{IR}_{CamelCase}$Dyn (there are no stat. significant values).

FLT	$\text{RHINO}_{Features}$	RHINO_{Bugs}	$\text{JEDIT}_{Features}$	JEDIT_{Bugs}
$\text{IR}_{SamuraiDyn}$	0.713	N/A	0.307	0.928
$\text{IR}_{OracleDyn}$	0.265	N/A	0.095	0.937

Similarly, Table 7.9 shows the p-values of the Wilcoxon signed-rank test applied on the effectiveness measures produced by the IRDyn FLTs. The results show that no technique produced statistically better results than any other technique. This observation helps in answering the research questions RQ2 and RQ4, that the splitting technique used is not as important if dynamic information is considered. When dynamic information is involved, no technique produced statistically significant results than the other for any of the datasets. If we look at the same results (*i.e.*, the effectiveness measure) from three different points of view (*i.e.*, box plots, percentages and statistical analysis), we derive the following conclusions. First, there are instances where a better identifier splitting technique (*i.e.*, Oracle) improves feature location. This has been the case for the Rhino, for the $\text{Rhino}_{Features}$ dataset. Second, there are cases when even a perfect identifier splitting technique cannot help in the process of feature location. Such an example is given by the $\text{jEdit}_{Features}$ dataset, when the effectiveness measure is improved for a few cases, but the difference is not statistically significant. Moreover, there are instances where the perfect splitting technique can have negative impact on feature location, as it was the case for the jEdit_{Bugs} dataset. In this case, the original CamelCase splitting technique produced better results than the Oracle in terms of percentages (cf. Table 7.4), but the difference is still not statistically significant. Finally, there is one instance, $\text{Rhino}_{Feature}$ dataset, where splitting helps when textual information is used.

However, when dynamic information is used, all the splitting techniques produce equivalent results from a statistical point of view.

7.3 Qualitative Analysis

This section presents some observations after examining the results produced by the splitting techniques and after examining the queries. One of the problems that we encountered using Samurai was that it tended to split certain types of identifiers into many meaningless terms, some of them having between one to three characters. Examples of identifiers from Rhino, where Samurai split them incorrectly were: *debugAccelerators*, *tolocale*, *imitating*, *imlementation*, etc. Their incorrect Samurai splitting was: *debug Ac ce le r at o rs*, em tol ocal e, *imi ta ting*, *i ml eme n tat ion* (cf. Table 7.10). For these examples, CamelCase performed better, as it correctly split the first identifier (debug accelerators), but it left the other ones unaltered.

Table 7.10 Examples of splitted identifiers from Rhino using CamelCase and Samurai. The identifiers which are split correctly are highlighted in bold.

Original Identifier	CamelCase	Samurai
GETPROP	getprop	**GET PROP**
readadapterobject	readadapterobject	**read adapter object**
SHORTNUMBER	shortnumber	**SHORT NUMBER**
debugAccelerators	**debug accelerators**	debug Ac ce le r at o rs
tolocale	tolocale	tol ocal e
imitating	**imitating**	imi ta ting

One of the benefits of using Samurai was that it accurately split same-case identifiers composed of multiple words. For these cases, CamelCase left the identifiers unmodified. Examples of such identifiers from Rhino include *SHORTNUMBER*, *readadapterobject*, *GETPROP* which are correctly split by Samurai as *SHORT NUMBER*, *read adapter object*, and *GET PROP*, and are left unchanged by CamelCase (cf. Table 7.10). However, there were some cryptic identifiers that were almost impossible to split using CamelCase or Samurai. Examples of such identifiers from Rhino include *ldbl*, *njm*, *pun*, *rve*, *wbdry*, etc. In these cases, inferring the meaning from the context in which these identifiers appeared was the only way to split them correctly.

We observed a vocabulary mismatch problem, which produced inconsistencies between the identifiers used in the queries, and the identifiers used in the code. This problem seemed to be less noticeable for features, and more severe for bugs. For jEdit, the issues that described

features often contained terms that were later used in the code as identifiers for classes, methods, variables, etc. For example, jEdit's feature #16084869 ("Support 'thick' caret") contained in its description many identifiers that were also found in the name of the methods (*e.g.*, *thick*, *caret*, *text*, *area*, etc.). For features, their queries were expressive, and more consistent with the source code vocabulary, so they benefitted less from an Oracle splitting. Hence, when using feature descriptions as queries for both Rhino and jEdit, the median effectiveness of the FLTs, regardless of splitting, were about 20 for Rhino (cf. Fig. 7.1(a)) and about 10 for jEdit (cf. Fig. 7.1(c))).

On the other hand, the vocabulary of the queries extracted from bug reports was less consistent with the source code vocabulary, and a splitting technique, helped bridge this gap. For example, jEdit's bug #157550510 ("C+j bug") reported a problem with the "join lines" implementation; yet nowhere in its description were the words *join* or *lines* mentioned. In general, the identifiers from the bug descriptions were less consistent with the code, and this issue was reflected in terms of the effectiveness measures produced by the FLTs, when these bug descriptions were used as queries. For example, in Figure 1 (b) the median effectiveness for Rhino system was about 110 (as opposed to a median of 20 when features were used as queries). Also, Figure 1 (d), shows that the median effectiveness of the techniques that used bugs as queries was around 67, as opposed to 10, which was the median effectiveness when features were used as queries.

Another problem with the queries is that some identifiers were used just for communication between developers, and no matter what splitting technique was used, these identifiers provided no useful information, because they appeared only in the query vocabulary, and did not appear at all in the source code vocabulary. Examples of such identifiers included words that are common in communication, such as *btw* (*i.e.*, by the way), *thanks*, *hate*, *rant*, *greetings*, *fly*, *annoying*, etc., name of developers, *ApeHanger*, *Slava*, *Carlos*, etc.

7.4 Threats to Validity

Threats to **construct validity** are mainly due to mistakes in the oracle and gold sets. We cannot guarantee that no errors are present in the oracle. As the intent of the oracle is to explain identifier semantics, we cannot guarantee that there is no difference between oracle splits and splits of developers that originally created the identifiers. This problem is difficult and it relates to guessing the developers' intent. To limit this threat, different sources of information such as comments, source code context, and online documentation were used when producing the oracle. To minimize the risk to the accuracy of the gold set, we used data produced by other researchers, which was used in previous studies and made available

to the research community.

Threats to **internal validity** are due to the subjectivity in the manual building of the oracle and to the possible biases introduced by manually splitting identifiers. To limit this threat, the oracle was produced by a joint work among the authors, using CamelCase, Samurai and TIDIER. In addition, inconsistencies in splitting/mapping to dictionary words were discussed.

Threats to **conclusion validity** concern the relations between the treatment and the outcome. Proper tests were performed to statistically reject the null hypotheses. In particular, we used a non-parametric test (*i.e.*, Wilcoxon signed-rank test), which does not make any assumptions on the underlying distributions of the data. Furthermore, we adjusted significant p-values (cf. Table 7.8) using conservative Bonferroni correction. Our significant p-value remained significant as the limit in such case is equal to α-value/number of tests (*i.e.*, $0.05 / 3 = 0.01666 < 0.05$).

Threats to **external validity** concern the possibility of generalizing our results. To make our results as generalizable as possible, we used two Java applications from two different application domains but we cannot be sure that our findings will be valid for other domains, applications, programming languages or software engineering tasks (*i.e.*, different from feature location). More case studies are needed to confirm the results presented and to verify if indeed, in the general case, dynamic information reduces the gain of more sophisticated identifier split techniques.

7.5 Chapter Summary

Perfecting splitting techniques can improve the accuracy of feature location, easing program comprehension and thus, software evolution. This improvement is pronounced in situations where execution information cannot be collected (*e.g.*, mission critical and time critical applications). In fact, by splitting source code identifiers and mapping them to domain concepts, the localization of entities contributing to implementing some user observable functionality may be easier, which could minimize feature location effort. In this chapter, we presented an exploratory study of two FLTs (*i.e.*, IR and IRDyn) for locating bugs and features, utilizing three strategies for splitting identifiers: CamelCase, Samurai and manual splitting of identifiers. These FLTs and their preprocessing techniques were evaluated on two open-source systems, Rhino and jEdit, and compared in terms of their effectiveness measure.

The results of the IR-based FLT reveal that Samurai and CamelCase produced similar results. However, the IR_{Oracle} outperforms $\text{IR}_{CamelCase}$ in terms of the effectiveness measure, on the $\text{Rhino}_{Features}$ dataset. This supports our conjecture that when only textual informa-

tion is available, an improved splitting technique can help improve effectiveness of feature location. The results also show that when both textual and execution information are used, any splitting algorithm will suffice, as FLTs produced equivalent results. In other words, because execution information helps pruning the search space considerably, the benefit of an advanced splitting algorithm is comparably smaller than the benefit obtained from execution information; hence, the splitting algorithm will have little impact on the final results. Overall, our findings outline potential benefits of creating advanced preprocessing techniques as they can be useful in situations where execution information cannot be easily collected.

CHAPTER 8

Impact of Identifier Splitting on Traceability Recovery

Identifiers and comments represent an important source of information used by (semi-) automated techniques to recover traceability links among software artifacts (Antoniol *et al.*, 2002; Marcus *et al.*, 2005) and locate features in source code (Marcus *et al.*, 2004, 2005; Poshyvanyk *et al.*, 2007; Eaddy *et al.*, 2008a; Revelle et Poshyvanyk, 2009; Revelle *et al.*, 2010). The latter IR-based software maintenance tasks rely on the consistency of the source code lexicon available in the different artifacts and their effectiveness may worsen if programmers introduce non-meaningful identifiers.

Identifier splitting approaches (*e.g.*, (Lawrie *et al.*, 2010; Lawrie et Binkley, 2011; Guerrouj *et al.*, 2013a)) have been suggested to tackle the vocabulary mismatch problem that exists between source code and other project's artifacts with the aim of reaping the full benefits of IR-based techniques. However, there is a lack of research work on the impact of identifier splitting/expansion on traceability recovery. Thus, we perform an empirical study aiming at investigating the effect of identifier splitting on two traceability recovery techniques. The first technique uses LSI, while the second is based on VSM.

In this chapter, we first describe our empirical study design, then we show the results of our study and the qualitative analysis performed in support of our quantitative findings.

8.1 Empirical Study Design

The *goal* of this study is to compare the accuracy (*i.e.*, precision and recall) of two traceability recovery techniques; one is based on LSI (Liu *et al.*, 2007) and the second uses VSM (Eaddy *et al.*, 2008a) (cf. Chapter 2), when utilizing three identifier splitting algorithms: CamelCase, Samurai and Oracle (*i.e.*, manual splitting of identifiers). The *perspective* is of researchers who want to understand how approaches for splitting identifiers can impact accuracy of traceability recovery techniques, including best possible scenario where splitting is done by experts. The *context* of this investigation consists of three open-source systems: iTrust, Pooka, and Lynx, their main characteristics are described in Section 8.1.3.

8.1.1 Variable Selection and Study Design

The main independent variable is the type of splitting algorithm used: CamelCase, Samurai or Oracle (*i.e.*, manually split identifiers). In the following, we will use CamelCase and

baseline interchangeably.

The second independent variable is the technique used for traceability recovery. Thus, we have two traceability recovery techniques, and each has three configurations, which depend on the identifier splitting technique used. For example, $\text{LSI}_{CamelCase}$, $\text{LSI}_{Samurai}$, and LSI_{Oracle} are the LSI-based traceability recovery techniques that use LSI to compute similarities between documents, after applying the CamelCase and Samurai algorithms, and the Oracle splitting on the identifiers. Similarly, $\text{VSM}_{CamelCase}$, $\text{VSM}_{Samurai}$, and VSM_{Oracle} are the VSM-based traceability recovery techniques that use VSM to compute similarities between documents, after applying the three above-mentioned splitting techniques respectively. In Table 8.1, we summarize the various instances of traceability recovery techniques we dealt with.

Table 8.1 The configurations of the two studied traceability recovery (TR) techniques based on the splitting algorithm.

Splitting Algorithm	LSI-based TR	VSM-based TR
CamelCase	$\text{LSI}_{CamelCase}$	$\text{VSM}_{CamelCase}$
Samurai	$\text{LSI}_{Samurai}$	$\text{VSM}_{Samurai}$
Oracle (Manual Split)	LSI_{Oracle}	VSM_{Oracle}

The dependent variables considered in our study are the precision and recall provided by the traceability recovery techniques in question. The definition of precision and recall is provided in Chapter 2 of this thesis.

We aimed at answering the following overarching question: *How does different identifiers splitting techniques impact traceability recovery?*

We answered this question by examining these more specific research questions (RQs) :

1. **RQ1**: Does $\text{LSI}_{Samurai}$ outperform $\text{LSI}_{CamelCase}$ in terms of accuracy?
2. **RQ2**: Does LSI_{Oracle} outperform $\text{LSI}_{CamelCase}$ in terms of accuracy?
3. **RQ3**: Does LSI_{Oracle} outperform $\text{LSI}_{Samurai}$ in terms of accuracy?

By accuracy, we mean the precision and recall of the studied traceability recovery techniques.

To address these research questions, we first used three different splitting techniques, *i.e.*, CamelCase, Samurai, and oracle to built three corpora. The oracle was built for the three studied systems (*i.e.*, iTrust, Pooka, and Lynx) using the same multi-step strategy we adopted

in our study on feature location (cf. Chapter 7). Second, we used each corpus to recover the traceability links. Third, we computed the precision and recall of each traceability recovery set to measure the improvement brought by the identifier splitting technique in question.

8.1.2 Building Traceability Recovery Sets

We used each corpora built using CamelCase, Samurai, and splitting/expansion oracles created as described in Chapter 2. We performed standard document preprocessing steps on these corpora (Gotel et Finkelstein, 1993). For each corpus, we built different traceability recovery sets at different similarity threshold points. The similarity threshold helps to retrieve only a set of traceability links whose similarity is above than a certain level. These sets help to evaluate, which approach is better than the other at all the threshold values or some specific thresholds values. We used LSI and VSM to recover traceability links between requirements and source code documents.

We used a threshold t to prune the set of traceability recovery links, keeping only links whose similarities values are greater than or equal to $t \in [0, 1]$. We considered different values of t from 0.01 to 1 per steps of 0.01 to obtain different sets of traceability recovery links with varying precision and recall values. We used these different sets to assess which approach provides better precision and recall values.

8.1.3 Analyzed Systems

iTrust[1] is a medical application written in Java; it provides patients with a means to keep up with their medical history and records as well as communicate with their doctors. iTrust (version 10) dataset contains 35 and 218 requirements and classes respectively.

Pooka[2] is an e-mail client written in Java using the JavaMail API. Pooka (version 2.0) dataset contains 90 and 298 requirements and classes respectively. This dataset contains manually validated requirements to class traceability recovery links.

Lynx[3] is a basic textual Web browser. Lynx is entirely written in C. Lynx (version 2.8.5) dataset has 247 files, 174 KLOCs, and 2,067 methods. This dataset contains manually validated requirements to method traceability recovery links.

8.1.4 Analysis Method

To assess whether the differences in precision and recall values, in function of the threshold t, are statistically significant or not, we used the Mann-Whitney test defined in Chapter 2.

[1] `http://agile.csc.ncsu.edu/iTrust/`
[2] `http://www.suberic.net/pooka/`
[3] `http://lynx.isc.org/`

In addition to the statistical comparisons, we computed the effect-size of the difference using Cliff's delta (d) non-parametric effect size measure (Grissom et Kim, 2005) that we also explained in the background chapter of this thesis (cf. Chapter 2).

All the above-mentioned computations have been applied using the R statistical environment (Team, 2012).

8.1.5 Hypotheses

We formulated several null hypotheses in order to test whether an improved splitting algorithm has a higher precision (recall) than a simple splitting algorithm. For example:

1. $H_{0,1}$: $LSI_{CamelCase}$ and $LSI_{Samurai}$ provide equal precision.
2. $H_{0,2}$: $LSI_{CamelCase}$ and LSI_{Oracle} provide equal precision.
3. $H_{0,3}$: LSI_{Oracle} and $LSI_{Samurai}$ provide equal precision.

Similar null hypotheses were defined for the traceability recovery techniques using VSM.

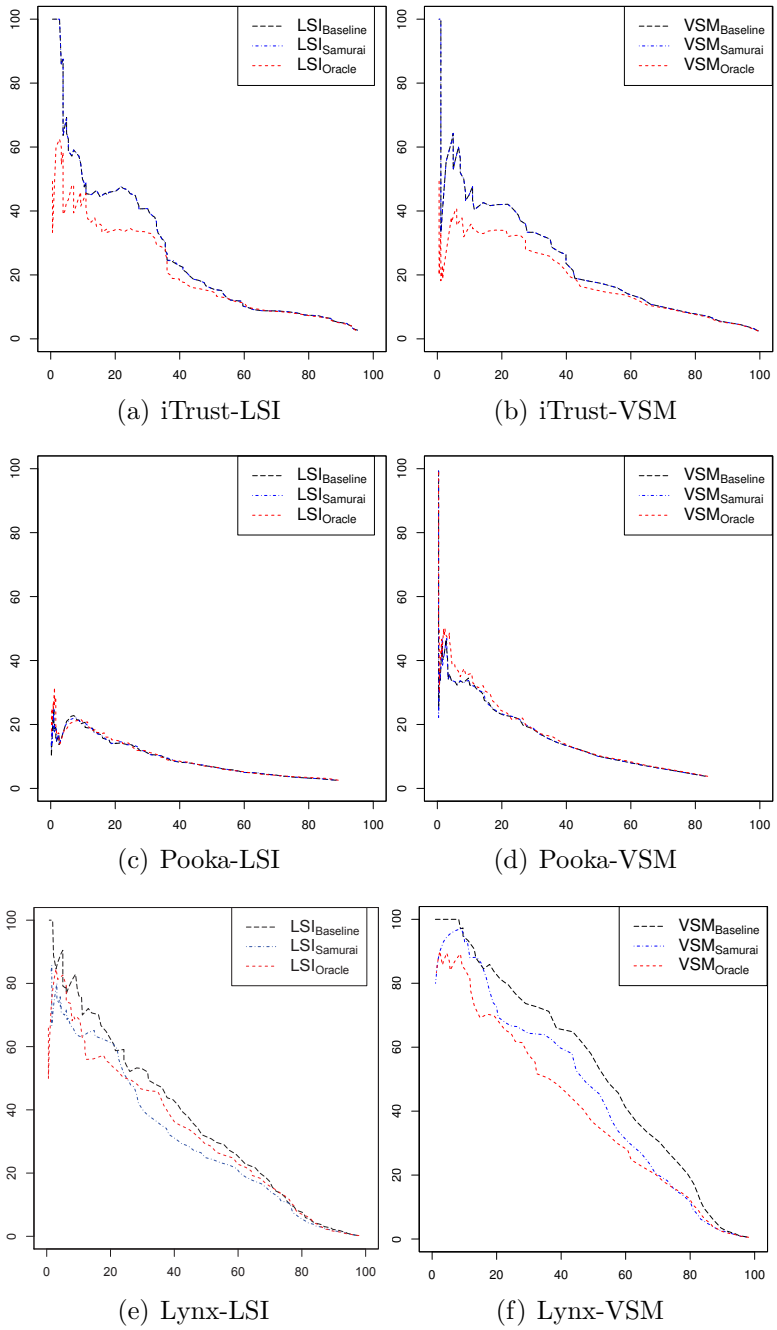
We also defined several alternative hypotheses for the case when a null hypothesis is rejected with high confidence. These alternative hypotheses state that an improved identifier splitting technique (*e.g.*, Samurai or Oracle) would produce higher accuracy than the baseline splitting technique (*i.e.*, CamelCase). The following alternative hypotheses correspond to the null hypotheses defined above.

1. $H_{a,1}$: $LSI_{Samurai}$ has statistically significantly higher precision than $LSI_{CamelCase}$.
2. $H_{a,2}$: LSI_{Oracle} has statistically significantly higher precision than $LSI_{CamelCase}$.
3. $H_{a,3}$: LSI_{Oracle} has statistically significantly higher precision than $LSI_{Samurai}$.

   The corresponding null and alternative hypotheses for the configuration using VSM were defined analogously.

8.2 Results and Discussion

Figure 8.1 shows, for the three systems (*i.e.*, iTrust, Pooka, Lynx), that using manually split oracle does not perform any better than the baseline splitting techniques. At some threshold points, in some cases, manually split oracle slightly provide better results. For example, VSM_{Oracle} provides slightly better precision and recall at certain threshold points. In other cases, the baseline provides better results than the oracle. For example, for lynx



(a) iTrust-LSI (b) iTrust-VSM

(c) Pooka-LSI (d) Pooka-VSM

(e) Lynx-LSI (f) Lynx-VSM

Figure 8.1 Precision and recall values of $\text{VSM}_{CamelCase}$, VSM_{Oracle}, $\text{VSM}_{Samurai}$, $\text{LSI}_{CamelCase}$, LSI_{Oracle}, and $\text{LSI}_{Samurai}$ with the threshold t varying from 0.01 to 1 by step of 0.01. The x axis shows recall and y axis shows precision.

using VSM, the baseline provides better results, *i.e.*, 100% of precision and recall rather than 87% for the oracle.

Table 8.2 shows the average precision and recall values at all the threshold points considered. The bold values in these tables represent the improved precision and–or recall values

Table 8.2 Average values of precision and recall for iTrust, Pooka, and Lynx. Bold values show the improvement brought by using Oracle.

	Precision			Recall		
	$\text{LSI}_{CamelCase}$	$\text{LSI}_{Samurai}$	LSI_{Oracle}	$\text{LSI}_{CamelCase}$	$\text{LSI}_{Samurai}$	LSI_{Oracle}
iTrust	36.49	36.49	28.39	36.61	36.61	34.23
Pooka	14.06	14.14	**15.64**	22.31	22.37	22.36
Lynx	45.43	39.08	39.40	41.99	40.82	41.55

	$\text{VSM}_{CamelCase}$	$\text{VSM}_{Samurai}$	VSM_{Oracle}	$\text{VSM}_{CamelCase}$	$\text{VSM}_{Samurai}$	VSM_{Oracle}
iTrust	48.99	48.99	25.82	23.77	23.77	23.07
Pooka	40.54	40.54	**42.07**	11.59	11.63	**12.19**
Lynx	64.26	57.84	49.91	37.66	37.05	**40.16**

over baseline splitting. As it can be noticed, for Pooka, LSI_{Oracle} produced better results (15.64%), in terms of precision, than $\text{LSI}_{Samurai}$ (14.14%) and $\text{LSI}_{CamelCase}$ (%14.06). In addition, VSM_{Oracle} produced higher precision (42.07%) than $\text{VSM}_{Samurai}$ and $\text{VSM}_{CamelCase}$ (40.54%). This is not the case for iTrust where $\text{LSI}_{CamelCase}$ and $\text{LSI}_{Samurai}$ produced the same precision (36.49%), and LSI_{Oracle} showed a lower precision (28.39%). Regarding iTrust using VSM, as for LSI, $\text{VSM}_{CamelCase}$ and $\text{VSM}_{Samurai}$ produced the same precision (48.99%) and VSM_{Oracle} showed a lower precision than them (25.82%). For what concerns Lynx, LSI_{Oracle} produced a lower precision (39.40%) than $\text{LSI}_{Samurai}$ (39.08%). However, $\text{LSI}_{CamelCase}$ showed a higher precision (45.43%) than both techniques. Traceability recovery techniques based on VSM, applied on Lynx, showed that $\text{VSM}_{CamelCase}$ (64.26%) produced higher precision than $\text{LSI}_{Samurai}$ (57.84%) and LSI_{Oracle} (49.91%).

Concerning the recall, Table 8.2 shows that, for Pooka, VSM_{Oracle} produced a higher recall (12.19%) than $\text{VSM}_{Samurai}$ (11.63%) and $\text{VSM}_{CamelCase}$ (11.59%). Results also indicated that, for Lynx, VSM_{Oracle} provided higher recall (40.16%) in comparison with $\text{VSM}_{Samurai}$ (37.05%) and $\text{VSM}_{CamelCase}$ (37.66%). Regarding iTrust, VSM_{Oracle}, $\text{VSM}_{Samurai}$, and $\text{VSM}_{CamelCase}$ produced almost similar recall results ($\sim$23%). The results obtained for recall using LSI do not show any improvement (cf. Table 8.2).

In Tables 8.3 and 8.4, we report the p-values and effect size of different comparisons between splitting techniques in terms of precision and recall respectively. The bold values in the latter tables represent the improvement brought by the manually built oracle and italic values represent the improvement brought by Samurai splitting technique. If the p-value is not in bold but significant, this means that CamelCase (or Samurai) produces better results than the Oracle. If the p-value is not in italics but significant, this means that CamelCase produces better results than Samurai.

Results were statically improved using manually built oracle with none and–or large effect size for Pooka. In some cases, manually built oracle and Samurai statistically decreased the

Table 8.3 Precision: p-values and effect size of different identifiers splitting techniques.

Precision			
iTrust			
Approach 1	Approach 2	p-value	Cliff's d
$\text{LSI}_{CamelCase}$	$\text{LSI}_{Samurai}$	0.76	0.01
$\text{LSI}_{CamelCase}$	LSI_{Oracle}	<0.01	0.54
LSI_{Oracle}	$\text{LSI}_{Samurai}$	<0.01	0.54
$\text{VSM}_{CamelCase}$	$\text{VSM}_{Samurai}$	0.42	0.0002
$\text{VSM}_{CamelCase}$	VSM_{Oracle}	<0.01	0.77
VSM_{Oracle}	$\text{VSM}_{Samurai}$	<0.01	0.77
Pooka			
Approach 1	Approach 2	p-value	Cliff's d
$\text{LSI}_{CamelCase}$	$\text{LSI}_{Samurai}$	0.97	0.02
$\text{LSI}_{CamelCase}$	LSI_{Oracle}	**<0.01**	**0.45**
LSI_{Oracle}	$\text{LSI}_{Samurai}$	<0.01	0.45
$\text{VSM}_{CamelCase}$	$\text{VSM}_{Samurai}$	0.72	0.07
$\text{VSM}_{CamelCase}$	VSM_{Oracle}	**<0.01**	**0.17**
VSM_{Oracle}	$\text{VSM}_{Samurai}$	<0.01	0.12
Lynx			
Approach 1	Approach 2	p-value	Cliff's d
$\text{LSI}_{CamelCase}$	$\text{LSI}_{Samurai}$	<0.01	1.00
$\text{LSI}_{CamelCase}$	LSI_{Oracle}	<0.01	0.84
LSI_{Oracle}	$\text{LSI}_{Samurai}$	<0.01	0.62
$\text{VSM}_{CamelCase}$	$\text{VSM}_{Samurai}$	<0.01	0.56
$\text{VSM}_{CamelCase}$	VSM_{Oracle}	<0.01	0.22
VSM_{Oracle}	$\text{VSM}_{Samurai}$	<0.01	0.22

Table 8.4 Recall: p-values and effect size of different identifiers splitting techniques.

Recall			
iTrust			
Approach 1	Approach 2	p-value	Cliff's d
$\text{LSI}_{CamelCase}$	$\text{LSI}_{Samurai}$	1.00	0.01
$\text{LSI}_{CamelCase}$	LSI_{Oracle}	<0.01	1.00
LSI_{Oracle}	$\text{LSI}_{Samurai}$	<0.01	1.00
$\text{VSM}_{CamelCase}$	$\text{VSM}_{Samurai}$	1.00	0.00
$\text{VSM}_{CamelCase}$	VSM_{Oracle}	<0.01	0.39
VSM_{Oracle}	$\text{VSM}_{Samurai}$	<0.01	0.39
Pooka			
Approach 1	Approach 2	p-value	Cliff's d
$\text{LSI}_{CamelCase}$	$\text{LSI}_{Samurai}$	*0.03*	*0.05*
$\text{LSI}_{CamelCase}$	LSI_{Oracle}	0.40	0.39
LSI_{Oracle}	$\text{LSI}_{Samurai}$	0.98	0.39
$\text{VSM}_{CamelCase}$	$\text{VSM}_{Samurai}$	*<0.01*	*0.13*
$\text{VSM}_{CamelCase}$	VSM_{Oracle}	**<0.01**	**0.72**
VSM_{Oracle}	$\text{VSM}_{Samurai}$	<0.01	0.74
Lynx			
Approach 1	Approach 2	p-value	Cliff's d
$\text{LSI}_{CamelCase}$	$\text{LSI}_{Samurai}$	<0.01	1.00
$\text{LSI}_{CamelCase}$	LSI_{Oracle}	<0.01	1.00
LSI_{Oracle}	$\text{LSI}_{Samurai}$	<0.01	0.91
$\text{VSM}_{CamelCase}$	$\text{VSM}_{Samurai}$	<0.01	0.36
$\text{VSM}_{CamelCase}$	VSM_{Oracle}	**<0.01**	**0.89**
VSM_{Oracle}	$\text{VSM}_{Samurai}$	<0.01	0.92

accuracy of traceability recovery techniques. As it can be noticed from Tables 8.3 and 8.4, manually built oracle and Samurai provided statistically better results in 17% of the cases. In 19% of the cases, there was no effect on the results. However, in 20% of the case, accuracy statistically went down with small and medium effect-size, and it went down with large

effect-size in 44% of the cases.

We concluded that advanced splitting techniques may provide better results than simple techniques in some cases and that, for these analyzed systems, baseline splitting techniques provided, overall, accuracy results almost similar to perfect splitters (*i.e.*, oracles).

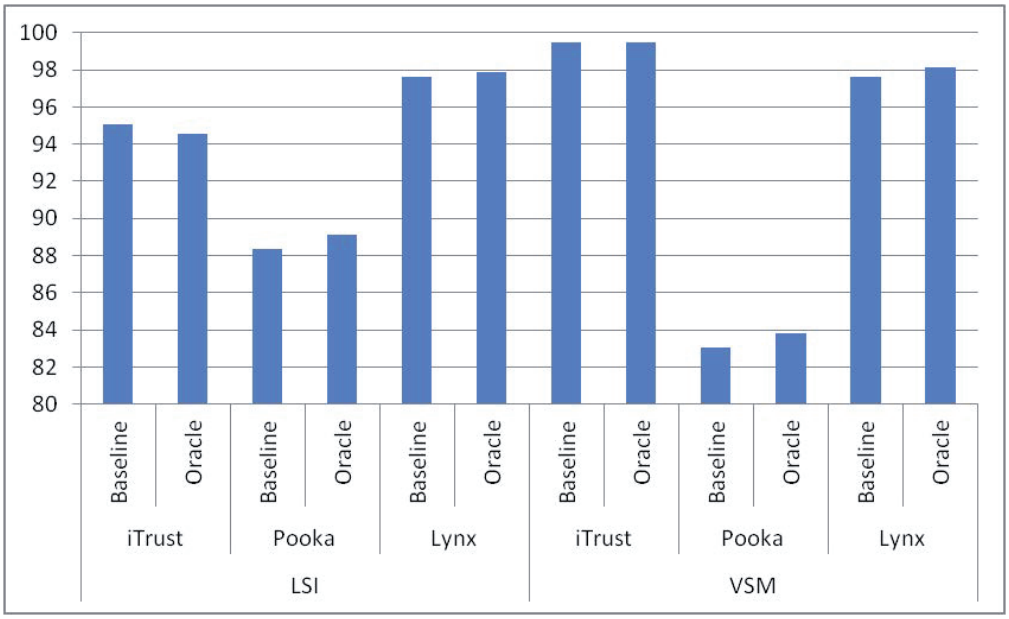


Figure 8.2 Percentage of the traceability links recovered (or missed) by the baseline and oracle.

Figure 8.2 shows, however, that the oracle recovered more links than baseline splitting technique (*i.e.*, CamelCase). Only in the case of iTrust using LSI, oracle missed some links. The latter observation could be justified by the LSI's k value impact since the matrix size was changed after using manually split oracle. In the case of iTrust using VSM, both the oracle and baseline provided the same recall. In addition, we observed that in 67% of times using the oracle recovered more links than the baseline splitting technique.

In summary, we can conclude from this study that advanced identifier splitting approaches can help recover more traceability links than simple techniques in some cases. However, in the general case, a simple identifier splitting approach (*e.g.*, CamelCase) can be sufficient for such a task. We believe, more studies should be performed on several projects belonging to different programming languages to generalize our conclusions.

8.3 Qualitative Analysis

In some cases, we noticed that advanced splitting techniques, and oracles increased the noise in the data. Consequently, it impacted the accuracy of IR techniques. For example,

in Lynx, requirement 534 is "the browser should be able to manage store erase session information". Whereas a C method `LYMain.c.i__nobrowse_fun` is related to browse directories functionality. Baseline splitting techniques could not split the term "nobrowse" and there was no link created between requirement 534 and `LYMain.c.i_nobrowse_fun.txt`. Samurai and manual oracle split the identifier "nobrowse" into "no browse". Consequently, it linked to the file `LYMain.c.i__nobrowse_fun.txt`. However, this is a false positive link. Similar kind of splitting caused higher number of false positive links.

Another observation we had concerns the presence of acronyms and short words in requirements which creates a vocabulary mismatch between requirements containing such words and normalized source code. It is, therefore, clear in the latter cases, that identifier split/expansion would negatively impact the results due to the quality of the considered data.

In summary, we concluded that identifier splitting and expansion techniques can help improve IR techniques. However, this improvement requires data of quality. The latter observation confirms our qualitative findings reported in (Dit *et al.*, 2011) and where the quality and expressiveness of queries describing the features to be located impacted the obtained results. This qualitative result can also be supported by the results obtained by Binkley *et al.* (Binkley *et al.*, 2012) who showed that the improvement brought by identifier splitting in favor of IR-based feature location is most pronounced for shorter, more natural, queries.

8.4 Threats to Validity

Threats to **construct validity** are mainly due to erroneous manual splits and mistakes in traceability recovery oracles. In fact, the semantic of identifiers reflects developer's intent, knowledge, and experience. Thus, identifiers can be split differently from the original developers who created them. To limit this threat, we analyzed several sources of information (*e.g.*, user manuals and online documentation) in addition to source code inspection. We accessed the latter sources of information for about 60% of the identifiers. For what concerns the oracles used to evaluate the studied traceability recovery methods, we used the original traceability matrices provided by the original developers to mitigate such threat.

Threats to **internal validity** are due to the subjectivity in the manual building of the oracle and to the possible biases introduced by manually splitting identifiers. To limit this threat, the oracle was produced by a joint work among the authors.

Threats to **conclusion validity** concern the relations between the treatment and the outcome. Proper tests and their convenient effect-size measures were performed to statistically

reject the null hypotheses. In particular, we used a non-parametric test, *i.e.*, Mann-Whitney, which do not make any assumption on the underlying distributions of the data.

Threats to **external validity** concern the possibility of generalizing our results. To make our results as generalizable as possible, we used different software systems (*i.e.*, iTrust, Lynx, and Pooka) belonging to different application domains and different programming languages, but we cannot be sure that our findings will be valid for other domains, applications, programming languages or software engineering tasks (*i.e.*, code summarization).

8.5 Chapter Summary

Identifier splitting can improve traceability recovery, and thus software maintenance. In this chapter, we presented an empirical study on traceability recovery using two different IR techniques, *i.e.*, LSI and VSM and three identifier splitting techniques: CamelCase, Samurai, and manual splitting of identifiers. These two traceability recovery techniques were evaluated on three open-source systems: iTrust, Pooka, and Lynx, and compared in terms of their precision and recall.

The results of our study indicated that advanced splitting techniques help increase precision and recall in some cases. In addition, our qualitative analysis showed that the impact or improvement brought by such techniques depended on the quality of studied data.

CHAPTER 9

CONCLUSION

The main goal of this research has been to show context-awareness for source code vocabulary normalization, develop context-aware approaches for vocabulary normalization, *i.e.*, TIDIER and TRIS, empirically evaluate them, and finally use one (*i.e.*, TIDIER) to investigate the impact of vocabulary normalization on feature location and traceability recovery.

Software programs, especially legacy systems, are often poorly documented. In this case, the only up-to-date source of information for developers is the source code. In source code, identifiers are key means that support developers during their understanding tasks (Takang *et al.*, 1996; Caprile et Tonella, 1999, 2000; Lawrie *et al.*, 2006, 2007b). The latter unstructured data lends itself for further analysis using IR techniques that can be leveraged to support maintenance tasks such as feature location, traceability recovery, code summarization, etc. The problem is that developers often compose source code identifiers with abbreviated words and acronyms, and do not always use consistent mechanisms and explicit separators when creating identifiers. Developers and–or tools must therefore use the available contextual information to disambiguate concepts conveyed by such identifiers and thus reap the full benefit of IR-based approaches. Unfortunately, there has been really very little empirical evidence on the relevance of context for source code vocabulary normalization and the impact of identifier splitting on software maintenance tasks. To tackle these challenges, we experimentally investigate the effect of context on identifier splitting, we show that source code files are more helpful than functions, and that the application-level contextual information does not help any further. External documentation only helps in some circumstances (Guerrouj *et al.*, 2013b). We also propose context-aware vocabulary normalization approaches, *i.e.*, TIDIER and TRIS. TIDIER is inspired by speech recognition techniques, it exploits contextual information in the form of specialized dictionaries and mimics the process of transforming words via contraction rules. TIDIER has been empirically evaluated on a large set of open-source systems (Guerrouj *et al.*, 2013a). TRIS formalizes the source code vocabulary normalization problem as a graph optimization (minimization) problem to find the optimal path (*i.e.*, optimal splitting-expansion) in an acyclic weighted identifier graph, it uses the relative frequency of source code terms as a local context to determine the most likely identifier splitting-expansion. TRIS relies on a tree-based representation that considerably reduces its computation time, it has also been empirically evaluated on a set of open-source systems (Guerrouj *et al.*, 2012). Finally, we investigate the impact of source code vocabulary

normalization on two IR-based software maintenance tasks, *i.e.*, feature location and traceability recovery. Their results outline potential benefits of developing advanced identifier splitting approaches as they can still be useful in some cases (Dit *et al.*, 2011).

9.1 Summary of Contributions

The main contributions of this thesis are as follows:

- two user studies to investigate the effect of context on source code vocabulary normalization. This work has been published in 2013 in the *Empirical Software Engineering Journal*;
- an approach inspired by speech recognition techniques for source code vocabulary normalization (TIDIER) and its empirical evaluation. This work has been published in 2013 in the *Journal of Software Evolution and Process*;
- an fast implementation for source code vocabulary normalization dealing with normalization as a graph minimization problem (TRIS) and its empirical evaluation. This work has been published in 2012 in the *Proceedings of the* 19^{th} *IEEE Working Conference on Reverse Engineering*;
- an empirical study to analyze the impact of source code vocabulary normalization on two feature location techniques. The first uses IR while the second combines IR and dynamic information. This work has been published in 2011 in the *Proceedings of the* 19^{th} *IEEE International Conference on Program Comprehension*;
- an empirical study to investigate the impact of source code vocabulary normalization on traceability recovery; This work is in preparation for submission to the *Empirical Software Engineering Journal*, 2013.
- an empirical study design to analyze the impact of identifiers styles on software quality by mining software repositories. This is an on-going work undergoing identifier styles used by developers when joining open-source projects and showing whether specific styles introduce bugs into software projects and impact internal quality measures, namely the semantic coupling and cohesion.

Context-Awareness for Source Code Vocabulary Normalization: To correctly split and expand identifiers, especially non-trivial ones, developers and tools need context.

To show the effect of context on vocabulary normalization, we performed two user studies involving 63 participants from the École Polytechnique de Montréal, and used, as objects, a set of identifiers randomly-sampled from a corpus of C open-source programs. In particular, we considered an internal context consisting of the content of functions and source code files in which the identifiers appear, and an external context involving external documentation. The main findings of the two studies indicate that the file-level context is more helpful than the function-level one, and that the application level, did not bring further improvements. Also, in general, external documentation did not introduce significant benefits, likely because the acronyms contained in the identifiers are domain specific. Results also show that the participants' level of English and the knowledge of the domain (Linux in both studies) have a significant effect on vocabulary normalization (Guerrouj *et al.*, 2013b). Overall, the obtained results confirm our belief about the relevance of contextual information in program comprehension. Such information is helpful not only to humans when performing program comprehension tasks, but also to automatic tools that rely on source code lexicon to perform various kinds of tasks, including feature location (Dit *et al.*, 2011; Binkley *et al.*, 2012).

TIDIER: From a comparative analysis of source code vocabulary normalization techniques that exist when we started addressing this problem (*i.e.*, CamelCase and Samurai) and a literature review of the various theories and techniques used for entity recognition, we developed a context-aware approach, TIDIER, which is inspired by speech recognition techniques. TIDIER uses contextual information in the form of specialized dictionaries and assumes the use of transformations rules to create identifiers. We evaluated TIDIER on identifiers randomly-extracted from a large corpus of C programs, and compared it with prior works, *i.e.*, CamelCase and Samurai (Enslen *et al.*, 2009). TIDIER outperformed previous approaches when using context-aware dictionaries built at the level of programs, enriched with domain knowledge, *i.e.*, common acronyms, abbreviations, and C library functions. In addition, it was also able to correctly expand 48% of abbreviations for a set of 73 abbreviations (Guerrouj *et al.*, 2013a). Moreover, TIDIER has been used to assess the impact of vocabulary normalization on feature location (Dit *et al.*, 2011).

TRIS: As a fast and accurate solution for source code vocabulary normalization, we developed TRIS, which deals with normalization as a graph optimization (minimization) problem to find the optimal path (*i.e.*, the optimal normalization) in an acyclic weighted identifier graph. TRIS uses the relative frequency as a local context to select the best possible split-expansion. It has been evaluated on several C, C++, and Java systems and compared with other approaches, *i.e.*, CamelCase, Samurai (Enslen *et al.*, 2009), TIDIER (Guerrouj *et al.*, 2013a), and GenTest (Lawrie *et al.*, 2010). TRIS significantly outperforms CamelCase, Samurai, and TIDIER with a medium to large effect size on C systems. In addition, it shows a

non-statistically significant improvement of 4%, in terms of identifier splitting correctness, over GenTest. TRIS produces one optimal split and expansion fast, using an identifier processing algorithm having a quadratic complexity in the length of the identifier to split/expand (Guerrouj *et al.*, 2012).

Impact of Identifier Splitting on Feature Location: We applied identifier splitting on two FLTs using three splitting strategies (*i.e.*, CamelCase, Samurai, and Oracles built using TIDIER). The first FLT is based on IR while the second uses the combination of IR and dynamic analysis. Our study was applied on two open-source systems, Rhino and jEdit. The results of our empirical evaluation show that FLTs using IR can benefit from better preprocessing algorithms. However, the results for FLT using the combination of IR and dynamic analysis do not show any improvement while using manual splitting, indicating that any preprocessing technique will suffice if execution data is available.

Overall, our results show the need for more sophisticated source code preprocessing techniques as they can still be useful when dynamic information is not available (Dit *et al.*, 2011).

Impact of Identifier Splitting on Traceability Recovery: We applied three identifier splitting strategies (*i.e.*, CamelCase, Samurai, and Oracles built using TIDIER) on two traceability recovery techniques. The first technique uses LSI while the second is based on VSM. These two traceability recovery techniques were evaluated on three open-source systems: iTrust, Pooka, and Lynx, and compared in terms of their precision and recall. The results of our study highlight that advanced splitting techniques help increase precision and recall in some cases but, in general, they perform the same as the simple CamelCase on the studied systems. In addition, our qualitative analysis showed that the impact or improvement brought by such techniques depend on the quality of studied data. We are currently performing more studies on other systems using other identifier splitting and expansion techniques to generalize our conclusions on the impact of identifier splitting and expansion on traceability recovery and feature location.

In summary, we bring empirical evidences of the relevance of context for source code vocabulary normalization. We also propose two novel context-aware approaches and their practical implementations for the normalization task. Finally, we investigate the impact of identifier splitting on feature location and traceability recovery. The obtained results are promising and can be used by both practitioners and researchers.

9.2 Limitations

Despite the above promising results, our findings are also exposed to some limitations such as:

Limitation of User Studies on the Effect of Contexts on Vocabulary Normalization: Our controlled experiments were performed in an academic context, *i.e.*, with participants belonging to a population of Canadian students (Bachelor, Master, Ph.D.) and post-docs. Many of them already had previous industrial experience. However, real developers can possibly perform differently to this population. It is, therefore, possible to obtain different results in industrial settings with developers having different skills and levels of experience.

Limitation of TIDIER: TIDIER performs well, however, it has some limitations. First, it implements a set of word transformation rules that we assume the most used by software developers when creating identifiers. These word transformations may not be helpful for other software such as mathematical one where a number of variables such as i, j, and k are declared. Also, TIDIER has a cubic complexity in the number of characters composing the identifier, words in the dictionary, and maximum number of characters composing dictionary words. Thus, it may require a considerable amount of time when dealing with large software systems.

Limitation of TRIS: TRIS is accurate and faster than TIDIER. However, it shares with TIDIER the inconvenience of being based on word transformations rules that may be not helpful to expand some identifiers. In addition, TRIS is sensitive to the quality of the used software application dictionaries.

Limitation of our Investigation on the Impact of Identifier Splitting on Feature Location: We observed from the exploration of the analyzed data that the quality and expressiveness of queries impact the results of identifier splitting on feature location. In fact, in some cases, we found that queries contain identifiers used just for informal communication between developers, and no matter what splitting/expansion technique is used, these identifiers provided no useful information, because they only appear in the query vocabulary, and do not appear at all in the source code vocabulary. Examples of such identifiers include words that are common in communication, name of developers, etc. Thus, the effect of identifier splitting could be hidden by such a factor. The impact of queries has been also highlighted by Binkley *et al.* who show that vocabulary normalization improve feature location techniques that use short, more natural queries (Binkley *et al.*, 2012). Recently, the quality of IR queries has been addressed by Haiduc *et al.* (Haiduc *et al.*, 2013b) who propose approaches for the prediction of the quality of IR queries and techniques for their reformulation (Haiduc *et al.*,

2013a).

Limitation of our Empirical Study on the Impact of Identifier Splitting on Traceability Recovery: Different IR techniques showed different results when applying identifier splitting on traceability recovery. The latter observation means that the type of the used IR technique can impact the results of such kind of studies. As for feature location, the quality of the data play also a role and impact the quality of the obtained results. In fact, we noticed, in some cases, a vocabulary mismatch between requirements and source code due to the presence of short acronyms in software requirements. Hence, it is clear that in the latter case the splitting/expansion of identifiers will negatively impact the results of any traceability recovery technique since it will introduce noise (false positives). Thus, the quality of software artefacts to be linked may influence the results of such an empirical investigation.

9.3 Future Work

Future work should be devoted to further experiments on more larger software systems, and to the application of our techniques/results in industrial settings. Below, we describe how we plan to extend the work presented in this thesis:

Context-Awareness for Vocabulary Normalization: We aim at replicating this experimental study using eye-tracking tools, to better observe the way developers investigate the contextual information when performing identifier splitting and expansion. Another research direction we wish to explore is to implement a context-aware approach and tool that—within an Integrated Development Environment—support developers program understanding, not only by suggesting possible identifier splitting/expansions, but also by providing contextual information useful when reading and understanding an identifier. We also would like to involve people from industry in this kind of study instead of being limited to academic contexts.

TIDIER: In the future, it could be interesting to implement other word transformations based on surveys conducted with software developers. We also want to improve the string-edit distance guiding TIDIER, speed up its algorithm, and use semantic information.

TRIS: We would like to extend TRIS evaluation to larger systems using other words transformations rules. In addition, we plan to compare it to more recent approaches such as (Lawrie et Binkley, 2011) and LINSEN (Corazza *et al.*, 2012).

Impact of Identifier Splitting on Feature Location: We plan to extend our evaluation to other software systems belonging to other systems such as C, C++ or COBOL. In addition, we plan to investigate the impact of other clever identifier splitting/expansion techniques such as Normalize (Lawrie et Binkley, 2011) to analyze whether they will show further improvements.

Impact of Identifier Splitting on Traceability Recovery: As for feature location, it could be interesting to evaluate the studied traceability recovery techniques on large software systems written in programming languages different from Java. The reason is that Java developers adhere to naming conventions and identifiers creation rules and, thus, the improvement brought by advanced splitting techniques may be equal to the one brought by any other simple CamelCase splitter (Dit *et al.*, 2011).

Mining Software Repositories to Study the Impact of Identifier Style on Software Quality: Several research works (De Lucia *et al.*, 2006; Lawrie *et al.*, 2007b,a; Abebe *et al.*, 2012) have tried to assess the quality of identifiers. However, there is little empirical evidence on the impact of identifier style on software quality.

To address this challenge, we are currently conducting an empirical study where the context consists of six (Java and C) open-source projects: ArgoUML[1], Ant[2], Apache[3], Samba[4], Hibernate[5], and PostgrSQL[6].

The main research questions that we are addressing are as follows:

1. **RQ1**: How open-source projects are written in terms of identifier styles?
2. **RQ2**: Do open-source developers adhere to the style of the project they join when naming identifiers or do they bring their own style?
3. **RQ3**: How identifier style vary with respect to the type of identifier?
4. **RQ4**: Do developers' characteristics (*i.e.*, experience, activity focus, etc.) lead them to adopt a specific identifier style?
5. **RQ5**: Does a specific identifier style (*e.g.*, abbreviations or acronyms) introduce bugs in software systems?
6. **RQ6**: Does a specific identifier style (*e.g.*, abbreviations or acronyms) impact internal quality attributes, in particular, the semantic coupling between classes and the semantic cohesion between methods of a project?
7. **RQ7**: Is source code vocabulary normalization able to help improve cases of poor semantic coupling and cohesion?

[1] `http://argouml.tigris.org/`
[2] `http://ant.apache.org/`
[3] `http://www.apache.org/`
[4] `http://www.samba.org/`
[5] `http://www.hibernate.org/`
[6] `http://www.postgresql.org/`

To address these research questions, we follow the methodology described below.

For each project, we extract, using the distributed version control Git[7], the identifiers used in the projects and also the identifiers added by each developer. We classify identifiers according to whether they are names of methods, local variables, parameters, or attributes. Then, we infer the identifier style of the projects and the identifier style of developers. We do so for the four type of identifiers considered, *i.e.*, name of methods, local variables, attributes, and parameters. We infer the identifier style using a statistical model, namely, the Hidden Markov Model (HMM) (Baggenstoss, 2001). We choose the Baum-Welch (Baum, 1970) algorithm to train our data sets and to define parameters of our HMM. We use HMM models because they are especially known for their application in temporal pattern recognition such as speech, handwriting, and gesture recognition (Juang et Rabiner, 1991; Starner et Pentl, 1995), part-of-speech tagging (Thede et Harper, 1999), musical score following (Pardo et Birmingham, 2005), and bio-informatics analyses, such as the CpG island detection and splice site recognition (R. Durbin et Mitchison, 1998) (**RQ1**, **RQ2**, and **RQ3**).

We also extract information about developers of the projects such as the number of files they changed, number of commits they did, their experience in the project, etc. We define their experience as the difference between the date of the last and first commits they did. In addition, we extract information about the number of bugs they introduced (if any) and the summary of the introduced bugs. Furthermore, we compute the activity focus of developers as defined by Bird *et al.* (Bird *et al.*, 2008). The activity focus is the average directory tree distance between all pairs of files that are committed to by developers within each team. The aim is to see whether the adoption of a specific identifier style is related to developers' characteristics or not (**RQ4**).

The next challenge we are going to address is whether a specific identifier style (*e.g.*, abbreviations or acronyms) introduces bugs in the systems and whether it impacts internal quality measures, namely, semantic coupling and cohesion (Bavota *et al.*, 2013) (**RQ5** and **RQ6**). Finally, we would like to see if normalizing source code identifiers using TRIS (Guerrouj *et al.*, 2012) or Normalize (Lawrie et Binkley, 2011) in cases of poor semantic coupling and cohesion can help improve these internal quality metrics (**RQ7**).

[7] `http://git-scm.com/`

9.4 Publications

Our publications related to this thesis are as follows:

Journal articles

- **Latifa Guerrouj**, Massimilano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. *An Experimental Investigation on the Effects of Contexts on Source Code Identifiers Splitting and Expansion.* Empirical Software Engineering Journal (EMSE). DOI: 10.1007/s10664-013-9260-1, to appear (2013).

- **Latifa Guerrouj**, Massimilano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. *TIDIER: An Identifier Splitting Approach Using Speech Recognition Techniques.* Journal of Software Evolution and Process (JSEP). 25(6): 569-661 (2013).

Conference articles

- **Latifa Guerrouj**, Philippe Galinier, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Massimiliano Di Penta. *TRIS: a Fast and Accurate Identifiers Splitting and Expansion Algorithm.* Proceedings of the 19^{th} IEEE Working Conference on Reverse Engineering (WCRE), October 2012.

- Bogdan Dit, **Latifa Guerrouj**, Denys Poshyvanyk, Giuliano Antoniol. *Can Better Identifier Splitting Techniques Help Feature Location?* Proceedings of the 19^{th} IEEE International Conference on Program Comprehension (ICPC), June 2011.

- Nioosha Madani, **Latifa Guerrouj**, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, Giuliano Antoniol. *Recognizing Words from Source Code Identifiers Using Speech*

Recognition Techniques. Proceedings of the 14^{th} IEEE European Conference on Software Maintenance and Reengineering (CSMR), Mars 2010. This paper received the Best Paper award of CSMR'10.

- **Latifa Guerrouj**. *Normalizing Source Code Vocabulary to Enhance Program Comprehension and Software Quality*. Proceedings of the 35^{th} ACM International Conference on Software Engineering (ICSE), May 2013.

- **Latifa Guerrouj**. *Automatic Derivation of Concepts Based on the Analysis of Source Code Identifiers*. Proceedings of the 17^{th} Working Conference on Reverse Engineering (WCRE), October 2012.

During my Ph.D., I co-organized the 2^{nd} Workshop on Mining Unstructured Data (MUD'12) collocated with the 19^{th} Working Conference on Reverse Engineering (WCRE'12). The workshop involved topics related to my dissertation:

- Alberto Bacchelli, Nicolas Bettenburg, **Latifa Guerrouj**. *Mining Unstructured Data because "Mining Unstructured Data is Like Fishing in Muddy Waters!"*. Proceedings of the 19^{th} Working Conference on Reverse Engineering (WCRE), October 2012.

BIBLIOGRAPHY

ABEBE, S. L., ARNAOUDOVA, V., TONELLA, P., ANTONIOL, G. et GUÉHÉNEUC, Y.-G. (2012). Can Lexicon Bad Smells Improve Fault Prediction? *Proceedings of the 19th Working Conference on Reverse Engineering*. pp. 235–244.

ABEBE, S. L., HAIDUC, S., MARCUS, A., TONELLA, P. et ANTONIOL, G. (2008). Analyzing the Evolution of the Source Code Vocabulary. *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*. pp. 189–198.

ALI, N., GUÉHÉNEUC, Y.-G. et ANTONIOL, G. (2013). Trustrace: Mining Software Repositories to Improve the Accuracy of Requirement Traceability Links. *IEEE Transactions on Software Engineering*, vol. 39, pp. 119–146.

ANQUETIL, N. et LETHBRIDGE, T. (1998). Assessing the Relevance of Identifier Names in a Legacy Software System. *Proceedings of CASCON*. pp. 213–222.

ANTONIOL, G., CANFORA, G., CASAZZA, G., LUCIA, A. D., et MERLO, E. (2002). Recovering Traceability Links Between Code and Documentation. *IEEE Transactions on Software Engineering*, vol. 28, pp. 970–983.

ANTONIOL, G., CAPRILE, B., POTRICH, A. et TONELLA, P. (2000). Design Code Traceability for Object-Oriented Systems. *Annals of Software Engineering*, vol. 9, pp. 35–58.

ANTONIOL, G., CAPRILE, B., POTRICH, A. et TONELLA, P. (2001). Design Code Traceability Recovery: Selecting the Basic Linkage Properties. *Science of Computer Programming*, vol. 40, pp. 213–234.

ANTONIOL, G. et GUÉHÉNEUC, Y.-G. (2005). Feature Identification: A Novel Approach and a Case Study. *Proceedings of the 21st International Conference on Software Maintenance*. pp. 357–366.

BAEZA-YATES, R. et RIBEIRO-NETO, B. (1999). *Modern Information Retrieval*. Addison-Wesley.

BAEZA-YATES, R. A. et PERLEBERG, C. H. (1992). Fast and Practical Approximate String Matching. *Combinatorial Pattern Matching, Third Annual Symposium*. pp. 185–192.

BAGGENSTOSS, P. (2001). A Modified Baum Welch Algorithm for Hidden Markov Models with Multiple Observation Spaces. *IEEE Transactions on Speech and Audio Processing*, vol. 9, pp. 411–416.

BAKER, R. D. (1995). Modern permutation test software. *Randomization Tests*.

BASILI, V., CALDIERA, G. et ROMBACH, D. H. (1994). *The Experience Factory Encyclopedia of Software Engineering*. John Wiley and Son.

BAUM, L. E. (1970). A Maximization Technique Occurring in the Statistical Analysis of Probabilistic Functions of Markov Chains. *Annals of Mathematical Statistics*, vol. 41, pp. 164–171.

BAVOTA, G., DIT, B., OLIVETO, R., DI PENTA, M., POSHYVANYK, D. et DE LUCIA, A. (2013). An Empirical Study on the Developers Perception of Software Coupling. *Proceedings of the 35th International Conference on Software Engineering*. pp. 692–701.

BIGGERSTAFF, T. J., MITBANDER, B. G. et WEBSTER, D. E. (1994). Program Understanding and the Concept Assignment Problem. *Communications of the ACM*, vol. 37, pp. 72–82.

BINKLEY, D., DAVIS, M., LAWRIE, D., MALETIC, J. I., MORRELL, C. et SHARIF, B. (2013). The Impact of Identifier Style on Effort and Comprehension. *Empirical Software Engineering*, vol. 2, pp. 219–276.

BINKLEY, D., DAVIS, M., LAWRIE, D. et MORRELL, C. (2009). To Camelcase or Under_score. *Proceedings of the 17th International Conference on Program Comprehension*. pp. 158–167.

BINKLEY, D., DAWN, D. L. et UEHLINGER, C. (2012). Vocabulary Normalization Improves IR-Based Concept Location. *Proceedings of the 28th International Conference on Software Maintenance*, vol. 41, pp. 588–591.

BIRD, C., PATTISON, D. S., D'SOUZA, R. M., FILKOV, V. et DEVANBU, P. T. (2008). Latent Social Structure in Open-Source Projects. *Proceedings of the 16th International Symposium on Foundations of Software Engineering*. pp. 24–35.

BLAND, J. M. (2000). *An Introduction to Medical Statistics (Third Edition)*. Oxford University Press.

BRANTS, T. et FRANZ, A. (2006). *Web 1t 5-gram version 1*. Linguistic Data Consortium, Philadelphia.

CANFORA, G., CERULO, L. et DI PENTA, M. (2009). Tracking Your Changes: a Language-Independent Approach. *IEEE Transactions on Software Engineering*, vol. 27, pp. 50–57.

CANFORA, G., CERULO, L. et DI PENTA, M. (2010). An Empirical Study on the Maintenance of Source Code Clones. *Empirical Software Engineering*, vol. 15, pp. 1–34.

CAPRILE, B. et TONELLA, P. (1999). Nomen est Omen: Analyzing the Language of Function Identifiers. *Proceedings of the 6th Working Conference on Reverse Engineering*. pp. 112–122.

CAPRILE, B. et TONELLA, P. (2000). Restructuring Program Identifier Names. *Proceedings of the International Conference on Software Maintenance*. pp. 97–107.

COHEN, J. (1988). *Statistical power analysis for the behavioral sciences*. Lawrence Earlbaum Associates.

CONOVER, W. J. (1998). *Practical Nonparametric Statistics*. Wiley.

CORAZZA, A., MARTINO, S. D. et MAGGIO, V. (2012). LINSEN: An Efficient Approach to Split Identifiers and Expand Abbreviations. *Proceedings of the 28th International Conference of Software Maintenance*. pp. 233–242.

CORMEN, T. H., LEISERSON, C. E. et RIVEST, R. L. (1990). *Introductions to Algorithms*. MIT Press.

CORNELISSEN, B., ZAIDMAN, A., DEURSEN, A. V., MOONEN, L. et KOSCHKE, R. (2009). A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering*, vol. 35, pp. 684–702.

DE LUCIA, A., DI PENTA, M. et OLIVETO, R. (2010). Improving Source Code Lexicon via Traceability and Information Retrieval. *IEEE Transactions on Software Engineering*, vol. 37, pp. 205–226.

DE LUCIA, A., DI PENTA, M., OLIVETO, R., PANICHELLA, A. et PANICHELLA, S. (2011). Improving IR-based Traceability Recovery Using Smoothing Filters. *Proceedings of the 19th International Conference on Program Comprehension*. pp. 21–30.

DE LUCIA, A., DI PENTA, M., OLIVETO, R. et ZUROLO, F. (2006). Improving Comprehensibility of Source Code via Traceability Information: a Controlled Experiment. *Proceedings of 14th International Conference on Program Comprehension*. pp. 317–326.

DE LUCIA, A., FASANO, F., OLIVETO, R. et TORTORA, G. (2004). Enhancing an artefact management system with traceability recovery features. *Proceedings of the 20th International Conference on Software Maintenance*. pp. 306–315.

DE LUCIA, A., FASANO, F., OLIVETO, R. et TORTORA, G. (2007). Recovering Traceability Links in Software Artefact Management Systems Using Information Retrieval Methods. *ACM Transactions on Software Engineering and Methodology*, vol. 16, pp. 1–50.

DEERWESTER, S., DUMAIS, S. T., FURNAS, G. W., LANDAUER, T. K. et HARSHMAN, R. (1990). Indexing by Latent Semantic Analysis. *IEEE Transactions on Software Engineering*, vol. 41, pp. 391–407.

DEIßENBÖCK, F. et PIZKA, M. (2005). Concise and Consistent Naming. *Proceedings of the 13th International International Workshop on Program Comprehension*. pp. 97–106.

DEMEYER, S., DUCASSE, S. et NIERSTRASZ, O. (2000). Finding Refactorings via Change Metrics. *Proceedings of the 2000 Conference on Object-Oriented Programming Systems Languages and Applications*. pp. 166–177.

DIT, B., GUERROUJ, L., POSHYVANYK, D. et ANTONIOL, G. (2011). Can Better Identifier Splitting Techniques Help Feature Location? *Proceedings of the 19th International Conference on Program Comprehension*. pp. 11–20.

EADDY, M., AHO, A., ANTONIOL, G. et GUÉHÉNEUC, Y.-G. (2008a). CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis. *Proceedings of 16th International Conference on Program Comprehension*. pp. 53–62.

EADDY, M., ZIMMERMANN, T., SHERWOOD, K., GARG, V., MURPHY, G., NAGAPPAN, N. et AHO, A. V. (2008b). Do Crosscutting Concerns Cause Defects. *IEEE Transactions on Software Engineering*, vol. 34, pp. 497–515.

EISENBARTH, T., KOSCHKE, R. et SIMON, D. (2003). Locating Features in Source Code. *IEEE Transactions on Software Engineering*, vol. 29, pp. 210–224.

ENSLEN, E., HILL, E., POLLOCK, L. et SHANKER, K. V. (2009). Mining Source Code to Automatically Split Identifiers for Software Analysis. *Proceedings of the 6th International Working Conference on Mining Software Repositories*. pp. 16–17.

FEILD, H., BINKLEY, D. et LAWRIE, D. (2006). An Empirical Comparison of Techniques for Extracting Concept Abbreviations from Identifiers. *Proceedings of IASTED International Conference on Software Engineering and Applications*.

GAO, J. N. J., HE, H., CHEN, W. et ZHOU, M. (2002). Resolving Query Translation Ambiguity Using a Decaying Co-occurrence Model and Syntactic Dependence Relations. *Proceedings of the 25th Annual International SIGIR conference on Research and development in information retrieval*. pp. 183–190.

GOTEL, O. et FINKELSTEIN, A. (1993). *An Analysis of the Requirements Traceability Problem*. Department of Computing Imperial College. TR-93-41.

GOTEL, O. et FINKELSTEIN, A. (1994). An Analysis of the Requirements Traceability Problem. *Proceedings of the 1st International Conference on Requirements Engineering*. pp. 94–101.

GRANT, S., CORDY, J. R. et SKILLICORN, D. B. (2008). Automated Concept Location Using Independent Component Analysis. *Proceedings of the 15th Working Conference on Reverse Engineering*. pp. 138–142.

GRISSOM, R. J. et KIM, J. J. (2005). *Effect sizes for research: A broad practical approach*. Addison-Wesley.

GUERROUJ, L., DI PENTA, M., ANTONIOL, G. et GUÉHÉNEUC, Y.-G. (2013a). TIDIER: An Identifier Splitting Approach using Speech Recognition Techniques. *Journal of Software Evolution and Process*, vol. 25, pp. 569–661.

GUERROUJ, L., DI PENTA, M., GUÉHÉNEUC, Y.-G. et ANTONIOL, G. (2013b). An Experimental Investigation on the Effects of Context on Source Code Identifiers Splitting and Expansion. *Empirical Software Engineering. Doi: 10.1016/S0164-1212(00)00029-7*.

GUERROUJ, L., GALINIER, P., GUÉHÉNEUC, Y.-G., ANTONIOL, G. et DI PENTA, M. (2012). TRIS: A Fast and Accurate Identifiers Splitting and Expansion Algorithm. *Proceedings of the 19th Working Conference on Reverse Engineering*. pp. 103–112.

HAIDUC, S., BAVOTA, G., MARCUS, A., OLIVETO, R., DE LUCIA, A. et MENZIES, T. (2013a). Automatic Query Reformulations for Text Retrieval in Software Engineering. *Proceedings of the 35th International Conference on Software Engineering*. pp. 842–851.

HAIDUC, S., ROSA, G. D., BAVOTA, G., OLIVETO, R., DE LUCIA, A. et MARCUS, A. (2013b). Query Quality Prediction and Reformulation for Source Code Search: the Refoqus Tool. *Proceedings of the 35th International Conference on Software Engineering*. pp. 1307–1310.

HAYES, J. H., DEKHTYAR, A., SUNDARAM, S. et HOWARD, S. (2004). Helping Analysts Trace Requirements: An Objective Look. *Proceedings of the 12th Requirements Engineering Conference*. pp. 249–261.

HILL, E., BINKLEY, D., LAWRIE, D., POLLOCK, L. et VIJAY-SHANKER, K. (2011). *An Empirical Comparison of Identifier Splitting Techniques*. Loyola University. TRLoyola928.

HOLM, S. (1979). A Simple Sequentially Rejective Bonferroni Test Procedure. *Scandinavian Journal of Statistics*, vol. 6, pp. 65–70.

JUANG, B. H. et RABINER, L. R. (1991). Hidden Markov Models for Speech Recognition. *Technometrics*, vol. 33, pp. 251–272.

KERSTEN, M. et MURPHY, G. C. (2006). Using Task Context to Improve Programmer Productivity. *Proceedings of the 14th International Symposium on Foundations of Software Engineering*. pp. 1–11.

LAWRIE, D. et BINKLEY, D. (2011). Expanding Identifiers to Normalize Source Code Vocabulary. *Proceedings of the 27th International Conference on Software Maintenance*. pp. 113–122.

LAWRIE, D., FEILD, H. et BINKLEY, D. (2006). Syntactic Identifier Conciseness and Consistency. *Proceedings of the 6th International Workshop on Source Code Analysis and Manipulation*. pp. 139–148.

LAWRIE, D., FEILD, H. et BINKLEY, D. (2007a). Quantifying Identifier Quality: an Analysis of Trends. *Empirical Software Engineering* , vol. 12, pp. 359–388.

LAWRIE, D., MORRELL, C., FEILD, H. et BINKLEY, D. (2007b). Effective Identifier Names for Comprehension and Memory. *Innovations in Systems and Software Engineering*, vol. 3, pp. 303–318.

LAWRIE, D. J., BINKLEY, D. et MORRELL, C. (2010). Normalizing Source Code Vocabulary. *Proceedings of the 17th Working Conference on Reverse Engineering*. pp. 112–122.

LEVENSHTEIN, V. I. (1966). Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Cybernetics and Control Theory*, vol. 10, pp. 707–710.

LIU, D., MARCUS, A., POSHYVANYK, D. et RAJLICH, V. (2007). Feature Location via Information Retrieval-based Filtering of a Single Scenario Execution Trace. *Proceedings of the 22nd International conference on Automated Software Engineering*. pp. 234–243.

LUKINS, S., KRAFT, N. A. et ETZKORN, L. (2010). Bug Localization Using Latent Dirichlet Allocation. *Proceedings of the 15th Working Conference on Reverse Engineering*. pp. 972–990.

M-A.D. STOREY, F.D. FRACCHIA, H. M. (1999). Cognitive Design Elements to Support the Construction of a Mental Model During Software Exploration. *Journal of Systems and Software*, vol. 44, pp. 171–185.

MADANI, N., GUERROUJ, L., DI PENTA, M., GUÉHÉNEUC, Y.-G. et ANTONIOL, G. (2010). Recognizing Words from Source Code Identifiers using Speech Recognition Techniques. *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*. pp. 68–77.

MADER, P., GOTEL, O. et PHILIPPOW, I. (2009). Getting Back to Basics: Promoting the Use of a Traceability Information Model in Practice. *Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering*. pp. 21–25.

MALETIC, J. I. et COLLARD, M. L. (2009). Tql: A Query Language to Support Traceability. *Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering*. pp. 16–20.

MARCUS, A., FENG, L. et MALETIC, J. (2003). Syntactic Identifier Conciseness and Consistency. *Proceedings of the 2003 Symposium on Software Visualization*. pp. 27–ff.

MARCUS, A., MALETIC, J. et SERGEYEV, A. (2005). Recovery of Traceability Links Between Software Documentation and Source Code. *International Journal of Software Engineering and Knowledge Engineering*, vol. 15, pp. 811–836.

MARCUS, A. et MALETIC, J. I. (2003). Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing. *Proceedings of the 25th International Conference on Software Engineering*. pp. 125–137.

MARCUS, A., POSHYVANYK, D. et FERENC, R. (2008). Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, vol. 34, pp. 287–300.

MARCUS, A., SERGEYEV, A., RAJLICH, V. et MALETIC, J. (2004). An Information Retrieval Approach to Concept Location in Source Code. *Proceedings of the 11th Working Conference on Reverse Engineering*. pp. 214–223.

MAYRHAUSER, A. V. et VANS, A. M. (1995). Program Comprehension During Software Maintenance and Evolution. *Computer*, vol. 28, pp. 44–55.

MERLO, E., MCADAM, I. et MORI, R. D. (2003). Feed-Forward and Recurrent Neural Networks for Source Code Informal Information Analysis. *Journal of Software Maintenance*, vol. 15, pp. 205–244.

MICHALEWICZ, Z. et FOGEL, D. B. (2004). *How to Solve It: Modern Heuristics - (2nd edition)*. SV.

NEY, H. (1984). The Use of a One-stage Dynamic Programming Algorithm for Connected Word Recognition. *IEEE Transactions on Acoustics Speech and Signal Processing*, vol. 32, pp. 263–271.

PANIS, M. C. (2010). Successful Deployment of Requirements Traceability in a Commercial Engineering organization...Really. *Proceedings of the 18th International Requirements Engineering Conference*. pp. 303–307.

PARDO, B. et BIRMINGHAM, W. (2005). Modeling Form for On-line Following of Musical Performances. *Proceedings of the 20th national conference on Artificial intelligence*. pp. 1018–1023.

PORTER, M. (1980). An Algorithm for Suffix Stripping. *Journal of the American Society for Information Science*, vol. 14, pp. 130–137.

POSHYVANYK, D., GUÉHÉNEUC, Y.-G., MARCUS, A., ANTONIOL, G. et RAJLICH, V. (2007). Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *IEEE Transactions on Software Engineering*, vol. 33, pp. 420–432.

POSHYVANYK, D. et MARCUS, A. (2006). The Conceptual Coupling Metrics for Object-Oriented Systems. *Proceedings of 22nd International Conference on Software Maintenance*. pp. 469–478.

R. DURBIN, S. R. EDDY, A. K. et MITCHISON, G. (1998). *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press.

RAJLICH, V. et WILDE, N. (2002). The Role of Concepts in Program Comprehension. *Proceedings of 10th International Workshop on Program Comprehension*. pp. 271–280.

REVELLE, M., DIT, B. et POSHYVANYK, D. (2010). Using Data Fusion and Web Mining to Support Feature Location in Software. *Proceedings of the 18th International Conference on Program Comprehension*. pp. 14–23.

REVELLE, M. et POSHYVANYK, D. (2009). An Exploratory Study on Assessing Feature Location Techniques. *Proceedings of the 17th International Conference on Program Comprehension*. pp. 218–222.

RICCA, F., DI PENTA, M., TORCHIANO, M., TONELLA, P. et CECCATO, M. (2010). How Developers' Experience and Ability Influence Web Application Comprehension Tasks Supported by UML Stereotypes: A Series of Four Experiments. *IEEE Transactions on Software Engineering*, vol. 36, pp. 96–118.

ROBILLARD, M. P., COELHO, W. et MURPHY, G. C. (2004). How Effective Developers Investigate Source Code: An Exploratory Study. *IEEE Transactions on Software Engineering*, vol. 30, pp. 889–903.

SAKOE, H. et CHIBA, S. (1978). Dynamic Programming Algorithm Optimization for Spoken word Recognition. *IEEE Transactions on Acoustics Speech and Signal Processing*, vol. 26, pp. 43–49.

SALTON, G. et BUCKLEY, C. (1988). Term-Weighting Approaches in Automatic Text Retrieval. *Information Processing and Management*, vol. 24, pp. 513–523.

SHARIF, B. et MALETIC, J. (2010). An Eye tracking Study on CamelCase and under_score Identifier Styles. *Proceedings of the 18th International Conference on Program Comprehension*. pp. 196–205.

SHEPHERD, D., FRY, Z., GIBSON, E., POLLOCK, L. et VIJAY-SHANKER, K. (2007). Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns. *Proceedings of the 6th International Conference on Aspect-oriented software development*. pp. 212–224.

SHERBA, S. A. et ANDERSON, K. M. (2003). A Framework for Managing Traceability Relationships Between Requirements and Architectures. *Second International Software Requirements to Architectures Workshop (STRAW 03)*. pp. 150–156.

SHESKIN, D. (2007). *Handbook of Parametric and Non-Parametric Statistical Procedures (fourth edition)*. Chapman and All.

SILLITO, J., MURPHY, G. C. et VOLDER, K. D. (2008). Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering*, vol. 34, pp. 434–451.

SOLOWAY, E., BONAR, J. et EHRLICH, K. (1983). Cognitive Strategies and Looping Constructs: an Empirical Study. *Commununications of the ACM*, vol. 26, pp. 853–860.

STARNER, T. et PENTL, A. (1995). Visual Recognition of American Sign Language Using Hidden Markov Models. *International Workshop on Automatic Face and Gesture Recognition*. pp. 189–194.

TAKANG, A., GRUBB, P. A. et MACREDIE, R. D. (1996). The Effects of Comments and Identifier Names on Program Comprehensibility: an Experiential Study. *Journal of Program Languages*, vol. 3, pp. 143–167.

TEAM, R. C. (2012). *A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing.

THEDE, S. M. et HARPER, M. P. (1999). A Second-Order Hidden Markov Model for Part-of-Speech Tagging. *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics on Computational Linguistics*. pp. 175–182.

WOHLIN, C., RUNESON, P., HOST, M., OHLSSON, C., REGNELL, B. et WESSLSEN, A. (2000). *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers.

ZOU, X., SETTIMI, R. et CLELAND-HUANG, J. (2010). Improving Automated Requirements Trace Retrieval: a Study of Term-Based Enhancement Methods. *Empirical Software Engineering*, vol. 15, pp. 119–146.

Appendix A

TIDIER descriptive statistics of precision and recall

Table A.1 TIDIER, Samurai, and CamelCase descriptive statistics of precision.

Method	Dictionary	1Q	Median	3Q	Mean	σ
CamelCase		0.00	0.50	1.00	0.45	0.44
Samurai		0.00	0.50	1.00	0.50	0.43
TIDIER	English dictionary	0.00	0.25	0.67	0.38	0.41
	English dict. + domain kn.	0.25	0.50	1.00	0.58	0.40
	WordNet	0.00	0.50	1.00	0.43	0.41
	Function	0.00	0.00	0.00	0.14	0.28
	File	0.00	0.00	0.50	0.30	0.37
	Application	0.00	0.50	1.00	0.51	0.40
	Application + domain kn.	0.50	1.00	1.00	0.72	0.37

Table A.2 TIDIER, Samurai, and CamelCase descriptive statistics of recall.

Method	Dictionary	1Q	Median	3Q	Mean	σ
CamelCase		0.00	0.50	1.00	0.44	0.44
Samurai		0.00	0.50	1.00	0.50	0.43
TIDIER	English dictionary	0.00	0.33	1.00	0.40	0.42
	English dict. + domain kn.	0.33	0.67	1.00	0.64	0.39
	WordNet	0.00	0.50	1.00	0.45	0.41
	Function	0.00	0.00	0.00	0.14	0.29
	File	0.00	0.00	0.60	0.33	0.39
	Application	0.00	0.50	1.00	0.55	0.41
	Application + domain kn.	0.50	1.00	1.00	0.75	0.36

Appendix B

User studies on context and vocabulary normalization: characteristics of applications, identifers oracle and box plots of precision and recall

Table B.1 reports the characteristics of the 34 applications from which the 50 identifiers used in our study were sampled.

Table B.1 Applications from which we sampled the identifiers used in Exp I and Exp II.

Application	Description	Files	Comments	Size (KLOC)	Sampled Identifiers
gnuradio-3.2.2	Multimedia	1,520	51,515	96,352	dbsm_start
acct-6.5.1	Login/accounting utils	59	2,854	8,887	acct_file
binutils-2.20	Unix utils	1,644	207,028	1,071,651	arm_reg_parse, dupok, gmon_io_write
cpio-2.9	Archiving util	204	9,217	30,309	hol_cluster_base
dico-2.0	DICT server	242	10,546	56,823	argv_to_scm, assoc
emacs-19.34	Editor	347	15,151	18,633	dfp, load_scnptr
freebsd-8.0.0	OS kernel	21,609	1,884,742	5,822,143	rrt_prev
g77-0.5.19.1	Fortran to C translator	237	14,105	100,451	FFEBAD_severityFATAL
gcal-3.01	Calendar	74	15,855	61,824	HD_SYLVESTER
gcc-2.7.2.2	C compiler	690	98,290	331,030	nvtbl
gcl-2.6.7	Common Lisp interpreter	1,492	90,899	331,940	bfd_abs_section_ptr, internal_auxent
glibc-2.0.4	C library	2,761	69,425	167,685	f_getlk
gmp-4.3.1	GNU Multiple Precision Arithmetic Library (GMP)	706	19,575	81,931	GMP_NUMB_MAX
gnubatch-1.1	Batch scheduling	511	10,593	11,2751	API_SIGNON
gnuspool-1.5	Spooling system	477	10,391	94,446	load_maind, pendulist
gprolog-1.3.1	Prolog interpreter	170	15,701	49,246	pl_stm_tbl
gs5.50	Postscript interpreter	792	51,425	169,099	pmat, PNG_INFO_PLTE
guile-1.8.7	Scheme interpreter/compiler	265	14,610	73,964	scm_set_smob_print
hurd-0.2	OS kernel	869	33,577	97,672	ipfrag
icecat-3.0.2-g1	Web browser	5,416	511,274	1,227,838	CKA_KEY_TYPE, CKR_SESSION_READ_ONLY, pBt, PRBool, SECOID_SetAlgorithmID
libextractor-0.5.22	Metadata extraction lib	4,516	18,319	2,455	AV_NOPTS_VALUE, esi
libjit-0.1.2	Just in time compilation	126	17,598	70,241	dpas_sem_is_rvalue, fpcw_new_offset
libunistring-0.9.1.1	Unicode manipulation	1,183	20,787	203,195	ENOTCONN
linux-2.6.31.6	OS kernel	1,801	194,437	753,366	ac_comm, blk_queue_io_stat
lynx-2.7.1	Web browser	196	21,727	70241	dumbterm
mifluz-0.24.0	Library for full-text inverted index	243	31,159	53,246	LF_ISSET
mtools-4.0.12	MS-DOS utilities for Unix	90	3,213	17,661	EXTCASE
nethack-3.2.2	Dungeon exploration game	313	21,363	154,412	NUMMONS
pnet-0.8.0	.NET porting	680	77,546	330,447	FFI_Ok
pspp-0.6.2	Statistical data analysis	727	29,977	134,958	case_data_rw_idx
radius-1.6	Remote user authentication	233	12,877	73,930	dict_value_iter_helper, grad_avp_t, mempcpy
sed-4.2.1	Regular expression interpreter	116	6,355	25,601	rm_so
sendmail-8.8.5	Mail server	52	10,594	34,444	denlstring
xaos-3.0	Fractal zoomer	109	1,978	26,894	cimage

Table B.2 reports the expansions of all the identifiers used in the experiments. The column *Separator* indicates whether underscore or CamelCase separators are used. The columns *Abbr.*, *Acro.* and *Plain* report the number of abbreviations, acronyms and plain English words composing each identifier.

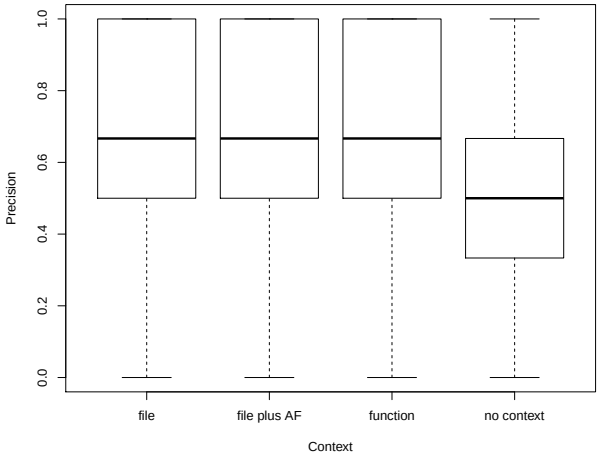
Table B.2 Splitting/expansion oracle and kinds of terms composing identifiers.

Identifiers	Separators	Oracle	Abbr.	Acro.	Plain
ac_comm	√	accounting communication	2	0	0
acct_file	√	accounting file	1	0	1
API_SIGNON	√	application programming interface sign on	0	1	2
argv_to_scm	√	argument vector to source code module	2	1	1
arm_reg_parse	√	arm register parse	1	1	1
assoc		association	1	0	0
AV_NOPTS_VALUE	√	average number options value	3	0	1
bfd_abs_section_ptr	√	binary file descriptor absolute section pointer	2	1	1
blk_queue_io_stat	√	block queue input output statistic	2	1	1
case_data_rw_idx	√	case data row index	2	0	2
cimage		current image	1	0	1
CKA_KEY_TYPE	√	check attribute key type	2	0	2
CKR_SESSION_READ_ONLY	√	check return session read only	2	0	3
dbsm_start	√	decibel per square meter start	0	1	1
denlstring		delete new line string	3	0	1
dfp		default face pointer	0	1	0
dict_value_iter_helper	√	dictionary value iterator helper	2	0	2
dpas_sem_is_rvalue	√	dynamic pascal semantic is right value	4	0	2
dumbterm		dumb terminal	1	0	1
dupok		duplicate ok	1	0	1
ENOTCONN		endpoint not connected	2	0	1
esi		extended source index	0	1	0
EXTCASE		extended case	1	0	1
f_getlk	√	file get lock	2	0	1
FFEBAD_severityFATAL	√	fortran front end bad severity fatal	0	1	3
FFI_Ok	√	foreign function interface ok	0	1	1
fpcw_new_offset	√	floating point control unit word new offset	0	1	2
gmon_io_write	√	graphic monitor input output write	2	1	1
GMP_NUMB_MAX	√	gnu multi precision number maximum	2	1	0
grad_avp_t	√	gnu radius attribute value pointer type	6	0	0
HD_SYLVESTER	√	holiday sylvester	1	0	1
hol_cluster_base	√	help option list cluster base	0	1	2
internal_auxent	√	internal auxiliary entities	2	0	1
ipfrag		internet protocol fragment	1	1	0
LF_ISSET	√	line feed is set	2	0	2
load_maind	√	load main directory	1	0	2
load_scnptr	√	load scan pointer	2	0	1
HD_SYLVESTER	√	holiday sylvester	1	0	1
mempcpy		memory pointer copy	3	0	0
NUMMONS		number monsters	2	0	0
nvtbl		non virtual table	3	0	0
pBt		pointer binary tree	0	1	9
pendulist		pending user list	2	0	1
pl_stm_tbl	√	prolog stream table	3	0	0
pmat		partitioned matrix	2	0	0
PNG_INFO_PLTE	√	portable network graphics information palette	2	1	0
PRBool		portable runtime boolean	3	0	0
rm_so	√	remove shared object	3	0	0
rrt_prev	√	rip routing table	1	1	0
scm_set_smob_print	√	scm set small object print	2	1	2
SECOID_SetAlgorithmID	√	security object identifier set algorithm identifier	4	0	2

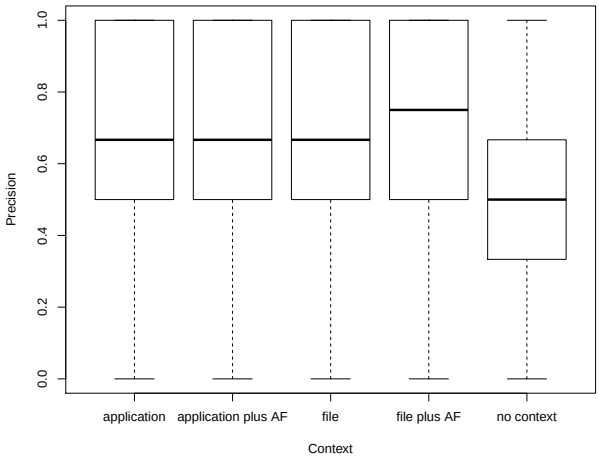
Detailed Results

This appendix reports figures detailing results presented and discussed in Section 4.4 of Chapter 4. Specifically, Fig. B.1 and Fig. B.2 show boxplots of Precision and Recall for the

different levels of context, respectively.

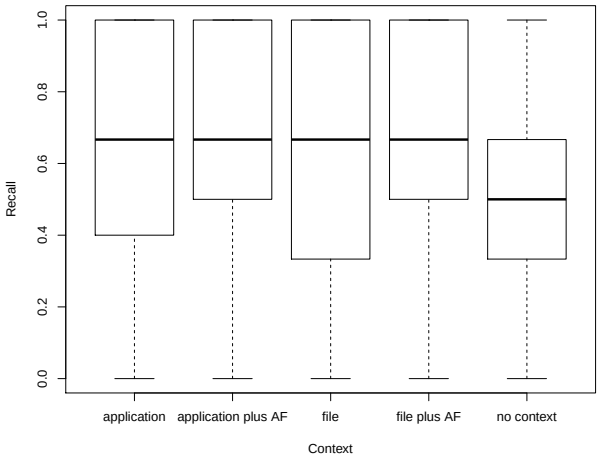


(a) Exp I

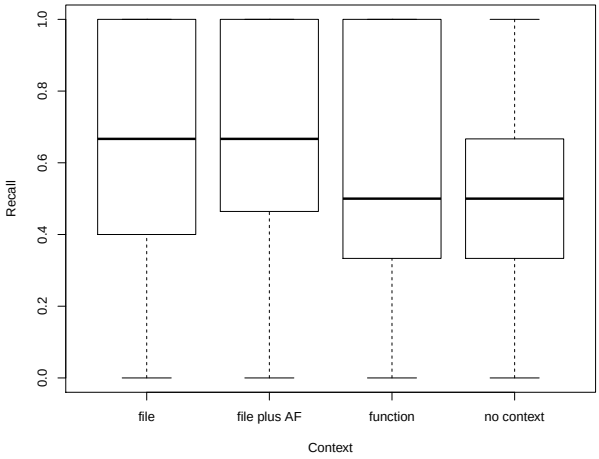


(b) Exp II

Figure B.1 Boxplots of precision for the different context levels (AF= Acronym Finder).



(a) Exp I



(b) Exp II

Figure B.2 Boxplots of recall for the different context levels (AF= Acronym Finder).