EPM–RT–2010-09

AUTOMATIC DERIVATION OF CONCEPTS BASED ON THE ANALYSIS OF IDENFIERS

Latifa Guerrouj
Département de Génie informatique et génie logiciel
École Polytechnique de Montréal

Septembre 2010

EPM-RT-2010-09

Automatic Derivation of Concepts Based on the Analysis of Idenfiers

Latifa Guerrouj
Département de génie informatique et génie logiciel
École Polytechnique de Montréal

Septembre 2010

Dépôt légal :
Bibliothèque nationale du Québec, 2010
Bibliothèque nationale du Canada, 2010

EPM-RT-2010-09
Automatic Derivation of Concepts Based on the Analysis of Identifiers
par : Latifa Guerrouj
Département de génie informatique et génie logiciel
École Polytechnique de Montréal

Université de Montréal

Automatic Derivation of Concepts Based on the Analysis of Identifiers

by
Latifa Guerrouj

Department of Computer Science and Software Engineering
Ecole Polytéchnique de Montréal

Research Proposal Submitted
in partial fulfilment of the requirements
for the degree of Doctor of Philosophy (Ph.D.) in software engineering

Août, 2010

Université de Montréal
Ecole Polytéchnique de Montréal

This research proposal :

Automatic Derivation of Concepts Based on the Analysis of Identifiers

presented by:

Latifa Guerrouj

has been evaluated by the jury made up of the following people:

Gabriela Nicolescu,	président-rapporteur
Guiliano Antoniol,	directeur de recherche
Yann-Gaël Guéhéneuc,	codirecteur
Ivan Labish,	membre du jury

ABSTRACT

The existing software engineering literature has empirically shown that a proper choice of identifiers influences software understandability and maintainability. Indeed, identifiers are developers' main up-to-date source of information and guide their cognitive processes during program understanding when the high-level documentation is scarce or outdated and when the source code is not sufficiently commented.

Deriving domain terms from identifiers using high-level and domain concepts is not an easy task when naming conventions (*e.g.*, Camel Case) are not used or strictly followed and–or when these words have been abbreviated or otherwise transformed. Our thesis aims at developing a contextual approach that overcomes the shortcomings of the existing approaches and maps identifiers to domain concepts even in the absence of naming conventions and–or the presence of abbreviations. We also aim to take advantage of our approach to enhance the predictability of the overall system quality by using identifiers when assessing software quality.

The key components of our approach are: dynamic time warping algorithm (DTW) used to recognize words in continuous speech, string-edit distance between terms and words as a proxy for the distance between the terms and the concepts they represent, plus words transformations rules attempting to mimic the cognitive processes of developers when composing identifiers with abbreviated forms.

To validate our approach, we apply it to identifiers extracted from different open source applications to show that our method is able to provide a mapping of identifiers to domain terms, compare it with the two families of approaches that to the best of our knowledge, exist in the literature with respect to an oracle that we have manually built. We also enrich our technique by using domain knowledge and context-aware dictionaries to analyze how sensitive are the performances of our approach to the use of contextual information and specialized knowledge.

Keywords: Identifier Splitting, Program Comprehension, Linguistic Analysis, Software Quality.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Research Context: Program Comprehension and Software Quality

There is a general consensus among researchers [CT99, CT00, DP05, LFB06, EH-PVS09] on the usefulness of identifier to improve software quality, program comprehension, and program understandability. Indeed, researchers have studied the usefulness of identifiers to recover traceability links, measure conceptual cohesion and coupling [MPF08, PM06], and, in general, as an asset that can highly affect source code understandability and maintainability [TGM96, LMFB07, LMFB06]. Researchers have also studied the quality of source code comments and the use of comments and identifiers by developers during their understanding and maintenance activities [LMFB06, FWG07, JH06]. They all concluded that identifiers can be useful if carefully chosen to reflect the semantics and the role of the named entities that they are intended to make up and that "it is the semantics inherent to words that determine the comprehension process". Stemming from Deißenböck and Pizka observation on the relevance of terms in identifiers to drive program comprehension, almost all previous works attempted to segment identifiers by splitting them into component terms and words to guide the cognitive process using these identifier fragments. Indeed, identifiers are often composed of terms reflecting domain concepts [LMFB06], referred to as *"hard words"*. Hard words are usually concatenated to form compound identifiers, using the Camel Case naming convention, *e.g.*, *drawImage*, or underscore, *e.g.*, *draw_image*. Sometimes, no Camel Case convention or other separator (*e.g.*, underscore) is used. Also, acronyms and abbreviations may be part of any identifier, *e.g.*, *drawimg* or *cntrapplicationgid*. The component words *draw, application*, the abbreviations *img, cntr*, and the acronym *gid* (*i.e.*, group identifier) are referred to as *"soft-words"* [LFB06]. Unlike Java developers who often relies on the use of English and the Camel Case convention,

which is a practice of creating identifiers by concatenating terms with capitalized first letter *e.g.*, *drawImage* to create identifiers, C programs usually concatenate terms into identifiers using an underscore as separator, *e.g.*, *draw_image*. In addition, C coding standards such as the Indian Hill coding standards or the GNU coding standards do not enforce Camel casing.

These practices, Camel Case and underscore concatenations, lead to the development of a family of algorithms to segment identifiers into component substrings. These algorithms have in common the assumption that the Camel Case convention and–or an explicit separator are used systematically to create identifiers. Recently, a more complex strategy is implemented by the Samurai tool [EHPVS09], it relies on a lexicon and uses greedy algorithms plus strings frequency tables to identify terms composing an identifier. However, the above mentioned approaches have some limitations. First, they are not always able to associate identifier substrings to words or terms, *e.g.*, domain-specific terms or English words, which could be useful to understand the extent to which the source code terms reflect terms in high-level artifacts [LDOZ06]. Second, they do not deal with word transformations, *e.g.*, abbreviation of *pointer* into *pntr*. It is the reason that motivated us to develop a new approach aiming at improving over the existing techniques in terms of correctly mapping source code identifiers to concepts even in the absence of naming conventions or the presence of abbreviations. A second purpose of this project is to take advantage of the results of this approach to understand how could identifiers be used as a metric to enhance the prediction of the overall system quality. The idea is to combine structural measures involving coupling and cohesion with informal information that is, in our case, identifiers to predict the quality of a software system.

In the next section, we will enumerate the main problems that we wish to address during the accomplishment of this research project.

1.2 Problem Statement: Identification of Concepts based on Identifiers

In the following, we will interchangeably use the terms concept and domain term due to the fact that our approach deals with concepts that could be design or domain level concepts.

We formulate our problem statement as follows:

How to build a contextual approach able to automatically derive domain terms and, thus, concepts based on the analysis of source code identifiers even in the absence of naming conventions and–or the presence of truncated/abbreviated words?

A study of related work highlighted the following problems, among which, some have already been addressed and others are stated as research directions.

Problem 1. There is a little research work that treats the problem of mapping source code identifiers to domain terms. To the best of our knowledge, only two families of approaches are available. The first is essentially based on the presence of naming conventions (*e.g.*, Camel Case) and explicit separators such as the underscore. The second approach uses greedy algorithms and a scoring function that relies on word frequencies, mined from the source code to determine likely splittings of identifiers.

Problem 2. The only existing approach that segments same case identifiers belongs to the latter family [EHPVS09]. It is called "Samurai" and assumes that an identifier is composed of words used (alone) in some other parts of the system. The main weakness of this technique is its system-dependent frequency tables that could lead to different splittings for the same identifier depending on the tables. Tables built from different programs may lead to different results. Also, 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.

Problem 3. Existing techniques that map source code identifiers to domain terms do not deal with abbreviations and acronyms that constitute additional forms of identifiers used by some developers when writing code.

Problem 4. The two families of approaches stated above do not show how close is a match (a split) to the unknown string (identifier to split). This estimation of similarity between the identifier to split and the resulting split could be very useful to understand the extent to which source code identifiers reflect terms in high-level artifacts.

Problem 5. The use of upper ontologies like WordNet by our first proposed approach called DECOS [MGD+10] leads, in some cases, to multiple candidate splits for the same identifier. As a term carries a single meaning in the context where it is used, it would be possible to incorporate context and specialized knowledge in our technique to see how sensitive are its performances to the use of such contextual information.

Problem 6. One of the main limitations of previous approaches is their inability to deal with abbreviations and word transformations when mapping identifiers to concepts. DECOS addressed this shortcoming by automatically generating a thesaurus of abbreviations via transformations applied in the context of a hill climbing search. These word transformations attempt to mimic the cognitive processes of developers when composing identifiers with abbreviated forms. However, our approach has a non-deterministic component in the way in which word transformation rules are applied and in the way in which the candidate words to be transformed are selected. Thus, the risk of choosing an inappropriate term among two having the same distance is not negligible.

Problem 7. DECOS is almost always able to find a splitting in a reasonable time, *i.e.*, within 2 minutes in case of a dictionary composed of 3,000 words. Yet, we are aware that the computation speed could be optimized. As we apply a hill climbing algorithm, in which a transformed word is added to the dictionary if and only if it reduces the global distance, the invocation of the procedure dedicated for the optimal matching computation will be done twice when the best candidate (word having the min distance) is added to the considered dictionary.

Problem 8. To evaluate the performances of our approach in term of the percentage of correctly segmented identifiers and to compare it with alternative

ones, it is necessary to have an oracle. Our experience in attempting to build and use an oracle has shown that this task is non trivial especially for a large set of projects (hundreds of GNU projects from which we sampled some identifiers).

Problem 9. The existing software engineering literature reports empirical evidence on the relation between various characteristics of a software system and software quality. A recent study [MPF08] has shown that informal information such us comments and identifiers, if combined with existing structural cohesion metrics, proves to be a better predictor of faulty classes when compared to different combinations of structural cohesion metrics. Since our method of deriving domain terms based on identifiers reveals the degree to which an identifier reflects the real context in which it is used, it would be interesting to expose the problem of integrating identifiers as a metric when predicting the overall software system quality.

1.3 Proposed Solutions: DECOS and TIDIER

Our global contribution consists in providing a new technique for detecting concepts based on the analysis of source code identifiers. The first approach that we have proposed is called DECOS (Detection of Concepts based on Source Code Identifiers) and relies on the use of dictionaries. DECOS associates identifiers to terms and words; words belonging to either a full English dictionary or to a domain-specific or an application-specific dictionary. Our second proposed solution is called TIDIER (Term IDentifier RecognIzER). TIDIER is a contextual approach for mappings identifiers to concepts, it is an extension of DECOS that incorporates specialized knowledge and contextual information. Both DECOS and TIDIER overcome the shortcomings of the existing tools and provide results that could be exploited further to improve the prediction of the quality of a given software system. This claim is justified by the percentages of precision and recall obtained by the two proposed approaches (DECOS and TIDIER) in comparison with those attained by previous approaches (Camel Case splitter and Samurai). Details about

correctness of mappings, precision and recall achieved by our techniques will be discussed in Chapter 4 and Chapter 6 that respectively show the results obtained by DECOS and TIDIER.

For each problem stated in the previous section, we propose the following solutions:

Solution 1. To overcome the limitations of previous approaches, we propose a novel approach called DECOS that can automatically map identifiers composed of transformed words such as abbreviations and acronyms to domain terms regardless of the kind of separators.

Solution 2. Unlike Samurai which is a contextual approach that maps identifers to concepts by mining the frequency of potential substrings from source code, DECOS relies on the use of dictionaries. It uses a dictionary containing words and terms belonging to an upper ontology, to the application domain, or both. In addition, our approach also uses context-aware dictionaries and dictionaries enriched with specialized knowledge. Examples of dictionaries that can be used for our purpose are, for example, WordNet (which contains around 90,000 entries) or dictionaries used by spell checkers, such as a-spell (which contains around 35,000 English words in a typical configuration). Each dictionary word may be associated to a set of known abbreviations in a way similar to a thesaurus. For example, the pointer entry in the dictionary can be associated to abbreviations *pntr*, *ptr* found as terms composing identifiers. Thus, if *pntr* is matched, the algorithm can expand it into the dictionary term *pointer*.

Solution 3. To deal with word abbreviation and transformation, the proposed approach assumes that there is a limited set of (implicit and–or explicit) rules applied by developers to create identifiers. It, therefore, implements a number of words transformation rules that we discuss in detail in the Chapter 3. The transformations are applied to dictionary words in the context of a hill climbing search [MF04].

Solution 4. The proposed approach shows how close is the match to the unknown string (identifier to split). In fact, it is based on the dynamic time warping

(DTW) algorithm that is able to provide a distance between an identifier and a set of words in a dictionary even if there is no perfect match between substrings in the identifier and dictionary words; for example, when identifiers are composed of abbreviations, *e.g.*, *getPntr*, *filelen*, or *DrawRect*. The DTW algorithm accepts the match as it identifies those dictionary words closest to identifier substrings. This estimation of similarity between the identifier and the resulting split could be very useful to understand the extent to which identifiers embedded in a given program reflect terms in high level artifacts.

Solution 5. To see how sensitive are the performances of our technique to the use contextual information and specialized knowledge in different dictionaries, we have built context-aware dictionaries. Similarly to Enslen approach, these contextual dictionaries contain function level, source code file level, and application level identifiers. We have used also an application dictionary, plus specialized knowledge: a dictionary based on the application dictionary augmented with domain knowledge (abbreviations, acronyms, and C library functions) in addition to the small and complete English dictionaries. The use of C library functions is justified by the fact that our contextual approach TIDIER deals with identifiers belonging to C applications. The dictionaries used will be detailed in Chapter 5.

Solution 6. To increase the accuracy of our results and to obtain the maximum number of appropriate candidates (terms composing an identifier), we will improve the heuristics used for choosing dictionary words to be transformed and word transformations, possibly by coupling our approach with the strategy derived from [EHPVS09], *i.e.*, favoring words already used in the same context or by enhancing our fitness function based mainly on the string-edit-distance.

Solution 7. The string-edit distance used by our technique 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 the distance matrix, with a complexity of $\mathcal{O}(M \times N)$. Since there are T dictionary words, the overall complexity is $\mathcal{O}(T \times M \times N)$. An increase of perfor-

mance could be achieved by saving the first edit-distance computation and, in the context of the hill climbing, recomputing only cells where the distance improves.

Solution 8. To evaluate the performances of the proposed approach and to compare it with alternative ones, it is necessary to have an oracle, *i.e.*, for each identifier, we will have a list of terms obtained after splitting it and, wherever needed, expanding contracted words. For example, a possible oracle for *counterPntr* would be *counter pointer*, obtained by splitting the identifier after the seventh character and after mapping the abbreviation *Pntr* to *pointer*. Ideally, a perfect oracle could exist; however, because its creation was infeasible to achieve for the hundreds of GNU projects (*e.g.*, wordnet, ispell, Internet) from which we sampled the identifiers, the oracle has been produced by two of other researchers and validated by another researcher to avoid bias and subjectivity. Thus, DECOS and TIDIER are compared against previous approaches with respect to a manually built oracle that helps evaluating the performance of the proposed solutions in terms of correctly mappings identifiers to concepts.

Solution 9. To take advantage of this research work in the enhancement of the prediction of the quality of a given software system, we propose to include a node called "identifiers" as a metric in the Software Quality Understanding through the Analysis of Design (SQUAD) quality model [Kho09]. We believe that combining such an informal information with the structural cohesion metrics already integrated by the developers of this quality model would help in improving the prediction of the quality of the analyzed software systems as it has been proven by Andrian Marcus, Denys Poshyvanyk and Rudolf Ferenc in their seminal work [MPF08]. Once the metric "identifiers" is taken into account by SQUAD, we perform experimental studies to analyze the effect of such informal information on the prediction of the quality of software systems.

1.4 Methodology

To answer our global research question, we propose a solution based on the string-edit distance between terms and words to quantify how close are words, representing concepts, to such terms and, thus, provide a measure of the likelihood that the terms refer to some words. To deal with abbreviations, our proposed techniques use a thesaurus of words and abbreviations and apply word transformation rules in the context of a hill-climbing search.

Some abbreviations are well-known and can thus be part of the thesaurus. In such case, each row of the thesaurus contains a word and its possible synonyms, *e.g.*, *directory* and *dir*. Some other abbreviations may not appear in the thesaurus because they are too domain and–or developer specific. To cope with such abbreviations, our approach find the best segmentation using a string-edit distance and a greedy search. The transformation rules are applied in the context of a hill-climbing algorithm that 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.

In a nutshell, our approach relies on input dictionaries and a distance function to segment (if necessary) simple and composed identifiers and associate the resulting terms with words in the dictionaries, even if the terms are truncated/abbreviated. 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).

we propose and follow a methodology where the tree main phases are as follows:

1. **Building a thesaurus:** To map terms or transformed words composing identifiers to dictionary words, we build a thesaurus of words and abbreviations. 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 which contains about 35,000 words, or of upper ontologies, *e.g.*, WordNet, which contains about 90,000 entries. Yet, to reduce the computation time, we build smaller dictionaries, *e.g.*, dictionaries containing the most frequently-used

English words only as well as specialized dictionaries containing acronyms and known abbreviations.

2. **Building an oracle:** To validate our approach, we need an oracle. This means that for each identifier, we will have a list of terms obtained after segmenting it and, wherever needed, adding contracted words. The oracle is produced as follows: (i) a splitting of each sampled identifier, and expanded abbreviations is produced independently by two researchers (ii) for all cases where there is a different splitting/expansion, we hold a discussion meeting and a consensus is reached. We perform the manual analysis of our approach relying on various sources of information of the projects, ranging from source code comments to user manuals.

3. **Validation:** To evaluate our approach, we apply it to derive concepts from identifiers of different systems and open source projects. In fact, we first apply our technique to JHotDraw and Lynx using an English dictionary composed of 3,000 words. Then, we apply it to a set of 1,026 C identifiers randomly extracted from a corpus of 340 open source programs using a single English dictionary of 2,600 words and also various dictionaries: a dictionary of about 2,800 words, the previous dictionary augmented with domain knowledge, *i.e.*, about 700 domain terms, acronyms, and well-known abbreviations, a full English dictionary of 175,000 words, contextual dictionaries, *i.e.*, dictionaries built using terms from the same function, file, or program, and contextual dictionaries augmented with domain knowledge. Our empirical study compares the three approaches (ours, Camel Case splitter and Samurai) on their splitting correctness (with respect to the oracle), and on their precision, recall, and F-Measure.

3. **Performing empirical studies:** To analyze the effect of source code identifiers on the prediction of the overall system quality, we propose to take into account identifiers when assessing software quality. The idea is to add a new metric that represents identifiers to Software Quality Understanding

through the Analysis of Design (SQUAD) model considered as a Bayesian quality model that does not combine informal information such as identifiers and comments with structural metrics when evaluating software quality. The Bayesian network used by SQUAD includes information on classes participating in design patterns, antipatterns, and code smell to assess the quality of systems. Bayesian networks are probability graphs used for the evaluation and the prediction of software quality. Each node of a bayesian network is a metric taken into account when predicting the overall system quality. These metrics (nodes) involve cohesion, coupling, and inheritance, etc. Once identifiers are added as a metric to the Bayesian network of SQUAD, we could design experiments to study the impact of using identifiers when evaluating quality. In fact, one research direction could be to analyze the effect of introducing non well-formed identifiers on the quality of a given system. A non-well formed identifier could be an identifier for which TIDIER is not able to find a split especially that our analysis method showed that TIDIER outperforms previous approaches of mapping identifiers to concepts. Thus, SQUAD model would be run taking into consideration not only structural metrics but also identifiers. By doing so, we could verify the possibility of enhancing the prediction of software quality based on identifiers and deduce how important is the use of meaningful identifiers for the improvement of quality of systems.

1.5 Proposal Outline

This proposal will begin with a background literature review in Chapter 2. Three areas related to our research are surveyed, including program comprehension, software quality, and traceability recovery. However, the focus is mostly on the investigation of identifiers in these three areas. Chapter 2 will also describe two state-of-the-art and state-of-the-practice approaches to map identifiers to domain terms.

In Chapter 3, we will present DECOS, our first proposed approach for detecting concepts represented by source code identifiers, explain its main components, and discuss the challenges with dealing with abbreviations.

In Chapter 4, we will report the results obtained by DECOS, and describe the case study that we have performed for validation and performance analysis of our technique.

In Chapter 5, we will introduce TIDIER, our tool designed for the automatic derivation of domain terms, and focus mainly on the contextual aspect that we have tried to introduce in our approach by the use of contextual information and specialized knowledge.

In Chapter 6, we will detail the empirical validation of our technique and compare it against previous approaches, with respect to the oracle that we have manually built.

In Chapter 7, we will conclude our proposal and recall our main contributions.

In Chapter 8, we will enumerate our research directions, and detail our research plan in terms of research schedule and publication plan.

CHAPTER 2

RELATED WORK

The important role of identifiers in program understanding, traceability recovery, feature and concept location motivates the large body of relevant work. In this chapter, we state the most relevant contributions to identifier splitting and present a description of the previous techniques that map source code identifiers to domain concepts.

2.1 Program Comprehension

In [BMW93], Biggerstaff *et al.* define comprehension of the code, they wrote : "A person understands a program when he or she is able to explain the program, its structure, its behavior, its effects on its operation context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program". Therefore, mapping of source code identifiers to domain terms is important when comprehending programs.

In [SIS02], Sulaiman reported that developers and maintainers face the challenging task of understanding of the system when the code is not sufficiently documented. Understanding of a legacy system is a time consuming activity especially when documentation is out-dated or does not exist. Hence, software maintainers must study the source code of the software systems.

Guidelines for the production of high-quality identifiers have been provided by Deißenböck *et al.* [DP05]. The authors highlighted that proper identifiers improve software quality. They believe that it is essential that identifiers and comments contain the concept that they represent. They introduced two rules for creating well-formed identifiers: conciseness and consistency. To verify the conciseness and consistency of identifiers, they provided a mapping from identifiers to the domain of concepts. The results of their study showed that inconcise or inconsistent identifiers

cause a complexity in comprehension of source code.

Takang *et al.* [TGM96] empirically analyzed the role played by identifiers and comments on source code comprehensibility. They conducted experiments to compare abbreviated identifiers to full-word identifiers and uncommented code to commented code. They had as subjects 89 undergraduates in computer science who studied a program for 50 minutes and used both an objective and subjective means of assessing comprehensibility. They tested different hypotheses:

1. Commented programs were more understandable than non-commented ones.

2. Programs that contain full identifiers are more understandable than those with abbreviations.

3. The combined effect of comments and full identifiers was more understandable than either independently.

The results of this study showed that commented programs are more understandable than non-commented programs and programs containing full-word identifiers are more understandable than those with abbreviated identifiers.

Lawrie *et al.* [LMFB07] studied the effect of identifier structure on developers' ability to manipulate code. They studied two hypotheses:

1. Well-constructed abbreviations and full natural-language identifiers help source code comprehension when compared to less informative identifiers.

2. Well-constructed abbreviations and full natural-language identifiers lead to better programmer recall than less informative identifiers.

The results of their empirical study showed that full-word identifiers lead to the best program comprehension.

Lawrie *et al.* [LMFB06] studied the effect of identifiers (three levels of identifier quality that are full-words, abbreviations, and single letters) in source code comprehension. They investigated two hypotheses: first, schooling and people with more work experience comprehend the source code better. Second, gender plays a

great role in confidence but not comprehension of the source code. They considered that if using full words identifier helps program comprehension over the abbreviated identifiers, then it is recommended to build tools that extract information from identifiers; for example, applying a standard dictionary. Their study showed that better comprehension is achieved when full word identifiers are used rather than single letter identifiers as measured by description rating and confidence in understanding.

Lawrie *et al.* also presented an approach and a tool named QALP (Quality Assessment using Language Processing) relying on the textual similarity between related software artifacts to assess software quality [LMFB07, LMFB06]. The QALP tool leverages identifiers and related comments to characterize the quality of a program. Their empirical study conducted with 100 programmers showed that full-words as well as recognizable abbreviations led to better comprehension. These results suggest that the identification of words composing identifiers, and, thus, of the domain concepts associated with them could contribute to a better comprehension.

Binkley *et al.* [BDLM09] investigated the use of different identifier separators in program understanding. They found that the Camel Case conventions led to better understanding than underscores and, when subjects are properly trained, that they performed faster with identifiers built using the Camel Case convention rather than with underscores.

Other works [CT00, MMM03] investigated the information carried by the terms composing identifiers, their syntactic structure and quality. The existence of *"hard terms"* that encode core concepts into identifiers was the main outcome of the study by Anquetil *et al.* [AL98].

An in-depth analysis of the internal identifier structure was conducted by Caprile *et al.* [CT99]. They reported that identifiers are chosen to convey relevant information about the role and properties of the program entities that they label. They also observed that identifiers are often the starting point for program comprehension, especially when high-level views, such as call graph, are available.

Caprile *et al.* [CT00] proposed a semiautomatic technique for the restructuring of identifiers with the goal of improving their meaningfulness and making identifiers self descriptive.

Methods related to identifier refactoring were suggested by Demeyer *et al.* [DDN00]. The authors proposed heuristics for detecting refactorings by calculating metrics over successive versions of a system. To validate their approach, they performed three case studies for which multiple versions are available with the aim of investigating how information of identifying refactoring helps in program comprehension.

De Lucia *et al.* [LDOZ06, DDO10] 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.

Researchers [JH06, LMFB06, FWG07] have also studied the quality of source code comments and the use of comments and identifiers by developers during understanding and maintenance activities. They all concluded that identifiers can be useful if carefully chosen to reflect the semantics and role of the named entities. Structure of the source code and comments help program comprehension and therefore reduce maintenance costs.

Fluri *et al.* [FWG07] examined the question whether source code and associated comments are really changed together along the evolutionary history of a software system. They developed an approach to map code and comments to observe their co-evolution over multiple versions and investigated three open source systems: ArgoUML, Azureus, and Eclipse JDT Core. Their study focused on the ratio between the source code and comments over the history of projects and the entities that are most likely to be commented, *e.g.*, classes, methods, and control statements. They noticed that comment density, the percentage of comment lines in a given source code base, is a good predictor of maintainability and hence survival of a software project. Specifically, they observed whether the comment density remains stable over time and whether developers maintain a strong commenting discipline over

a project's lifetime. Regarding the comment ratio over a project's lifetime, they found that it does not stay at a stable value.

Jiang and Hassan [JH06] studied source code comments in the PostgreSQL project over time. They measure how many header comments and non-header comments were added or removed to PostgreSQL over time. Header comments are comments before the declaration of a function; whereas non-header comments are all other comments residing in the body of a function or trailing the function. They discovered that apart from the initial fluctuation due to the introduction of a new commenting style; the percentage of functions with header and non-header comments remains consistent throughout the development history. They reported that the percentage of commented functions remains constant except for early fluctuation due to the commenting style of a particular active developer. A crucial role is recognized to the program lexicon and the coding standards in the so-called naturalization process of software immigrants [SH98].

2.2 Software Quality

In this section, we focus not only on research works on software quality but also those accomplished in traceability recovery due to their relation with the area of software quality and source code identifiers.

Some researchers [ACC+02, MACHH05, MM03] reported the use of identifiers to recover traceability links. They believe that analysis of the identifiers and comments can help to associate high-level concepts with program concepts and vice-versa because they capture information and developers' knowledge while writing the code. Thus, how meaningful identifiers are could be a quality program indicator.

Antoniol *et al.* [ACC+02] used an Information Retrieval (IR) method to recover traceability links between source code and free text documents. They applied both a probabilistic and a vector space information retrieval model in two case studies to trace C++ and Java source code units, manual pages, and functional require-

ments. A premise of their work is that programmers use meaningful names for program items, such as functions, variables, types, classes, and methods. The authors believe that the application-domain knowledge that programmers process when writing the code is often captured by the mnemonics of identifiers; therefore, the analysis of these mnemonics can help to associate high-level concepts with program concepts and vice-versa. They proposed a two-phase approach: first they prepared the document for retrieval by indexing its vocabulary extracted from the document; second they extracted and indexed a query for each source code component by parsing the source code component and splitting the identifiers to the composed words. With this method, Antoniol *et al.* computed the similarity between queries and documents and returned a ranked list of documents for each source code component. Recovering traceability links between source and documentation reveals the extent to which source code reflects design requirements and thus help developers improve the quality of their programs.

De Lucia *et al.* [DFOT07] used LSI to identify cases of low similarity between artifacts previously traced by software engineers. Their technique relied 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 of meaningless identifiers or poor comments in source code, may point to a poor development process and unreliable traceability links.

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.* [AD05] and the Conceptual Coupling of Classes proposed by Poshyvanyk *et al.* [PM06], which bring complementary information with respect to structural cohesion and coupling measures.

In [AD05], Marcus *et al.*proposed a new measure, named the Conceptual Cohesion of Classes (C3), for the cohesion of classes in Object Oriented software

systems. C3 is based on the analysis of unstructured information embedded in the source code involving comments and identifiers and is inspired by the mechanisms used to measure textual coherence in cognitive psychology and computational linguistics. The reported results show that combining C3 with existing structural cohesion metrics proves to be a better predictor of faulty classes when compared to different combinations of structural cohesion metrics such as cohesion, coupling, etc. Therefore, identifiers could be helpful when predicting the quality of a given system.

Similar results were obtained by Poshyvanyk *et al.* [PM06] who suggested the CoCC metric (COnceptual Coupling of Classes) to capture the coupling among classes based on semantic information obtained from source code identifiers and comments. Their case study showed that the conceptual measure captures new dimensions of coupling, which are not captured by existing coupling measures.

Antoniol *et al.* [ACC+02] observed that most of the application-domain knowledge that developers possess when writing code is captured by identifier mnemonics. They wrote that "[p]rogrammers tend to process application-domain knowledge in a consistent way when writing code: program item names of different code regions related to a given text document are likely to be, if not the same, at least very similar". Thus, how readily the semantics inherent to identifiers can be extracted is of key importance.

Abebe *et al.* [AHM+08] analyzed how the source code vocabulary changes during evolution. They performed an exploratory study of the evolution of two large open source software systems. The authors observed that the vocabulary and the size of a software system 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. Hence, meaningfulness of identifers reflects quality of software programs.

2.3 Derivation of Concepts

We present a description of two state-of-the-art and state-of-the-practice approaches to split identifiers into terms. These approaches are the Camel Case splitter, and Samurai by Enslen *et al.* [EHPVS09].

2.3.1 Camel Case Splitter

Camel Case Splitter is an approach that assumes the use of the Camel Case naming convention or the presence of an explicit separator. Camel Case is a naming convention in which a name is formed of multiple words that are joined together as a single word with the first letter of each of the multiple words capitalized so that each word that makes up the name can easily be read. The name derives from the hump or humps that seem to appear in any Camel Case name. For example, the words *FirstYearTeaching* or *numberOfBugs* use camel case rules. The Camel Case splitter splits 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 Camel Case 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. This approach cannot split *FFEINFO* or *kindtypereal* into terms, *i.e.*, the acronym *FFE* followed by *INFO* and the terms *kind*, *type*, and *real*.

We have implemented this technique as a baseline for comparison. In our implementation, we do not model cases in which developers assigned a specific meaning to some characters, *e.g.*, the digits 2 and 4 could stand for the terms *to* and *for*, respectively as in the identifiers *peer2peer* and *buffer4heap*. Furthermore, digits sequences and single character are pruned out.

2.3.2 Samurai Splitter

Samurai approach [EHPVS09] is an automatic approach and tool 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 part of the program or in other programs alone or as part of other identifiers.

2. Given two possible splittings of a given identifier, the split that more 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 the context of identifier. 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 systems.

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 Camel Case splitter but also uses the frequency tables. Given an identifier, first, Samurai applies **RuleA** and **RuleB** from the Camel Case 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 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 right 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 character, or a single upper-case character followed by same-case characters.

The starting point for 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[1] 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.)

Now, Let us consider, for example, the identifier *kindtypereal*, and assum that the first split is *kind* and *typereal*. Because neither *kind* nor *typereal* are common prefix/suffix, this split is kept. Now, if we 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 it 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 limitation of Samurai is its frequency tables. These tables could lead to different splittings for the same identifier depending on the tables from different systems. Tables built from different programs may lead to different results. Also, 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.

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

CHAPTER 3

DECOS

In this chapter, we introduce DECOS, our novel approach for mapping source code identifiers to concepts. First, we describe our technique and state its main components: DTW, string edit distance, and word transformation rules. Then, we detail each of these components in isolation.

3.1 Approach Description

Our global contribution consists in providing a new approach for detecting concepts represented by source code identifiers. The approach is supposed to overcome the shortcomings of the existing tools and to provide accurate results that could be exploited further to improve the prediction of the quality of a given software system.

Our first contribution tried to address some of the research questions stated in Chapter 1 by developing a novel technique that segments identifiers into composing words and terms. The approach is based on a modified version of the dynamic time warping (DTW) algorithm proposed by Ney for connected speech recognition [Ney84] (*i.e.*, for recognizing sequences of words in a speech signal) and on the Levenshtein string edit-distance [Lev66]. It further assumes that there is a limited set of (implicit and–or explicit) rules applied by developers to create identifiers and therefore uses words transformation rules, plus a hill climbing algorithm [MF04] to deal with word abbreviation and transformation. Word transformation rules that programmers can apply involve dropping all vowels (*e.g.*, *pointer* becomes *pntr*), and dropping one or more characters (*e.g.*, *pntr* becomes *ptr*).

Our goal is to provide a meaning to simple and composed identifiers, even in presence of such truncated/abbreviated words, *e.g.*, *objectPrt*, *cntr* or *drawrect*, by associating identifier (substrings) to terms and words; words belonging to either a

full English dictionary or to a domain-specific or an application-specific dictionary. The approach takes as input two artifacts (i) the set of identifiers to be segmented into words and terms and (ii) a dictionary containing words and terms belonging to an upper ontology, to the application domain, or both. Examples of dictionaries used by our first approach are, for example, WordNet or dictionaries used by spell checkers, such as a-spell. Each dictionary entry my be matched to a sequence of known abbreviations in a way similar to a thesaurus. For example, the *counter* word in the dictionary can be associated to abbreviations *cntr, ctr* found as terms forming identifiers. Thus, if *cntr* is matched, the algorithm can expand it into the dictionary term *counter*. The overall idea is to identify near optimal matching between substrings in identifiers and words belonging to the dictionary, using an approach inspired by speech recognition.

3.2 Approach Components

The key components of our identifier segmentation algorithm are DTW, word transformation rules, and hill climbing algorithm. In the next sections, we will present each component of our approach and shed light on the main advantages and drawbacks of applying DTW for identifier splitting.

3.2.1 Dynamic Programming Algorithm

Dynamic time warping (DTW) is an algorithm that was conceived for time series alignment. It measures similarity between two sequences which may vary in time or speed and analyze data which can be represented linearly. The distance between two series after warping is calculated. This distance measures how well the features of a new unknown sequence match those of reference template. DTW relies on a dynamic technique to compare point by point two series by building a matrix. It will build this matrix staring from bottomleft corner *i.e.*, the beginning of the time series. Each neighboring cell in the matrix is taken and the previous distance is added to the value of the local cell. The value in the topright cell contains the

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 3.1: Single word edit distance example

distance between the two strings that has the shortest path in this matrix (Lachlan 2007). One of its well known applications has been automatic speech recognition, to cope with different speaking speeds.

Speech recognition is the ability of a machine or a program to recognize almost anybody's speech words and phrases in spoken language. Its applications include call routing, speech-to-text, voice dialing and voice search. The term "speech recognition" is sometimes used interchangeably with "voice recognition". However, the two terms have different meanings. In fact, speech recognition is used to identify words in spoken language. However, voice recognition is a biometric technology used to identify a particular individual's voice. In this research study, we have performed our identifier splitting via an adaptation of the connected speech recognition algorithm proposed by Ney [Ney84] that, in turns, extends to connected words the isolated word DTW [SC78] algorithm.

3.2.2 String Edit Distance

The string-edit distance between two strings, also known as Levenshtein distance [Lev66], 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 each insertion and deletion are assumed to increase the distance between the two strings by one, whereas a substitution (*i.e.*, a deletion followed by one insertion) increases it by two [CLR90].

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 the algorithm efficiency. A naive implementation is typically exponential in the string length. A quadratic complexity implementation can be easily realized 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 3.1.

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 $D(i-1,j)$, $D(i-,j-1)$, 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.

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

			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
	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 3.2: Multiple words edit distance example

Unfortunately, the Levenshtein algorithm is not suitable to split identifiers because it only computes the distance between two given strings, not between the terms in a string (*i.e.*, identifier terms) and some other strings (*i.e.*, dictionary words). The details of Ney's algorithm are available in [Ney84].

We implemented an extension of the Levenshtein algorithm based on Ney's adaptation. This extension requires a dictionary (or a thesaurus) of known words (referred to as *speech template* in [Ney84, SC78]).

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 global minimum distance between the identifier *pntrcntr* and the dictionary entries *ptr* and *cntr* is calculated as follows. Initialization of distance matrices is performed as described above for the Levenshtein algorithm, except that one matrix is created for each word in the dictionary, as shown in Figure 3.2. 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 two in Figure 3.2 and the corresponding best term, *i.e.*, *ptr*, is then recorded.

At the beginning of column three computation (*i.e.*, to calculate $(3, 2)$), the

algorithm checks if it is less costly to move from one of cells $(1, 2)$, $(1, 3)$, $(2, 2)$ or instead, if 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). 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*. However, when the column of the character *c* of *pntrcntr* is computed (column six), the minimum distance recorded for dictionary terms at column five is one, as *ptr* just needs a character insertion to match *pntr*. Thus, the computation propagates the minimum distance in column five for *ptr*, *i.e.*, *ptr* matches *pntr* with distance one or in other words the algorithm detected that the word *ptr* ends at column five. Because the character *c* is matched in *cntr*, the distance 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 to compute the minimum edit distance between the terms in the identifier and the two words, and thus split the identifier.

3.2.3 Word Transformation Rules

The previous techniques that derive concepts from identifiers do not deal with word transformations, e.g., abbreviation of *image* into *img*. To overcome this shortcoming, we have developed a set of transformation rules attempting to mimic the cognitive processes of developers when composing identifiers with abbreviated forms. These abbreviated forms may be not part of the dictionary and need to be either generated from existing dictionary entries or added to it. Moreover, several words may have the same (minimum) distance from the substring to be matched when matching a substring of the identifier to the dictionary words to determine which candidate must be selected (the word having the minimum distance from the substring).

Let us consider the identifier *fileLen* and suppose that the dictionary contains the words *length*, *file*, *lender*, and *ladder*. Clearly, the word *file* matches with zero

distance 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 dropped. Finally, 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 generate the missing dictionary entry *len*.

To choose the most suitable word to be transformed, we use the following heuristic. We select the closest words, with non-zero distance, to the substring to be matched and repeatedly transform them using transformation rules chosen randomly among six possible rules. This process continues until a transformed word matches the substring being compared or when transformed words reach a length shorter than or equal to three characters. The available transformation rules are the following:

- *Delete all vowels:* All vowels contained in the dictionary word are deleted, *e.g.*, *pointer* $\rightarrow$ *pntr*;
- *Delete Suffix:* suffixes—such as *ing*, *tion*, *ed*, *ment*, *able*—are removed from the word, *e.g.*, *improvement* $\rightarrow$ *improve*;
- *Keeping the first n characters only*: the word is transformed by keeping the first n characters only, *e.g.*, *rectangle* $\rightarrow$ *rect* for $n = 4$;
- *Delete a random vowel:* one randomly chosen vowel from the word is deleted, *e.g.*, *number* $\rightarrow$ *numbr*;
- *Delete a random character: i.e.*, one randomly-chosen character is omitted, *e.g.*, *pntr* $\rightarrow$ *ptr*.

The transformations are applied in the context of a hill-climbing search. Hill climbing is a heuristic which belongs to the family of local search. The algorithm [MF04] searches for a (near) optimal solution of a problem by moving from the current solution to a randomly chosen, nearby one, and accepts this solution only if

it improves the problem fitness (the distance in our case). The algorithm terminates when there is no moves to nearby solutions improving the fitness.

In the following, we will describe the steps of the hill climbing algorithm that we have used to deal with words transformations. In this algorithm, a transformed word is added to the dictionary if and only if it reduces the global minimum distance.

1. Based on the current dictionary, we (i) split the identifier using DTW as previously explained in this chapter, (ii) compute the global minimum distance between the input identifier and all words contained in the dictionary, (iii) associate to each dictionary word a fitness value based on its distance computed in step (ii). If the minimum global distance in step (ii) is zero, the process terminates successfully; else

2. From dictionary entries with non-zero distance obtained at step (1), we randomly select one word having minimum distance and then:

   (a) We randomly select one transformation not violating transformation constraints, apply it to the word, and add the transformed word to a temporary dictionary;

   (b) We split the identifier via DTW and the temporary dictionary and compute the minimum global distance. If the added transformed word reduces the global distance, then we add it to the current dictionary and go to step (1); else

   (c) If there are still applicable transformations, and the string produced in step (a) is longer then three characters, we go to step (a);

3. If the global distance is non-zero and the iteration limit was not reached, then, we go to step (1), otherwise the program exit with failure.

In steps (a) and (b), our algorithm attempts to explore as much as possible of neighboring solutions by applying word transformations.

Thus, hill climbing, DTW and word transformation rules are the key components of our identifier segmentation algorithm.

3.3 Discussion

One of the limitations of DECOS is the usage of complex techniques inspired from dynamic programming and speech recognition techniques. In fact, DTW has several advantages and drawbacks when applied for identifier splitting. Among its advantages, we can state:

- DTW algorithm provides a distance between an identifier and a set of words in a dictionary even if there is no perfect match between substrings in the identifier and dictionary words; for example, when identifiers are composed of abbreviations, *e.g.*, *getPntr*, *filelen*, or *DrawRect*. It accepts a match by identifying the dictionary words closest to identifier substrings;

- DTW algorithm has the ability of performing an alignment when matching words from the dictionary, thus it is able to work even when the word to be matched is preceded or followed by other characters, *e.g.*, *xpntr*. It is therefore better than, for example, applying only the Levenshtein edit distance;

- DTW algorithm shows how close the match is to the unknown string by assigning a distance to matched substrings. For example, if we consider the identifier *fileLen*, we would discover that *file* matches the first four characters with a zero distance (thus $distance = 0$) and that *length* matches the five to seven characters (at $distance = 3$);

- The algorithm can favor the matching of the longest words, with respect to multiple words composing the longest one if the dictionary is sorted. Thus, the identifier *copyright* would be matched to the word *copyright* rather than to the composition of words *copy* and *right*, which also belong to the dictionary.

However DTW has also some disadvantages:

- The first disadvantage is its intrinsic quadratic complexity of a single match with a cubic cost when we consider a dictionary;

- The second drawback is the non-determinism in the way in which words to be transformed are chosen. This can be justified by the fact that sentence syntax and semantics are not involved as matching is done at the character level. A typical case of this non determinism is the identifier *fileLen* for which the substring *length* should be preferred over *lender* instead of choosing between the two as DTW does;

- DTW can not disambiguate complex situations leading to optimal non-zero distance split. Indeed, it is immediate to recognize the component words *image* and *edges* in the identifier *imagEdges*. However, *image* and *edges* match the identifier with a distance of 1 because the E character is shared by both terms in the identifier and, thus, the optimal minimum cost is 1 and not 0. The proposed approach deals with similar disadvantages by transforming words and running multiple times the DTW algorithm to build multiple candidate splittings. Clearly, any developer would use syntax and semantics as well as her knowledge of the domain and context implicitly: even if *imag* is not a well-formed English word, she will correctly split *imagEdges* into *image* and *edges*.

We share with previous works the goal of automatically splitting identifiers into component terms. However, we do not assume the use of neither Camel Case conventions nor a set of known prefixes or suffixes. In addition, DECOS automatically generates a thesaurus of abbreviations using transformation rules attempting to mimic the developers' cognitive processes when building identifiers.

In the next chapter, we will report the results obtained by DECOS, and describe the case study that we have performed for validation and performance analysis of our technique.

CHAPTER 4

DECOS RESULTS

This chapter shows in detail the empirical study that we have performed to evaluate the performance of DECOS in terms of correctly mapping identifiers to concepts, the research questions that we have addressed and the splitting results that we have obtained. It also reveals the threats to validity related to our case study.

4.1 Case Study: Correctness of Mapping Identifiers

To examine the behavior of the first suggested approach, we have conducted an empirical study having the following definition:

The *goal* of this study is to analyze the proposed identifier splitting approach, with the *purpose* of evaluating its ability to adequately identify dictionary words composing identifiers, even in presence of word transformations or abbreviations.

The *quality focus* is the precision and recall of the approach when identifying words composing the identifiers with respect to manually-built oracles.

The *perspective* is of researchers, who want to evaluate an approach for identifier splitting, that 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 words or in general to meaningful words, *e.g.*, words belonging to a requirements' dictionary.

The *context* consists of a dictionary and identifiers extracted from the source code of two software systems, JHotDraw and Lynx. The dictionary contains about 2,500 words extracted from a glossary found on the Internet [1], 500 most frequent English words [2], plus terms and words contained in Lynx and JHotDraw. *JHotDraw* [3]

[1] `http://www.matisse.net/files/glossary.html`
[2] `http://www.world-english.org/`
[3] `http://www.jhotdraw.org`

Table 4.1: Main characteristics of the two analyzed systems

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

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* [4] is known as "the textual Web browser", *i.e.*, it is a free, open-source, text-only Web browser and Gopher client for use on cursor-addressable, character cell terminals. Lynx is entirely written in C. Its development began in 1992 and it is now available on several platforms, including Linux, UNIX, and Windows. Table 4.1 reports some relevant figures about the two systems that we analyzed.

4.2 Research Questions

The main research questions that we have addressed are as follows:

1. **Research Question 1 (RQ1):** *What is the percentage of identifiers correctly split by DECOS?* This research question investigates the overall performance of our approach, comparing the results with the oracle that we have manually built.

2. **Research Question 2 (RQ2):** *How does DECOS perform compared with the Camel Case splitter?* This research question compares the performance of the proposed approach with the simple Camel Case splitter, specifically the capability of correctly splitting identifiers and of mapping substrings to dictionary words.

[4] `http://lynx.isc.org/`

3. **Research Question 3 (RQ3):** *What percentage of identifiers containing word abbreviations is DECOS able to map to dictionary words?* This research question evaluates the ability of the proposed approach to map identifier substrings to dictionary words when these substrings represent abbreviations of dictionary words.

4.3 Analysis Method

The above research questions aim at understanding if DECOS helps in decomposing identifiers. Thus, we implicitly assume that, given an identifier, there exists an exact subdivision of this identifier into terms and words that, possibly after transformations and once concatenated, compose the identifier. First, we limited our analysis to identifiers longer than or equal to three characters: 2,348 in JHotDraw and 12,194 in Lynx. We have explicitly split identifiers containing digits, *e.g.*, *name4Tag* into *name* and *tag* and *sent2user* into *sent* and *user*, because our approach cannot map *2* to the word *to* and *4* to *for*, which are the intended meanings of these terms.

To evaluate our approach, we selected the 957 JHotDraw and 3,085 Lynx composed identifiers for which it was possible to define a segmentation. We excluded from our analysis identifiers that were composed of one single English word, and identifiers for which it was not possible to clearly identify a splitting into dictionary words and an expansion of abbreviations. Examples of identifiers belonging to such a category are some identifiers extracted from Lynx source code, *e.g.*, *gieszczykiewicz*, *hmmm*, *ixoth*, *pqrstuvwxyz*, or *tiocgwinsz*. The 957 (3,085, respectively) identifiers were manually segmented into composing substring mapped into words and terms, thus, creating oracles for JHotDraw and for Lynx.

RQ1 aims at answering a preliminary research question about the applicability and usefulness of the proposed approach. To answer **RQ1**, we followed a two-steps approach. First, we executed the proposed algorithm in a single iteration mode and with no transformations. Thus, only identifiers composed of dictionary words

are split with zero distance. Not-split identifiers, *i.e.*, with splitting distance not equal to zero, were fed into the second phase. In the second phase, we applied our approach with an upper bound of 20,000 iterations, *i.e.*, 20,000 dictionary word transformations and DTW splits. We chose 20,000 iterations as we noticed that after such a number of iterations, the approach was almost always able to find a splitting in a reasonable time, *i.e.*, within 2 minutes with our dictionary composed of 3,000 words. After automatic splitting have been performed, results are compared against the oracle, to compute the percentage of correctly segmented identifiers.

In phase two, we only included those identifiers that were not split in phase one and for which the composing substrings were longer than or equal to three characters, as shorter substrings were conservatively considered as spurious characters, pre-/post-fix or errors, thus penalizing our approach. Also, matching such short identifiers by performing transformations of dictionary words would not be feasible as too many dictionary words, after a sequence of transformations, would match the (short) substrings. For example, in the identifier *fpointer* the character *f* can be generated by any dictionary words containing the letter *f*. Much in the same way, the substring *ly* in Lynx identifiers such as *lysize* can be expanded to several different words.

RQ2 aims at performing a comparison of DECOS with the Camel Case splitting approach. We implemented a basic Camel Case identifier splitting algorithm and compared its results with the manually-built oracle. To statistically compare percentage of correct splittings performed by the proposed approach with those of the Camel Case splitter, we use Fisher's exact test [She07] and tested the null hypothesis H_0: *the proportions of correct splittings obtained by the two approaches are not significantly different.*

To quantify the effect size of the difference between the two approaches, we also computed the *odds ratio* (OR) [She07] indicating the likelihood of an event to occur, defined as the ratio of the odds p of an event occurring in one sample, *i.e.*, the percentage of identifiers correctly split by our approach (experimental group),

Table 4.2: Percentage of correct classifications (RQ1)

Systems	Identifiers	Exact Splittings		Errors
		Single Iteration	Multiple Iterations	
JHotDraw	957	891 (93%)	920 (96%)	37
Lynx	3,085	2,169 (70%)	2,901 (94%)	217

to the odds q of it occurring in the other sample, *i.e.*, the percentage of identifiers correctly split by the Camel Case splitter (control group): OR $= \frac{p/(1-p)}{q/(1-q)}$. An odds ratio of 1 indicates that the event is equally likely in both samples. OR > 1 indicates that the event is more likely in the first sample (proposed approach) while an OR < 1 indicates the opposite (Camel Case splitter).

RQ3 aims at assessing the ability of DECOS to find identifiers splittings when component substrings are obtained by means of dictionary word transformations, such as in *rectpntr* using *pntr* instead of *pointer* and *rect* in place of *rectangle*. **RQ3** is addressed similarly to **RQ1**, comparing identifiers matched in phase two (as explained for **RQ1**) with the subset of the identifiers in the oracle that, according to our manual classification, contained abbreviations.

4.4 Study Results

This section reports results of the empirical study with the objective of addressing the already mentioned research questions.

RQ1: What is the percentage of identifiers correctly split by DECOS?

Table 4.2 reports for JHotDraw and Lynx the results of the identifier splittings obtained with our approach. In particular, the third column reports the number of identifiers exactly split in a single step, *i.e.*, with DTW distance zero and matching the oracle. Results indicate that, for both systems, a large percentage of identifiers have been created via simple concatenations of dictionary words. In fact, 93% of JHotDraw identifiers, and 70% of Lynx identifiers have been exactly split into dictionary words within a single iteration of our algorithm.

The fourth column cumulates results of the third columns with the number

Table 4.3: Performance of the Camel Case splitter

Systems	Correct Splitings	Errors
JHotDraw	674	83
Lynx	361	2,524

of composed identifiers made of dictionary words abbreviations split with zero distance within 20,000 iterations. In other words, the fourth columns shows the numbers and percentages of all the correctly-split identifiers. Finally, the fifth column shows the number of identifiers that were not exactly split or for which the splitting did not match the oracle.

Wrong splittings were due to identifiers containing acronyms or short abbreviations. For example, we believe that it is impossible to identify correctly component words of the acronyms such as *afaik*, *imho*, or *foobar*. For different reasons, we also believe that it is impossible to find the exact splittings of identifiers such as *fsize*; even if we consider that the context of the identifiers *fsize* could be reasonably associated with both concepts of *file size* and *figure size* depending on the JHotDraw code region where it is used, even though the letter *f* really means that the field is private.

Overall, about 96% of JHotDraw identifiers and 93% of Lynx identifiers were correctly segmented with zero distance. These results support our claim and conclusion that a very large fraction (above 90%) of identifiers can be exactly split by using our approach.

RQ2: How does DECOS compares to the Camel Case splitter?

Table 4.3 summarizes results of Camel Case splitting. Not surprisingly, the Camel Case approach works well on JHotDraw. Indeed, Java coding guidelines and identifier construction rules tend to promote Camel Case splitting and JHotDraw developers carefully followed coding standards and identifier creation rules. As the second line of the same table shows, this is not the case of Lynx, the C Web browser. Indeed, C coding standards such as the Indian Hill[5] coding standards or

[5] `http://www.chris-lott.org/resources/cstyle/`

Table 4.4: JHotDraw: results and statistics for selected identifiers in ten splits attempts. 25%, 50% and 75% indicate the first, second (median), and third quartiles of the results distribution respectively

Identifiers	Successes	Min.	25 %	50 %	75 %	Max.	Split I	Split II
bord**dec**	yes	208	617	1,346	1,938	8,831	bord decimal	bord decision
anchor**len**	yes	154	689	1,220	3,097	7,056	anchor length	anchor lender
draw**rect**	yes	29	779	2,385	4,877	8,629	draw rectangle	
drawround**rect**	yes	77	6,509	10,300	17,403	19,173	draw round rectangle	
fill**rect**	yes	898	3,549	5,942	10,932	12,659	fill rectangle	
javadraw**app**	yes	86	480	972	4,582	6,965	java draw apply	java draw append
net**app**	yes	76	788	1,529	4,183	7,394	net apply	net append
new**len**	yes	176	534	600	704	2,503	new length	new lender
nothing**app**	yes	90	305	1,425	4,803	9,956	nothing apply	nothing application
addcolumn**info**	yes	457	1,296	1,806	2,631	4,146	add column information	add column inform
add**lbl**	yes	43	793	1,130	3,498	4,843	add label	
case**comp**	yes	124	327	437	938	1,836	case compare	case complete
serialversion**uid**	No						serial version did	
selection**z**ordered	No						selection ordered	
jhotdraw	No						hot draw	
fimagewidth	No						him age width	
fimageheight	No						him age height	
write**ref**	No						write red	

the GNU coding standards[6] do not enforce Camel casing.

When comparing the performances of DECOS (see Table 4.2, considering results after the second phase, *i.e.*, the third column) with those of the Camel Case splitting (see Table 4.3), the Fisher's exact test indicated no significant (or marginal) difference for JHotDraw (p-value = 0.1) with a OR = 1.3, *i.e.*, the proposed approach has chances of correctly splitting an identifier 1.3 more times than the Camel Case splitter. For Lynx, differences are statistically significant (p-value < 0.001) and we have an extremely high OR=60, *i.e.*, chances of our approach to correctly split identifiers are 60 times higher than the Camel Case splitter.

Therefore, we conclude that DECOS performs better than Camel Case splitter on both systems and significantly better on Lynx.

RQ3: What percentage of identifiers containing word abbreviations is DECOS able to map to dictionary words?

Table 4.2 and 4.4 reports results aimed at addressing **RQ3**. The fourth and fifth columns of Table 4.2 show that a substantial fraction of identifiers containing abbreviations can be split into dictionary words that originate such abbreviations.

[6] `http://www.gnu.org/prep/standards/`

More precisely, 44% and 70% of JHotDraw and Lynx identifiers containing abbreviations were correctly split into component words. The percentage of success for the two systems is quite different and the reason is the different ways in which identifiers have been composed. Indeed, in Lynx, very short prefixes are much more frequent and cryptic than in JHotDraw. In particular, Lynx prefixes, such as *ly*, *ht*, or *hta*, make it hard to produce correct splittings without a specialized dictionary in which such prefixes are added with, possibly, the proper expansion. Thus, a solution could be to augment dictionaries used by DECOS with domain knowledge, *i.e.*, acronyms, and well-known abbreviations.

4.5 Threats to Validity

Threats to *construct validity* concern the relation between the theory and the observation. Here, this threat is mainly due to mistakes in the oracles. Indeed, we cannot exclude that errors are still present in the oracles, despite the corrections made and explained above. However, the discovered errors were less than 1% of the number of identifiers contained in the oracles, thus the presence of some errors would not greatly affect our results. Nevertheless, as the intent of the oracles is to explain identifiers semantics, we cannot exclude that a part of identifiers could have been split in different ways by the developers that originally created them.

Threats to *internal validity* concern any confounding factor that could influence our results. In particular, these threats can be due to subjectiveness during the manual building of the oracles. We attempted to avoid any bias in the oracles by using the same oracles and simple string matching when comparing Camel Case splitter with our approach. Furthermore, both oracles were built by the same researcher and manually verified by other two people. Whenever a disagreement was detected, a majority vote was taken. The size of the oracle was chosen large enough to ensure that even an error of a few percent in splits would not have affected algorithm comparison.

Threats to *Conclusion validity* concern the relationship between the treatment

and the outcome. Identifiers split exactly into dictionary words in a single iteration may sometime be split in a different way from the developers' intent. However, we do not claim any relation between the splitting produced and the semantics of the identifiers; this relation is left to the developers' judgment and experience. We limit ourselves to comparing our approach with the Camel Case splitter and validating the quality of computed splittings with respect to the oracles. Conclusion validity may play a role when we compared the effectiveness in detecting word abbreviations. To limit such a threat, we manually inspected all splittings produced with multiple iterations and word transformations.

Threats to *external validity* concern the possibility of generalizing our results. The study is limited to two systems: JHotDraw and Lynx. Yet, our approach is applicable to any other system. However, we cannot claim that similar results would be obtained with other systems. We have compared our approach with a Camel Case splitter but cannot be sure that their relative performances would remain the same on different systems. However, the two systems correspond to different domains and applications, have different sizes, are developed by different teams, with different programming languages. We believed this choice mitigates the threats to the external validity of our study.

4.6 Conclusion

To conclude this chapter, we have proposed DECOS: an approach inspired by Ney's extension of DTW algorithm to map identifiers to domain terms. DECOS uses word transformation rules and hill climbing heuristic to infer a segmentation in identifiers composed of dictionary words and also of word abbreviations.

We have applied DECOS to split identifiers of two systems, developed with different programming languages, and belonging to different application domains: JHotDraw and Lynx. Results have been evaluated comparing the obtained splittings with the manually-built oracles. They showed that DECOS outperforms a simple Camel Case splitter.

In particular, for Lynx, the Camel Case splitter was able to correctly split only about 18% of the identifiers versus 93% with our approach. On JHotDraw, the Camel Case splitter exhibited a correctness of 91% while our approach ensured 96% of correct results. DECOS was also able to map abbreviations to dictionary words, in 44% and 70% of cases for JHotDraw and Lynx, respectively.

To deal with the challenge of mapping C identifiers to concepts, we propose TIDIER that we present in the next Chapter.

CHAPTER 5

TIDIER: EXTENSION OF DECOS

In this chapter, we present TIDIER, our contextual approach of identification of concepts in source code. TIDIER is based on the same components of DECOS, it deals with C identifiers and uses context when mappings identifiers to concepts. The first section of this chapter introduces TIDIER, the second is devoted to the way we have built our thesaurus of words and abbreviations and the last one briefly recalls a typical run of TIDIER and some of its limitations.

5.1 TIDIER Contributions

DECOS was extended into TIDIER (Term IDentifier RecognIzER), an approach and tool that implements the components previously described in Chapter 3. TIDIER overcomes the limitations of Camel Case splitting and Samurai, in particular for C programs. In fact, Camel Case splitting and Samurai have very good performance when applied on Java programs because Java developers usually adhere strictly to the Camel Case convention. However, these approaches are not effective on C identifiers [MGD+10, EHPVS09].

This work extends our previous approach [MGD+10] with several contributions among which we can state:

1. A detailed description of our tool, TIDIER which is an extension of DECOS;
2. An extensive validation of TIDIER on identifiers randomly extracted from a large set of C programs;
3. Comparison of TIDIER with the two existing approaches: Camel Case splitter and Samurai (the comparison is detailed in Chapter 6);
4. Evidence on the relevance of contextual information as well as specialized knowledge in the splitting process.

The main research questions that we have address are the following:

1. **Research Question 1 (RQ1):** *How does TIDIER compare with alternative approaches, Camel Case splitting and Samurai, when C identifiers must be split?*

2. **Research Question 2 (RQ2):** *How sensitive are the performances of TIDIER to the use of contextual information and specialized knowledge in different dictionaries?*

3. **Research Question 3 (RQ3):** *What percentage of identifiers containing word abbreviations is TIDIER able to map to dictionary words?*

To answer these research questions, we analyzed a set of 340 open source programs: 337 programs from the GNU repository, two operating systems (the Linux Kernel release 2.6.31.6 and FreeBSD release 8.0.0), and the Apache Web server release 2.2.14. We extracted identifiers (including function, parameter, and structure names) and comments from these programs. We manually built an oracle of 1,026 identifiers randomly extracted from the 340 programs and not being plain English words or well-known terms. The oracle was built by two other researchers because of the large number of applications that we had to deal with.

Using this oracle, we report evidence of the superiority of TIDIER over Camel Case splitting and Samurai for C identifiers (**RQ1**). We show the usefulness to TIDIER of contextual information and specialized knowledge (**RQ2**). We also provide supporting evidence that TIDIER also successfully split identifiers abbreviations in about 48% of cases (**RQ3**).

As in Chapter 3, 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*. We recall that a term may or may not be a dictionary word and 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

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

meanings to words).

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 dictionary. It relies on 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. 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.

C Developers sometimes compose identifiers using abbreviations, this is probably an 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 at segmenting identifiers into terms and recovering the original non abbreviated words. For this reason, it 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. For example, when the abbreviation *ptr* is detected TIDIER knows that this is actually an abbreviation of *pointer*. In this chapter, we use the terms dictionary and thesaurus interchangeably to indicate a list of words each one with, possibly, an associated list of abbreviations.

Overall, the current implementation of TIDIER takes as input an identifier and a thesaurus and uses a simple string-edit distance to determine whether it is possible to split the identifier into a number of terms that have a small (or zero) distance with dictionary words.

Some abbreviations are well-known and can thus be part of the thesaurus. In such case, each row of the thesaurus contains a word and its possible synonyms, *e.g.*, *directory* and *dir*. Some other abbreviations may not appear in the thesaurus

because they are too domain and–or developer specific. To cope with such abbreviations, TIDIER behaviors similarly to the first approach. It assumes that there is a limited set of rules applied by developers to create identifiers and uses words transformation rules, plus a hill climbing algorithm to deal with word abbreviation and transformation. The 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 (the transformed word is equal or less than 3 characters) is reached.

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 because 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*.

The main components of TIDIER are therefore the string edit Levenshtein distance, word transformation rules, and the thesaurus of words and abbreviations that will be the topic of the next section.

5.2 TIDIER Main Component: Thesaurus of Words and Abbreviations

The thesaurus used by TIDIER plays a crucial role for the quality of results. It is built with accordance to the steps described in the methodology section of Chapter 1. In this part, the focus is on the kind of dictionaries used to build it:

1. *Small English dictionary referred to as "English Dictionary"*: an English dictionary built from the 1000 most frequent English words, the 250 most frequent technical words (from Oxford dictionary) and 275 most frequent business words (from Oxford dictionary) plus words from a glossary found on

the Internet[2]. 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 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*: similarly to Enslen *et al.* [EHPVS09], dictionaries containing function level, source code file level, and application level identifiers. We built these dictionaries using words appearing in the context where the identifiers are located.

5. *Application dictionary, plus specialized knowledge*: a dictionary based on the application dictionary 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 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

[2] `http://www.matisse.net/files/glossary.html`

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 list of terms need to be pruned of strings not corresponding to English words or technical terms before being considered usable dictionaries; in TIDIER the filtering is done by string comparison with the WordNet dictionary.

As the splitting output depends on TIDIER dictionaries, they must be carefully validated. To validate a dictionary, we perform a manual validation for small dictionaries or highly-specific dictionaries, such the abbreviations. For large dictionaries, we filter words using a trusted reference dictionary, such as WordNet.

5.3 Typical Run of TIDIER

A typical run of TIDIER can be described as follows. First, wherever possible, identifiers are split using explicit separators, namely special characters, *e.g.*, "_", ".", "$", "->", and the Camel Case convention. (A description of the Camel Case splitter is given in Chapter 2). Then, TIDIER applies transformations and computes the distance between the identifier and the thesaurus words by using a hill climbing search. For a given identifier and a given dictionary, the edit distance assigns a distance to each thesaurus word as well as the positions where it begins and ends in the identifier. The edit distance is the fitness function guiding the hill climbing search following the same steps detailed in Chapter 3.

CHAPTER 6

TIDIER RESULTS

This chapter is devoted to the case study that we have conducted to analyze the results of our second contribution. First, we describe our case study, then we discuss how we have built our oracle and dictionaries. The third section states the research questions that we have addressed. The fourth part concerns the study design and is followed by a section dedicated to our analysis method. Finally, we conclude this chapter with a conclusion about TIDIER.

6.1 Case Study: Precision and Recall

Similarly to the first case study, the *goal* of this one is to analyze TIDIER with the *purpose* of evaluating its ability to adequately recognize 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 identifier-splitting approaches. The *perspective* is of researchers, who want to understand how the approach for identifier splitting 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 requirements' dictionary.

Differently form the context of our first case study, the *context* here 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

[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	–

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 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 systems; as the name suggest 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.

6.2 Research Questions

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

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

2. **Research Question 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. **Research Question 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.

6.3 Analysis Method

Before introducing our analysis method, we present the planning and study design related to the empirical study that we have performed.

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

1. Camel Case splitter;

2. Samurai approach;

3. TIDIER approach.

The second independent variable is the used dictionary (or set of used dictionaries), among those defined in Section 5.2. Thus, we have a number of possible

treatments equals to the number of different dictionaries plus two, *i.e.*, the two alternative approaches: Camel Case splitter and Samurai.

The first dependent variable considered in our study is the *correctness* of the splitting/mapping to dictionary words produced by the identifier-splitting approach with respect to the oracle. As a first, coarse-grained measure, we use a Boolean variable meaning that the splitting is correct (true) or not (false).

Let us suppose that we define the correct splitting of the identifier *cntrPtr* as *counter* and *pointer*; if the studied approach produces exactly the expected splitting, then the correctness is evaluated as 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. If the split is *almost* correct, *i.e.*, most of the terms are correctly identified, then correctness would still be zero.

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 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:

$$\text{F} - \text{Measure} = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

The part devoted to our analysis method try to address the research questions formulated in 6.2.

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 method is the same for both research questions.

We test the differences among different approaches using the Fisher's exact test because correctness is a categorical measure. We test the following null hypothesis H_0*: the proportion of correct splittings,* 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 also compute the odds ratio (OR) similarly to our first approach.

Precision, recall, and F-Measure are compared using a non-parametric test for pairwise median comparison, specifically the Wilcoxon paired test. We use a paired test as our samples are dependent, as we compute, for each identifier, the precision, recall and F-Measure for the different approaches. The Wilcoxon test tests 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.

We quantify the effect size of the difference using the Cohen d effect size for dependent variables, defined as the difference between the 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 effect size is considered small for $0.2 \leq d < 0.5$, medium for $0.5 \leq d < 0.8$, and large for $d \geq 0.8$ [Coh88]. We chose the Cohen d effect size because it is appropriate for our variables (in ratio scale) and because the different levels (small, medium, and large) are easy to interpret.

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 must be corrected. We used the Holm correction [Hol79], which 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

Table 6.2: Descriptive statistics of F-Measure

Method	Dictionary	1Q	Median	3Q	Mean	σ
CamelCase		0.00	0.40	1.00	0.44	0.43
Enslen		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

$n - 1$, and so on.

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. We identified the set of likely abbreviations in our sample as follows:

1. For each identifier, *e.g.*, *counterPtr*, we consider the split performed using the Camel Case splitter, *i.e.*, *counter ptr*, and the oracle, *i.e.*, *counter pointer*;

2. Then, we compare 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 split does not match with the one in the oracle, (ii) both terms start with the same letter, (iii) the term in the split does not appear in the English dictionary of 2774 words, and (iv) the term in the oracle appears in the English dictionary, we consider the term in the split 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 counted the percentage of abbreviations correctly mapped to dictionary words. We also computed the set

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

Approach 1	Approach 2	p-values	ORs
Camel Case	Samurai	0.63	0.95
English dictionary	Camel Case	**0.01**	0.73
English dictionary	Samurai	**0.01**	0.69
English dictionary	WordNet	1.00	0.95
English dictionary + domain kn.	Camel Case	<**0.001**	1.53
English dictionary + domain kn.	Samurai	<**0.001**	1.46
English dictionary + domain kn.	English dictionary	<**0.001**	2.13
Application	Camel Case	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.	Camel Case	<**0.001**	2.76
Application + Domain kn.	Samurai	<**0.001**	2.62

of abbreviations that were not correctly mapped, but with distance one from the oracle, *i.e.*, the mapping failed for a single character only. Thus, we identified and can discuss cases where the approach almost found the correct solution, even though it failed to correctly converge.

6.4 Study Results

We now present and discuss the results of our study to answer the research questions formulated in Section 6.2. Raw data of our study are available for replication purposes[5].

First, we evaluate the correctness of TIDIER when using different dictionaries and compare it with that of the two alternative approaches, *i.e.*, the Camel Case splitter and Samurai. The percentage of correctly split/map identifiers is reported in Figure 6.1.

The two bars at the bottom of the figure show the performances of the Camel Case splitter and Samurai, respectively, while the other bars show the performances

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

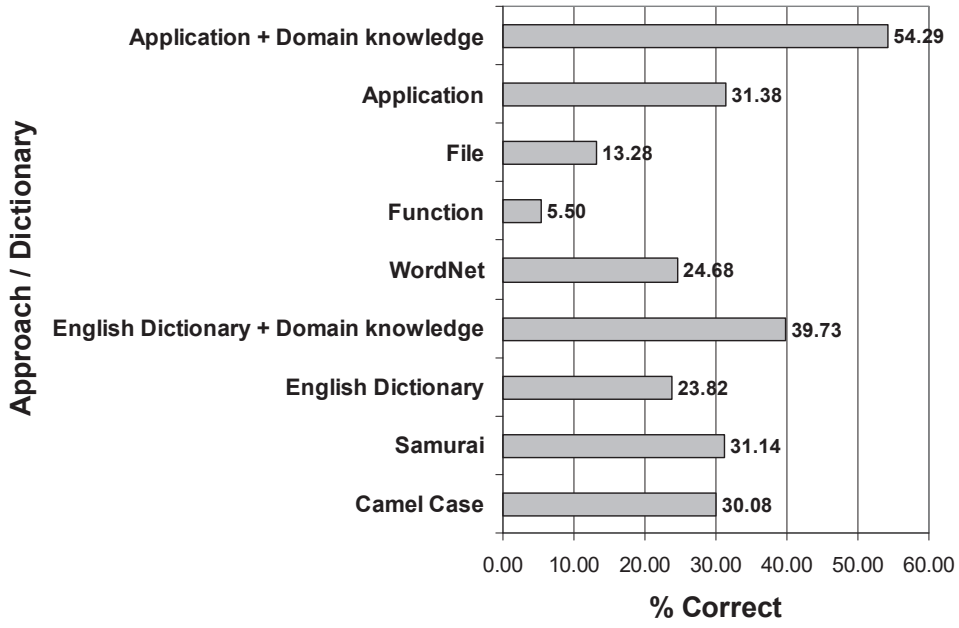

Figure 6.1: Percentages of correct identifier splittings

of TIDIER when using different dictionaries.

Table 6.3 reports 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, Samurai performs nearly as well as the Camel Case splitter and there are no statistically significant differences among them;
- When using only the simple English dictionary, TIDIER performs worse than the Camel Case splitter and Samurai. The percentage of correctly split identifiers is only 23.82%, while the Camel Case splitter 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 *OR*s of about 1.5 in favor of TIDIER wrt. the Camel Case splitter and Samurai;

- 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 set of terms that they capture;

- When adding domain knowledge to the program-level dictionary, TIDIER performs best with 54.29% of correct splittings, significantly higher than the alternative approaches and than when using the English dictionary. *OR*s are 2.76 and 2.62 times in favor of TIDIER wrt. the Camel Case splitter and Samurai respectively and 1.80 wrt. using the English dictionary with domain knowledge.

Table 6.2 shows the descriptive statistics (first quartile, median, third quartile, mean, and standard deviation) of the F-Measure computed as explained in Section 6.3 to evaluate the capability of the approaches to correctly and completely identify terms part of the identifiers. We do not show results of precision and recall separately 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.

Table 6.4 reports corrected results of the paired Wilcoxon test and the Cohen d effect size (positive values of d are in favor of Approach 1, negative values are

Table 6.4: Comparison among approaches: results of Wilcoxon paired test and Cohen d effect size

Approach 1	Approach 2	p-value	ORs
Camel Case	Samurai	**<0.001**	−0.15
English dictionary	Camel Case	**<0.001**	−0.12
English dictionary	Samurai	**<0.001**	−0.19
English dictionary	WordNet	**<0.001**	−0.11
English dictionary + domain kn.	Camel Case	**<0.001**	0.29
English dictionary + domain kn.	Samurai	**<0.001**	0.22
English dictionary + domain kn.	English dictionary	**<0.001**	0.61
Application	Camel Case	**<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.	Camel Case	**<0.001**	0.58
Application + Domain kn.	Samurai	**<0.001**	−0.51

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, < 0.2;
- when using the English dictionary with domain knowledge, TIDIER performs significantly better than the Camel Case splitter ($d = 0.29$) 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 (< 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 the Camel Case splitter 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 a program-level contextual dictionary augmented with domain knowledge is used.

Regarding **RQ2**, we conclude 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.

To answer **RQ3**, we ran five times TIDIER on each of the 73 abbreviations with 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%. Although this percentage is not high, to the best of our knowledge, TIDIER is the first and only approach able to deal with abbreviations.

The first block of Table 6.5 shows examples of abbreviations that were correctly mapped by TIDIER to dictionary words. Specifically, the table reports (i) the abbreviations, (ii) the oracle, (iii) the different mappings produced by TIDIER. The second block of Table 6.5 shows cases of wrong mapping, such as those of *auth* into *author* while the correct mapping was *authenticate*) or *dest* into *destroy* while the correct one was *destination*. Wrong mappings happen because TIDIER does not use semantic information thus it can generate mappings that are different from our oracle yet with a zero distance. Consistently with insights gained from **RQ1** and **RQ2**, wrong mappings suggest that domain-specific dictionaries can be useful to better support mapping of identifiers to concepts.

Out of the $73 - 35 = 38$ abbreviations not correctly mapped there are 16 identifiers wrongly mapped and 22 cases in which TIDIER was not able to produce a mapping with a zero distance. Some of these cases are shown in the third block of Table 6.5, where the numbers in parentheses report the achieved minimum distances. For example, *addr* was mapped to *add* instead of *address* with distance

Table 6.5: Examples of correct and wrong abbreviations

Match with the Oracle

Abbreviation	Oracle	Mapping 1	Mapping 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 Mappings

Abbreviation	Oracle	Mapping 1	Mapping 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	Mapping 1	Mapping 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)	

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** suggests that, indeed, TIDIER is able to deal with abbreviations used to build identifiers and can map them into dictionary words in 48% of the abbreviations considered in our sample. We claim that this result is promising because alternative approaches are not able to deal with abbreviations at all and because future work could improve the mappings, possibly using enhanced search heuristics.

6.5 Threats to Validity

This section discusses threats to the validity of our study that could impact its results.

Threats to *construct validity* concern the relation between the theory and the observation. This threat is mainly due to mistakes in the oracle. We cannot exclude that errors are present in the oracle. As the intent of the oracle is somehow 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.

Threats to *internal validity* concern any confounding factors that could have influenced our results. In particular, these threats are due to the subjectivity of 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 two of the authors independently and inconsistencies in splitting/mapping to dictionary words

were discussed. The size of the oracle was chosen large enough to ensure that even an error of a few percent would not affect dramatically the comparisons.

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 selected our sample of identifiers 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 [EHPVS09]. Differently from our previous work [MGD+10], we only consider C programs rather than Java program because Java identifiers are mostly built using the Camel Case convention and, quite often, using complete English words rather than abbreviations. Instead, the usage of a more complex splitting algorithm is particularly useful for the C programming language. Despite the sample size, we cannot exclude that performances would vary on other projects, *e.g.*, commercial source code, and programming languages.

6.6 Conclusion

We have presented TIDIER, a tool that is the extension of DECOS for mapping identifiers to concepts. TIDIER finds the minimum edit distance between the identifier terms and dictionary words. It can split identifiers even in the absence of explicit separator (*e.g.*, underscore or Camel Case convention) and deal with the

usage of abbreviations within identifiers in the context of a hill climbing search. The tool also seems to be useful to better assess the quality of identifiers or to identify identifiers refactoring actions.

TIDIER takes as input a thesaurus and an identifier to be mapped. It maps the identifier to domain terms that achieve the overall, minimum edit distance with respect to words (and their abbreviations) contained in the thesaurus. Abbreviation not present in the thesaurus are generated automatically using word transformation rules, mimicking developers' identifier creation process.

To quantify the performances of TIDIER, we applied it to split a set of 1,026 C identifiers randomly extracted from a corpus of 340 open source programs. The 1,026 identifiers were manually split into terms to build an oracle against which TIDIER, Camel Case splitter and Samurai [EHPVS09], were compared.

Reported 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 the domain knowledge 54% of correct splits, compared to 30% for the Camel Case splitter 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 present in the oracle.

CHAPTER 7

CONCLUSION

Our initial work introduced DECOS as a novel technique that maps identifiers to domains terms. DECOS is based on a modified version of DTW algorithm proposed by Ney for connected speech recognition and on the string edit-distance. It assumes that developers apply a limited set of rules to create identifiers and therefore uses words transformation rules, plus a hill climbing algorithm to deal with word abbreviation and transformation.

The extension of DECOS gave birth to TIDIER: an automatic tool that derives domain terms and thus concepts based on the analysis of source code identifiers. TIDIER relies on the same components of our first technique and is able to detect concepts that correspond to identifiers. Thus, it provides developers with hints that could help them comprehend programs during their understanding and maintenance activities. The power of DTW in revealing how close the match is to the unknown string could be an indicator about how program identifiers reflect terms in high level artifacts which can lead to a better assessment of the quality of identifiers.

To validate our work, we applied it to map identifiers of JHotDraw and Lynx and evaluate it by comparing its results with the manually-built oracles. Performance analysis showed that our first approach outperforms the Camel Case splitter. Unlike our previous work, we applied TIDIER not only to two applications but to a large corpus of open source programs and compared its results with those attained by previous alternatives. To see how sensitive TIDIER is to specialized knowledge, we enriched it by the use of domain knowledge and context-aware dictionaries. Reported results showed that, with program-level dictionaries augmented with domain knowledge, *i.e.*, common acronyms, abbreviations, and C library functions, TIDIER significantly outperforms the previous techniques.

CHAPTER 8

RESEARCH PLAN

In this section, we present an overview of the state of the proposed research, what has been completed, and what are the possible directions for future research. We also present a detailed plan for the possible publications to be produced from the obtained results.

8.1 State of the Research and Future Work

We have accomplished the following phases of our research:

- Study of the literature and existing related research work on the detection of concepts based on the analysis of source code and also research works that deal with identifiers and software quality;
- Development of a novel approach for mapping identifiers to domain terms, inspired from speech recognition techniques. The approach overcomes the shortcomings of previous approaches and can identify concepts that correspond to identifiers composed of transformed words, regardless of the kind of separators;
- Presentation of our first contribution in the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010) that took place in March 2010, in Madrid, Spain;
- Extension of our initial work into a novel approach called TIDIER and extensive validation of TIDIER on identifiers randomly extracted from a large set of C programs;
- Comparison between TIDIER, Camel Case splitter and Samurai in term of correctly mapping identifers to concepts;

- Evidence on the relevance of contextual information as well as specialized knowledge in the process of identification of concepts represented by identifers embedded in source code.

Future research directions towards the completion of the thesis requirements are as follows:

- To increase the accuracy of our results in term of choosing the domain terms that correspond to a given identifier. We plan to develop new word transformation rules that mimic the developers cognitive processes when building identifiers. We also want to develop a variant of our algorithm in which these transformation rules will be applied according to a determined priority instead of being randomly chosen;

- The proposed approach has a non-deterministic component in the way in which word transformation rules are applied and in the way in which the candidate words to be transformed are selected. This suggests the need for improving the heuristic used for the selection of the candidate word to be used when mapping an identifier to the appropriate concept;

- Many recent works proposed quality models for the assessment of software quality. However, to the best of our knowledge, none of these models consider informal information such as identifiers or comments. We suggest to integrate a new node called *identifiers* in one of such models namely, Software Quality Understanding through the Analysis of Design (SQUAD) [Kho09]. This would enable us to increase the accuracy of the results related to the prediction of the quality of a system. In fact, researchers [MPF08] have already shown that comments and identifiers if combined with existing structural cohesion metrics proves to be a better predictor of faulty classes when compared to different combinations of structural cohesion metrics.

- Our discussion with Dr.Andrian Marcus during MSR (Mining Software Repositories) summer school in June 2010, in Kingston, results in another research

direction aiming at taking advantage of our work when assessing software quality. Indeed, we can consider every identifier for which our approach was not able to find a mapping to a domain term, as a non well-formed identifier and then assess the impact of such identifiers on the quality of programs. This research idea could be empirically validated relying on non well-formed identifiers and Lines of code (LOC) as metrics for example.

8.2 Our Contributions

- A conference paper published in March 2010 and received the best paper award:

  [MGD+10] Nioosha Madani, Latifa Guerrouj, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. *Recognizing words from source code identifiers using speech recognition techniques*. Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010), March 15-18 2010, Madrid, Spain. IEEE CS Press, 2010;

- A journal paper submitted in June 2010:

  [GMGGon] Latifa Guerrouj, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. *TIDIER: An Identifier Splitting Approach using Speech Recognition Techniques*. Submitted for publication in the Journal of Software Maintenance and Evolution: Practice and Evolution (JSME), June 2010.

- A 4-page pdf of our research proposal was accepted, in July 2010, for publication in the proceeding of the 17th Working Conference on Reverse Engineering (WCRE 2010) that will take place in Boston, USA during 13-16 October 2010.

8.3 Publication Plan

We plan to publish possible upcoming publications of the research results in the following conferences and journal.

DATE	CONFERENCES	CONTRIBUTION
February 4^{th}, 2011	ICPC'11	An optimized TIDIER for the derivation of of concepts based on identifiers
April 6^{th}, 2011	ICSE'11	A hybrid heuristic based TIDIER for mapping source code identifiers to concepts
July 21^{st}, 2011	WCRE'11	A TIDIER assisted approach for the assessment of software quality
January 7^{th}, 2012	TSE	A complete TIDIER based on new word transformation rules mimicking developers when creating identifiers

Table 8.1: Publication plan

We wish to accomplish our thesis writing during *January* 7^{th}*, 2012 - July* 7^{th}*, 2012* and defend our thesis in *August 2012*.

BIBLIOGRAPHY

[ACC+02] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28:970–983, Oct 2002.

[AD05] Marcus Andrian and Poshyvanyk Denys. The conceptual cohesion of classes. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 133–142, 2005.

[AHM+08] Surafel L. Abebe, Sonia Haiduc, Andrian Marcus, Paolo Tonella, and Giuliano Antoniol. Analyzing the evolution of the source code vocabulary. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, pages 189–198, 2008.

[AL98] Nicolas Anquetil and Timothy Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, pages 213–222, December 1998.

[BDLM09] David Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. To camelcase or under_score. In *Proceedings of the 17th IEEE International Conference on Program Comprehension*, 2009.

[BMW93] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering*, pages 482–498, 1993.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introductions to Algorithms*. MIT Press, 1990.

[Coh88] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Lawrence Earlbaum Associates, 2nd edition edition, 1988.

[CT99] Bruno Caprile and Paolo Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 112–122, October 1999.

[CT00] Bruno Caprile and Paolo Tonella. Restructuring program identifier names. In *Proceedings of the 16th International Conference on Software Maintenance*, pages 97–107, 2000.

[DDN00] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems Languages and Applications*, pages 166–177, 2000.

[DDO10] Andrea De Lucia, Massimiliano Di Penta, and Rocco Oliveto. Improving source code lexicon via traceability and information retrieval. *IEEE Transactions on Software Engineering*, (to appear), 2010.

[DFOT07] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering traceability links in software artefact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology*, 16(4), 2007.

[DP05] Florian Deissenbock and Markus Pizka. Concise and consistent naming. In *Proceedings of the International Workshop on Program Comprehension*, May 2005.

[EHPVS09] Eric Enslen, Emily Hill, Lori L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *Proceedings of the 6th International Working Conference on Mining Software Repositories*, pages 71–80, 2009.

[FWG07] Beat Fluri, Michael Wursch, and Harald Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pages 70–79, 2007.

[GMGGon] Latifa Guerrouj, Di Penta Massimiliano, Antoniol Giuliano, and Yann-Gaël Guéhéneuc. Tidier: An identifier splitting approach using speech recognition techniques. *Journal of Software Maintenance and Evolution: Research and Practice*, May 2010 (submitted for publication).

[Hol79] Sture Holm. A simple sequentially rejective Bonferroni test procedure. *Scandinavian Journal of Statistics*, 6:65–70, 1979.

[JH06] Zhen Ming Jiang and Ahmed E. Hassan. Examining the evolution of code comments in postgresql. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 179–180, 2006.

[Kho09] Foutse Khomh. Squad: Software quality understanding through the analysis of design. In *Procedings of the 16th Working Conference on Reverse Engineering*, pages 303–306, 2009.

[LDOZ06] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Francesco Zurolo. Improving comprehensibility of source code via traceability information: a controlled experiment. In *Proceedings of 14th IEEE International Conference on Program Comprehension*, pages 317–326, 2006.

[Lev66] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, (10):707–710, 1966.

[LFB06] Dawn Lawrie, Henry Feild, and David Binkley. Syntactic identifier conciseness and consistency. In *Proceedings of the 6th IEEE Interna-*

tional Workshop on Source Code Analysis and Manipulation, pages 139–148, Sept 27-29 2006.

[LMFB06] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? a study of identifiers. In *Proceedings of 14th IEEE International Conference on Program Comprehension*, pages 3–12, Athens, Greece, 2006. IEEE CS Press.

[LMFB07] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3(4):303–318, 2007.

[MACHH05] Jonathan I. Maletic, Giuliano Antoniol, Jane Cleland-Huang, and Jane Huffman Hayes. 2005.

[MF04] Zbigniew Michalewicz and David B. Fogel. *How to Solve It: Modern Heuristics - (2nd edition)*. Springer-Verlag, Berlin Germany, 2004.

[MGD+10] Nioosha Madani, Latifa Guerrouj, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Recognizing words from source code identifiers using speech recognition techniques. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, March 2010.

[MM03] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the International Conference on Software Engineering*, pages 125–137, 2003.

[MMM03] Ettore Merlo, Ian McAdam, and Renato De Mori. Feed-forward and recurrent neural networks for source code informal information analysis. *Journal of Software Maintenance*, 15(4):205–244, 2003.

[MPF08] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented

systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.

[Ney84] Herman Ney. The use of a one-stage dynamic programming algorithm for connected word recognition. *IEEE Transactions on Acoustics Speech and Signal Processing*, 32(2):263–271, Apr 1984.

[PM06] Denys Poshyvanyk and Andrian Marcus. The conceptual coupling metrics for object-oriented systems. In *Proceedings of 22nd IEEE International Conference on Software Maintenance*, pages 469 – 478, 2006.

[SC78] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics Speech and Signal Processing*, 26(1):43–49, Feb 1978.

[SH98] Susan Elliott Sim and Richard C. Holt. The ramp-up problem in software projects: a case study of how software immigrants naturalize. In *Proceedings of the 20th international conference on Software engineering*, pages 361–370, 1998.

[She07] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.

[SIS02] Sahida Sulaiman, Norbik Bashah Idris, and Shamsul Sahibuddin. Production and maintenance of system documentation: What, why, when and how tools should support the practice. In *Proceedings of the 9th Asia-Pacific Software Engineering Conference*, 2002.

[TGM96] Armstrong Takang, Penny A. Grubb, and Robert D. Macredie. The effects of comments and identifier names on program comprehensibility: an experiential study. *Journal of Program Languages*, 4(3):143–167, 1996.

L'École Polytechnique se spécialise dans la formation d'ingénieurs et la recherche en ingénierie depuis 1873

École Polytechnique de Montréal

École affiliée à l'Université de Montréal

Campus de l'Université de Montréal
C.P. 6079, succ. Centre-ville
Montréal (Québec)
Canada H3C 3A7

www.polymtl.ca

ÉCOLE
POLYTECHNIQUE
MONTRÉAL