UNIVERSITÉ DE MONTRÉAL

HEURISTIC SPLITTING OF SOURCE CODE IDENTIFIERS

NIOOSHA MADANI

DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AVRIL 2010

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

HEURISTIC SPLITTING OF SOURCE CODE IDENTIFIERS

présenté par : MADANI, Nioosha

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de :

M. PESANT Gilles, Ph.D., président

M. ANTONIOL Giuliano, Ph.D., membre et directeur de recherche

M. DESMARAIS Michel, Ph.D., membre

ACKNOWLEDGMENT

I thank my supervisor, Dr. Giuliano Antoniol who was really helpful and offered invaluable assistance, support and guidance. Special thanks to Dr.Yann-Gaël Guéhéneuc for his supports and assistance; and my deepest gratitude to all my friends, especially Soccer-lab and Ptidej team members. I wish to express my love and gratitude to my husband and my beloved families; for their understanding & endless love, through the duration of my studies.

RÉSUMÉ

La maintenance regroupe l’ensemble des activités effectuées pour modifier un logiciel après sa mise en opérations. La maintenance est la phase la plus coûteuse du développement logiciel. La compréhension de programmes est une activité cognitive qui repose sur la construction de représentations mentales à partir des artefacts logiciels. Les développeurs passent un temps considérable à lire et comprendre leurs programmes avant d’effectuer des changements.

Une documentation claire et concise peut aider les développeurs à inspecter et à comprendre leurs programmes. Mais, l'un des problèmes majeurs que rencontrent les développeurs durant la maintenance est que la documentation est souvent obsolète ou tout simplement pas disponible. Par conséquent, il est important de rendre le code source plus lisible, par exemple en insistant auprès des développeurs pour qu’ils ajoutent des commentaires dans leur code et respectent des règles syntaxiques et sémantiques en écrivant les identificateurs des concepts dans leurs programmes. Mais certains identificateurs sont constitués de termes des mots qui sont abrégés ou transformés. La reconnaissance des termes composants les identificateurs n'est pas une tâche facile, surtout lorsque la convention de nommage n'est pas respectée.

À notre connaissance deux familles d’approches existent pour décomposer les identificateurs : la plus simple considère l’utilisation du renommage et la présence des séparateurs explicites. La stratégie la plus complète est implémentée par l’outil Samurai (Enslen, Hill et al. 2009), elle se base sur le lexique et utilise les algorithmes gloutons pour identifier les mots qui constitue les identificateurs. Samurai est une technique qui considère que si un identificateur est utilisé dans une partie du code, il est probablement utilisé dans le même contexte que son code d’origine (*root*).

Toutefois, les approches mentionnées ci-dessus ont leurs limites. Premièrement, elles sont pour la plus part incapables d’associer des sous chaînes d’identifiants à des mots ou des termes; comme par exemple, des termes spécifiques à un domaine ou des mots de la langue anglais. Ces associations pourrait être utile pour comprendre le degré d'expressivité des termes décrit dans le code source par rapport aux artefacts de haut niveau qui leurs sont associés (De Lucia, Di Penta et al. 2006). Deuxièmement, ils sont incapables de prendre en compte les transformations de mots, tel que l'abréviation de *pointeur* en *pntr*.

Notre approche est inspirée de la technique de reconnaissance de la parole. La décomposition que nous proposons est basée sur une version modifiée de l'algorithme *Dynamic Time Warping* (DTW) proposé par Herman Ney pour la reconnaissance de la parole (Ney 1984) et sur une métrique qui est la distance de Levenshtein (Levenshtein 1966). Elle a été développée dans le but de traiter les limitations des approches existantes surtout celles qui consistent en la segmentation des identificateurs contenant des abréviations et à la gestion des transformations des mots du dictionnaire.

L'approche proposée a été appliquée à des identificateurs extraits de deux programmes différents : JHotDraw et Lynx. Les résultats obtenus ont été comparés aux oracles construits manuellement et également à ceux d'un algorithme de "splitting" basé sur la casse. Les résultats obtenus ont révélé que notre approche a un aspect non-déterministe relatif à l'établissent des méthodes de transformation appliquées et aux mots du dictionnaire et aux choix des mots du dictionnaire qui subissent ces transformations. Ils montrent que l'approche proposée à de meilleurs résultats que celle basée sur la casse. En particulier, pour le programme Lynx, le Camel Case Splitting n'a été en mesure de décomposer correctement qu'environ 18% des identificateurs, contrairement à notre approche qui a été capable de décomposer 93% des identificateurs. En ce qui concerne JHotDraw, le Camel Case splitter a montré une exactitude de 91% tandis que notre approche a assuré 96% de résultats corrects.

ABSTRACT

Maintenance is the most costly phase of software life cycle. In industry, the maintenance cost of a program is estimated at over 50% of its total life cycle costs (Sommerville 2000). Practical experience with large projects has shown that developers still face difficulties in maintaining their program (Pigoski 1996). Studies (Corbi 1989) have shown that over half of this maintenance is devoted to understanding the program itself. Program comprehension is therefore essential.

Program comprehension is a cognitive activity that relies on the construction of mental representations from software artifacts. Comprehension is more difficult for source code (Takang, Grubb et al. 1996). Several tools for understanding have been developed (Storey 2006); these tools range from simple visual inspection of the text (such as the explorers of code) to the dynamic analysis of program performance through program execution. While many efforts focus on automating the understanding of programs, a significant part of this work must still be done manually, such as: analyzing the source code, technical reports, and documentation.

A clear and concise documentation can help developers to inspect and understand their programs. Unfortunately, one of the major problems faced by developers, during maintenance, is that documentation is often outdated, or not available. Indeed, developers are often concerned about time and costs constraints, neglecting to update the documentation of different versions of their programs.

In the source code, identifiers and comments are key means to support developers during their understanding and maintenance activities. Indeed, identifiers are often composed of terms reflecting domain concepts. Usually, identifiers are built by considering a set of rules for choosing the character sequence. Some identifiers are composed of terms that are abbreviated and transformed of the words. Recognizing the terms in the identifiers is not an easy task when naming convention is not used.

In this thesis we will use a technique inspired from speech recognition, Dynamic Time Warping and meta-heuristic algorithms, to split identifiers into component terms. We propose a novel approach to identify terms composing identifiers that is organized in the following steps:

A dictionary of English words is built and will be our source of words. We take an identifier and look through the dictionary to find terms that are exactly contained in the identifier. For each word of a dictionary, we will compute the distance between word and the input identifier. For terms that exactly exist in both dictionary and identifier, the distance is zero and we obtain an exact splitting of the identifier and the process terminate successfully.

Other words of the dictionary with non-zero distance may indicate that the identifier is built from terms that are not exactly in the dictionary and some modification should be applied on the words. Some words of the dictionary have more characters than the terms in the identifier. Some transformations such as deleting all vowels or deleting some characters will be applied on the words of the dictionary. The modification of the words is applied in the context of a Hill Climbing search. For each new transformed word, we will calculate its distance to the input identifier via Dynamic Time Warping (DTW).

If the recently created word reduces the global minimum distance then we add that word to the current dictionary otherwise another transformation is applied. We will continue these steps until we reach to the distance of zero or the character number of the dictionary word become less than three or all the possible transformation have been applied. The identifier is split with words such that their distances are zero or have the lowest distance between other words of the dictionary.

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, we carried out a case study on two software systems, JHotDraw and Lynx. Results based on manually-built oracles indicate that the proposed approach outperforms a simple Camel Case splitter. In particular, for Lynx, the

Camel Case splitter was able to correctly split only about 18% of identifiers versus 93% with our approach, while on JHotDraw, the Camel Case splitter exhibited a correctness of 91%, while our approach ensured 96% of correct results. Our approach was also able to map abbreviations to dictionary words, in 44% and 70% of cases for JHotDraw and Lynx, respectively. We conclude that DTW, Hill Climbing and transformations are useful to split identifiers into words and propose future directions or research.

CONDENSÉ EN FRANÇAIS

La maintenance est la phase la plus coûteuse du développement logiciel. Dans l'industrie, le coût de maintenance d'un programme est estimé à plus de 50% de son coût de développement total (Sommerville 2000). L'expérience pratique, avec de grands projets, a montré que les développeurs font toujours face à des difficultés dans la maintenance de leurs programmes (Pigoski 1996). Des études (Corbi 1989) ont prouvé que plus de la moitié de la maintenance est consacrée à la compréhension du programme lui-même. La maîtrise de la compréhension des programmes s'avère donc indispensable.

La maintenance regroupe l'ensemble des activités effectuées pour modifier un logiciel après sa mise en opérations. Selon Swanson (Swanson 1976), il existe quatre types de maintenances : **la maintenance corrective qui** consiste à corriger des erreurs; **la maintenance perfective** qui cherche à ajouter et à modifier des fonctionnalités pour répondre aux nouveaux besoins de ses utilisateurs; **la maintenance adaptative** qui est l'adaptation du programme à un nouvel environnement d'opérations, tels qu'une nouvelle plateforme matériel ou à un nouveau système d'exploitation; et **la maintenance préventive** qui est l'effort pour prévenir les problèmes futurs. Toutes ces catégories de maintenance nécessitent des développeurs une compréhension approfondie du programme.

La compréhension de programmes est une activité cognitive qui repose sur la construction de représentations mentales à partir des artefacts logiciels. La compréhension est plus difficile pour les représentations orientées texte (par exemple, le code source) (Takang, Grubb et al. 1996). Les développeurs passent un temps considérable à lire et comprendre leurs programmes avant d'effectuer des changements. Plusieurs outils d'aide à la compréhension ont été développé (Storey 2006), ces outils vont de la simple inspection visuelle du texte (tels que, les explorateurs de code) à l'analyse dynamique de l'exécution du programme en passant par l'analyse statique du programme. Bien que beaucoup d'efforts portent sur l'automatisation de la compréhension de programmes, une partie significative de ce travail doit encore se faire manuellement, tels que : l'analyse du code source, la lecture de rapports techniques et de la documentation.

Une documentation claire et concise peut aider les développeurs à inspecter et à comprendre leurs programmes. Mais, l'un des problèmes majeurs que rencontrent les développeurs durant la maintenance est que la documentation est souvent obsolète ou tout simplement pas disponible. En industrie, les logiciels évoluent rapidement pour rester utiles. Les développeurs sont souvent préoccupés par des contraintes (de temps et de coûts) et négligent la mise à jour de la documentation au fur et à mesure que les versions de leurs programmes changent.

Depuis quelques années, certains auteurs (Takang, Grubb et al. 1996; Anquetil and Lethbridge 1998; Merlo, McAdam et al. 2003; Lawrie, Morrell et al. 2006; Lawrie, Morrell et al. 2007) reconnaissent que le code source d'un programme est une source d'information importante et fiable pour la compréhension du programme. Par conséquent, il est important de rendre le code source plus lisible, par exemple en insistant auprès des développeurs pour qu'ils ajoutent des commentaires dans leur code et respectent des règles syntaxiques et sémantiques en écrivant les identificateurs des concepts dans leurs programmes. En effet, les identificateurs sont souvent composés de termes reflétant les concepts du domaine; ils sont donc très utiles pour les activités de compréhension. Ils sont construits en respectant un ensemble de règles, pour choisir la séquence de caractères la plus représentative du concept. Mais certains identificateurs sont constitués de termes des mots qui sont abrégés ou transformés. La reconnaissance des termes composants les identificateurs n'est pas une tâche facile, surtout lorsque la convention de nommage n'est pas respectée.

Dans le cadre de **l'analyse des identificateurs**, Deißenböck et Pizka(Deißenbock and Pizka 2005) ont proposé deux règles de construction d' identificateurs concis et cohérents. Les identificateurs qui ne respectent pas ces deux règles augmentent la complexité de compréhension et ses coûts associés. Ces identificateurs sont souvent identifiés à l'aide des techniques de Deißenböck et Pizka qui sont basées sur un *mapping* entre le code source et la documentation. Dans un autre travail, Tonella et Caprile (Caprile and Tonella 1999) affirment que la maintenance d'un code source doit améliorer la lisibilité des identificateurs pour les rendre plus significatifs et compréhensibles. Ils ont aussi proposé une approche semi-automatique pour la restructuration de noms des identificateurs afin de les rendre auto-descriptive (Caprile and Tonella 2000).

Plusieurs travaux (Antoniol, Canfora et al. 2002; Marcus and Maletic 2003; Maletic, Antoniol et al. 2005) existent pour étudier **l'utilité des identificateurs dans la traçabilité entre la documentation et le code source**. De nombreux chercheurs (Jiang and Hassan 2006; Lawrie, Morrell et al. 2006; Fluri, Wursch et al. 2007) affirment que l'analyse des identificateurs et des commentaires peut aider à associer les concepts de haut niveau d'abstraction à ceux du code source, car les identificateurs encapsulent beaucoup d'informations et de connaissances des développeurs durant l'écriture du code. La traçabilité d'informations permet aux développeurs de comprendre les relations et les dépendances entre les divers artéfacts logiciels. Les identificateurs et les commentaires reflètent les concepts du domaine du logiciel.

Plusieurs chercheurs (Takang, Grubb et al. 1996; Anquetil and Lethbridge 1998; Merlo, McAdam et al. 2003; Lawrie, Morrell et al. 2006; Lawrie, Morrell et al. 2007) ont étudié l'impact des commentaires et des identificateurs sur la compréhension des programmes. Leurs études ont montré que la bonne utilisation des identificateurs et commentaires peut influencer significativement la compréhension d'un programme. Les chercheurs ont étudié également la qualité des commentaires et des identificateurs figurant dans le code source durant les tâches de compréhension de maintenance. Ils ont conclu que les identificateurs peuvent être très utiles si ils sont choisis efficacement pour refléter la sémantique et les rôles des entités nommées. La structure du code source et des commentaires aident à la compréhension des programmes et réduit ainsi les coûts de maintenance. Leurs études ont porté sur le rapport (ratio) entre le code source et les commentaires durant tout l'historique du projet, ainsi que les entités qui sont presque toutes commentées.

À notre connaissance deux familles d'approches existent pour décomposer les identificateurs : la plus simple considère l'utilisation du renommage et la présence des séparateurs explicites. La stratégie la plus complète est implémentée par l'outil Samurai (Enslen, Hill et al. 2009), elle se base sur le lexique et utilise les algorithmes gloutons pour identifier les mots qui constitue les identificateurs. Samurai est une technique qui considère que si un identificateur est utilisé dans une partie du code, il est probablement utilisé dans le même contexte que son code d'origine (*root*).

Toutefois, les approches mentionnées ci-dessus ont leurs limites. Premièrement, elles sont pour la plus part incapables d'associer des sous chaînes d'identifiants à des mots ou des termes; comme par exemple, des termes spécifiques à un domaine ou des mots de la langue anglais. Ces associations pourrait être utile pour comprendre le degré d'expressivité des termes décrit dans le code source par rapport aux artefacts de haut niveau qui leurs sont associés (De Lucia, Di Penta et al. 2006). Deuxièmement, ils sont incapables de prendre en compte les transformations de mots, tel que l'abréviation de *pointeur* en *pntr*.

Notre approche est inspirée de la technique de reconnaissance de la parole. La décomposition que nous proposons est basée sur une version modifiée de l'algorithme *Dynamic Time Warping* (DTW) proposé par Herman Ney pour la reconnaissance de la parole (Ney 1984) et sur une métrique qui est la distance de Levenshtein (Levenshtein 1966). Elle a été développée dans le but de traiter les limitations des approches existantes surtout celles qui consistent en la segmentation des identificateurs contenant des abréviations et à la gestion des transformations des mots du dictionnaire.

Notre approche décompose les identificateurs contenant n'importe quels mots transformés. Le processus de segmentation se fait en une seule itération, si les mots qui composent l'identificateur à traiter figurent dans le dictionnaire en entrée. Sinon, nous faisons appel aux transformations. Nous appliquons un ensemble de transformations successives sur les mots du dictionnaire en nous appuyant sur un algorithme de descente jusqu'à ce que nous puissions faire la correspondance entre les termes constituant l'identificateur et ces mots transformés. La correspondance exacte est atteinte si la distance entre un mot du dictionnaire et le terme traité est nulle. Nous avons développé cinq transformations ; nous citons à titre d'exemple celle portant sur la suppression de toutes les voyelles contenues au niveau d'un mot choisi du dictionnaire, celle portant sur la suppression des *m* derniers caractères et puis la suppression d'un caractère aléatoire.

Un mot transformé est considéré comme étant un nouveau mot du dictionnaire courant si et seulement si il réduit la distance. Sinon et si il reste encore des transformations à appliquer et la longueur du mot transformé est inférieure ou égale à trois caractères nous appliquons le même processus de segmentation. Sinon, si on atteint le nombre maximal d'itérations et que la distance est non nulle, le processus se termine sur un échec.

La nouvelle approche s'inspire du fait que les développeurs forment les identificateurs en appliquant un nombre de règles et des transformations sur les mots; ces dernières sont traitées à l'aide d'un algorithme de descente de Hill Climbing (Michalewicz and Fogel 2004) après sélection des individus (mots du dictionnaire) à l'aide d'un operateur de sélection qui est dans notre cas une Roulette biaisée dont l'intérêt apparait au niveau des algorithme génétique et évolutionnistes pour sélectionner les individus appropriés à partir d'une population. Il est à noter que le Hill Climbing est un algorithme de recherche locale qui par d'une solution initiale puis tente d'améliore cette solution dans un espace de recherche. À chaque itération, le coût de la solution trouvée est comparée à celle trouvée. Si elle est meilleure, la nouvelle solution remplace la solution actuelle. De même, le voisinage de la nouvelle solution est étudié. Si une meilleure solution est trouvée, nous opterons pour la nouvelle solution, et ainsi de suite.

Notre approche utilise un dictionnaire et des identificateurs extraits du code du logiciel ; le dictionnaire est composé des mots extraits d'un glossaire disponible sur

l'Internet, des mots anglais, et des termes les plus fréquents dans le code source du logiciel à étudier. La segmentation est effectuée par l'algorithme DTW qui a été conçu dans le but d'étudier l'alignement entre deux séries temporelles. Il a été adapté dans notre cas pour trouver le chemin de coût minimal entre l'identificateur et les mots du dictionnaire. En effet, le calcul de la distance se fait en initialisant pour chaque mot du dictionnaire une forme de référence qui est une matrice où l'axe des abscisses représente le mot à décomposer et l'axe des ordonnées représente le mot du dictionnaire. Chaque matrice est calculée en faisant une comparaison entre les caractères correspondants appartenant à l'identificateur et au mot du dictionnaire. La dernière cellule de la matrice de coût représente la distance globale minimale entre les identificateur et les mots du dictionnaire. Cette dernière est basée sur une distance de Levenshtein qui refléte le nombre de suppressions, d'insertions ou de substitutions nécessaires pour transformer une chaîne de caractères en une autre chaîne de caractères.

L'approche proposée a été appliquée à des identificateurs extraits de deux programmes différents : JHotDraw et Lynx. Les résultats obtenus ont été comparés aux oracles construits manuellement et également à ceux d'un algorithme de "splitting" basé sur la casse. Les résultats obtenus sont généralement encourageants. L'analyse de nos résultats a été faite suite à une étude empirique dans l'objectif de répondes aux questions de recherche suivantes : *Quel est le pourcentage d'identificateurs correctement* décomposes *par l'approche proposée ? Comment l'approche proposée performe par rapport au Camel splitter ? Quel est le pourcentage d'identificateurs contenant des abréviations que notre approche est en mesure de décomposer ?*

Les questions de recherche ci-dessus visent à comprendre si l'approche proposée contribue à décomposer les identificateurs. Ainsi, on suppose implicitement qu'étant donné un identificateur, il existe une décomposition exacte de ce dernier en des termes et des mots qui, éventuellement après des transformations et une fois concaténés, composent l'identificateur.

Les résultats obtenus ont révélé que notre approche a un aspect non-déterministe relatif à l'établissent des méthodes de transformation appliquées et aux mots du dictionnaire et aux choix des mots du dictionnaire qui subissent ces transformation. Ils montrent que l'approche proposée a de meilleurs résultats que celle basée sur la casse. En particulier, pour le programme Lynx, le Camel Case Splitting n'a été en mesure de décomposer correctement qu'environ 18% des identificateurs, contrairement à notre approche qui a été capable de décomposer 93% des identificateurs. En ce qui concerne JHotDraw, le Camel Case splitter a montré une exactitude de 91% tandis que notre approche a assuré 96% de résultats corrects.

Nos futurs travaux de recherche portent sur l'évaluation de notre approche sur d'autres programmes et sur l'introduction de nouvelles heuristiques améliorées pour la sélection des mots issus du dictionnaire ainsi que le choix des transformations à appliquer. Le fait de combiner notre approche de recherche avec celle de Enslen et al. (Enslen, Hill et al. 2009) en limitant la recherche aux mots utilisés dans la même méthode, la même classe ou le même paquetage sera aussi considérée comme direction de recherche.

Pour conclure, nous pouvons dire que nos travaux de recherche ont été publiés à la 14 Conférence européenne sur la rétro-conception et la maintenance (CSMR) en mars 2010. L'article publié est intitulé "Recognizing Words from Source Code Identifiers using Speech Recognition Techniques". Il a été rédige par: Nioosha Madani, Latifa Guerrouj, Massimiliano Di Penta, Yann-Gaël Guéhéneuc et Giuliano Antoniol. Cet article a eu le prix du meilleur article de la conférence.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF INITIALS AND ABBREVIATIONS

BP	Back Points
D	Distance
DP	Dynamic Programming
DTW	Dynamic Time Warping
HC	Hill Climbing
IR	Information Retrieval
LSI	Latent Semantic Indexing
LD	Levenshtein Edit Distance
OR	Odds Ratio
RQ	Research Question
SBSE	Search Based Software Engineering
SA	Simulated Annealing

CHAPTER 1 INTRODUCTION

One of the problems that developers face when understanding and maintaining a software system is that, very often, documentation is scarce, outdated, or simply not available. This problem is not limited to open source projects but is also true in industry: as systems evolve, documentation is not updated due to time pressure and the need to reduce costs. Consequently, the only up-to-date source of information is the source code and therefore identifiers and comments are key means to support developers during their understanding and maintenance activities. The paramount role of program identifiers in program understanding, traceability recovery, feature and concept location tasks motivate the large body of relevant work in this area.

In the following, we will refer to any substring in a compound identifier as a *term*, while an entry in a dictionary (*e.g.*, the English dictionary) will be referred to as a *word*. A term may or may not be a dictionary word. A term carries a single meaning in the context where it is used; while a word may have multiple meanings (upper ontologies like WordNet[1] associate multiple meanings to words).

Indeed, identifiers are often composed of terms reflecting domain concepts (Lawrie, Morrell et al. 2006), referred to as *"hard words"*. Hard words are usually concatenated to form compound identifiers, using the Camel Case naming convention, *e.g.*, *drawRectangle*, or underscore, *e.g.*, *draw_rectangle*. Sometimes, no Camel Case convention or other separator is used. Also, acronyms and abbreviations may be part of any identifier, *e.g.*, *drawrect* or *pntrapplicationgid*. The component words *draw, application*, the abbreviations *rect, pntr*, and the acronym *gid* (*i.e.*, group identifier) are referred to as *"soft-words"(Lawrie, Feild et al. 2006).*

This thesis proposes a novel approach to segment 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 (Ney 1984) (*i.e.*, for recognizing sequences of words in a speech signal) and on the Levenshtein string edit-distance (Levenshtein 1966). The approach assumes that there is a limited set of (implicit

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

and–or explicit) rules applied by developers to create identifiers. It uses words transformation rules, plus a hill climbing algorithm (Michalewicz and Fogel 2004) to deal with word abbreviation and transformation.

The main contributions of this thesis are the following:

1) A new approach to split identifiers, inspired from speech recognition. The approach overcomes limitations of previous approaches and can split identifiers composed of transformed words, regardless of the kind of separators;

2) Evidence that the approach can be used to map transformed words composing identifiers to dictionary words and, therefore, to build a thesaurus of the identifiers;

3) Results of applying our approach on two software systems belonging to different domains, JHotDraw (written in Java) and Lynx (written in C). Results based on manually-built oracles indicate that the approach can correctly split up to 96% of the identifiers and can even be used to identify errors in the oracle.

This thesis is organized as follows: The next chapter summarizes relevant works and relates our work to the existing literature. Analyzing identifiers, splitting identifiers, recovering traceability links between source code and documentation, and also measuring the conceptual cohesion and coupling are discussed in this chapter. In Chapter 3, we describe the novel approach to split identifiers, also reporting a primer on Ney's connected-words recognition algorithm (Ney 1984). Chapter 4 reports an empirical study aimed at evaluating the proposed approach. We report results from our experimental study carried out 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. Finally, Chapter 5 concludes the thesis and outlines future work.

CHAPTER 2 RELATED WORK

Program comprehension is the activity of building mental abstraction from software artifacts. Program comprehension is an essential and central part of reuse, maintenance, and reverse engineering, and many other activities in software engineering. Often, a large fraction of maintenance time is spent reading code to understand what functionality of the program it implements.

Comprehension of the code is defined by Biggerstaff et al. (Biggerstaff, Mitbander et al. 1993) as follows: “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”.

Developers and maintainers face the challenging task of understanding of the system when the code is not sufficiently documented. Understanding of an existing system is a time consuming activity especially when its documents are out-dated or do not exist. Then software maintainers must study the source code of the software systems (Sulaiman S. 2002). Indeed the only up-to-date source of information is the source code. In the source code, identifiers and comments are key means to support developers during their understanding and maintenance activities.

2.1 Identifier Analysis

Developers use two sources of domain information such as identifier names and comments for understanding the programs. Identifiers constructed by developers may contain useful information that is often the starting point for the program comprehension activities.

Well-formed variable names, as described by Deißenböck and Pizka, can improve code quality (Deißenbock and Pizka 2005). Deißenböck and Pizka highlighted that proper identifiers improve software quality. These authors believe that it is essential that the identifiers and comments contain the concept that they represent. They introduced two rules for creating well-formed identifiers: conciseness and consistency. In order to verify the conciseness and consistency of identifiers, they provided a mapping from identifiers to the domain of concepts. Identifiers that are not concise or consistent cause a complexity in comprehension.

The motivation for their work is the observation that “lousy naming in one place spoils comprehension in numerous other places”, while the basis for their work is found in the quote “research on the cognitive processes of language and text understanding shows that it is the semantics inherent to words that determine the comprehension process”(Deißenbock and Pizka 2005).

More in detail, Deißenböck and Pizka define three rules, one for *concise* and two for *consistent* identifier names. The authors name an identifier a *concise identifier* if its semantics exactly match the semantics of the concept it is used to represent. For example, *drawInputRectangle* concisely represents the concept of drawing an input rectangle. A related notion, *correctness*, allows an identifier to represent a more general concept. For example, *drawRectangle* correctly but not concisely represents the concept of drawing a rectangle but not the input rectangle. While the identifier of *foo* neither correctly nor concisely represents the concept.

Homonyms and synonyms are two aspects which cause inconsistencies in identifiers. When two or more words spelled and pronounced alike but different in meaning, they are a homonymous (*e.g.*, marc, mark, marquee or root, route). A synonym is one of two or more words or expressions that have the same or nearly the same meaning in some or all senses (*e.g.*, search, seek, look for). Identifiers that fail to be concise or consistent increase the comprehension complexity and its associated costs. Conciseness and consistency can be identified via Deißenböck and Pizka's techniques that provide a mapping from identifiers to concepts.

According to Tonella and Caprile (Caprile and Tonella 1999; Caprile and Tonella 2000), identifiers are one of the most important sources of information about system concepts. Tonella and Caprile analyze the lexical, syntactical, and semantical structure of function identifiers by means of a segmentation technique, a regular language, and a conceptual classification. Caprile and Tonella then investigate the structure of function identifiers in C programs. They build a dictionary of identifier fragments and then propose a grammar that describes and shows the roles of the fragments. They use concept analysis to perform a classification of the words in the dictionary. The analysis involves breaking identifiers into well-separated words (*i.e.*, hard words). The restructuring involves two steps. First, a lexicon is standardized by using only standard terms as composing words within identifiers. Second, the arrangement of standard terms into a sequence must respect a grammar that conveys additional information. Tonella and Caprile believe that these types of analyses are useful in the context of reverse engineering of existing systems. In fact, "renovation of old code may include making identifiers more meaningful and understandable and its comprehension may exploit the information extracted" (Caprile and Tonella 1999).

In another work, Caprile and Tonella proposed a semiautomatic technique for the restructuring of identifiers with the goal of improving their meaningfulness and making identifiers self descriptive (Caprile and Tonella 2000).

Also, methods to refactor identifiers were proposed in (Demeyer, Ducasse et al. 2000). The authors proposed heuristics for detecting refactorings by calculating metrics over successive versions of a system. They validated their approach with three case studies for which multiple versions are available with the goal of investigating how information of identifying refactoring helps in program comprehension.

2.2 Traceability Links between Source Code and Documentation

Traceability information helps software engineers to understand the relationships and dependencies among various software artifacts. Several papers proposed in the literature to recover traceability links between source code and documentation. In various software activities, such as program comprehension, software maintenance, and software verification and validation, traceability between the free-text documentation and its source code is an essential information. Researchers have studied the usefulness of identifiers to recover traceability links (Antoniol, Canfora et al. 2002; Marcus and Maletic 2003; Maletic, Antoniol et al. 2005). 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.

Antoniol et al. (Antoniol, Canfora et al. 2002) used an Information Retrieval (IR) method to recover traceability links between free-text documentation and source code. They applied their approach in tracing C++ and Java source code units, manual pages, and functional requirements. In their method, identifiers in the source code are assumed to be meaningful names that are derived from the application, *i.e.,* identifiers share the semantics of the problem domain. They proposed 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 word. With this method, Antoniol et al. (Antoniol, Canfora et al. 2002) computed the similarity between queries and documents and returned a ranked list of documents for each source code component.

Marcus and Maletic (Marcus and Maletic 2003) proposed to use IR methods to support software engineering tasks and to recover source code to documentation links. They presented a method to recovery traceability links between documentation and source code, using an information retrieval method, namely Latent Semantic Indexing (LSI).

2.3 Conceptual Cohesion and Coupling

Identifiers and comments reflect concepts from the domain of the software system. In object-oriented systems, classes contain these identifiers and comments.
Analyzing the identifiers and comments in classes can be used to measure the cohesion in a system. Systems that contain high cohesion and low coupling among the classes facilitate comprehension, testing, reusability,and maintainability.

Poshyvanyk and Marcus (Poshyvanyk and Marcus 2006) worked on a set of operational measures for the conceptual coupling of classes in object-oriented systems, which formulates and captures new dimensions of coupling, named conceptual coupling, based on the semantic information obtained from the source code, encoded in identifiers and comments. They measured the strength of conceptual similarities among methods of different classes. They used information retrieval techniques to model and analyze the semantic information embedded through comments and identifiers. Their results show that the conceptual coupling captures new dimensions of coupling, which are not captured by existing coupling measures.

Marcus et al. (Marcus, Poshyvanyk et al. 2008) propose a new measure for the cohesion of classes in Object-Oriented software systems based on the analysis of the unstructured information embedded in the source code, such as comments and identifiers. The new measure named the Conceptual Cohesion of Classes (C3) captures the conceptual aspects of class cohesion. It measures how strongly the methods of a class relate to each other conceptually. They used Latent Semantic Indexing to extract this information for cohesion measurement.

2.4 Effects of Comments and Identifier Names on Program Comprehensibility

Many researchers investigated the effects of comments and identifiers on program comprehension (Takang, Grubb et al. 1996; Anquetil and Lethbridge 1998; Merlo, McAdam et al. 2003; Lawrie, Morrell et al. 2006; Lawrie, Morrell et al. 2007). They showed that effective use of comments and identifiers can significantly increase program comprehension.

Takang et al. (Takang, Grubb et al. 1996) investigated the combined impact of comments and identifiers on program comprehension using of controlled experimentation based on existing program comprehension theories. They note that " The quality of identifier names is an issue that merits closer consideration and exploration in its own right" (Takang, Grubb et al. 1996). Takang and his colleagues conducted an experiment using 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.

In this experiment, the authors noted that: "the impact of identifier names on comprehension of small or familiar programs might not be significant enough to be reflected in the test scores."

Lawrie et al. (Lawrie, Morrell et al. 2007) 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 help developers' recall when compared to less informative identifiers.

They investigated if "the initials of a concept name provide enough information to represent the concept? If not, and a longer identifier is needed, is an abbreviation satisfactory or does the concept need to be captured in an identifier that includes full words?“ Their results showed that full-word identifiers lead to the best program comprehension.

Lawrie et al. (Lawrie, Morrell et al. 2006) 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 experienced 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. They noted 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. It also shows that in many cases abbreviations are as useful as the full word identifiers”.

Lawrie et al. (Lawrie, Feild et al. 2006) studied the restriction and extension of Deißenböck and Pizka’s rules that is computable without a mapping from names to concepts. They define a syntax-based conciseness and consistency that does not require an expert-constructed mapping from identifiers to concepts. They performed two case studies. First, they considered all conciseness and consistency failures from two small programs; they then compared the output of the tool to a human oracle. Second, they considered a sampling of the conciseness and consistency failures from the larger program eMule, a 170 KLoC C++ program. Finally, a longitudinal study addressed the question “does evolution introduce conciseness and consistency failures?” They found that full word identifiers lead to the best comprehension; although, there is no statistical difference between full words and abbreviations.

Researchers have also studied the quality of source code comments and the use of comments and identifiers by developers during understanding and maintenance activities (Jiang and Hassan 2006; Lawrie, Morrell et al. 2006; Fluri, Wursch et al. 2007). 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. (Fluri, Wursch et al. 2007) applied an approach to map code and comments to study their co-evolution over multiple versions. They investigated whether source code and associated comments are really changed together along the evolutionary history of three open source systems, ArgoUML, Azureus, Eclipse, and 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 find that it does not stay at a stable value.

Jiang and Hassan (Jiang and Hassan 2006) 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 found 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 consistant throughout the development history". They mentioned 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 (Sim and Holt 1998).

2.5 Identifier Splitting

The first step in analyzing words from identifiers requires splitting identifiers into their constituent words. Two families of approaches exist to split compound identifiers: the simplest one assumes the use of the Camel Case naming convention or the presence of an explicit separator. A more complex strategy is implemented by the Samurai tool and relies on a lexicon and uses greedy algorithms to identify component words (Enslen, Hill et al. 2009). 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, *FirstYearSalary* or *numberOfDays* use camel case rules.

The advantage of Camel Case is that, in any computer system where the letters in a name must be contiguous (no spaces), a more meaningful name can be created using a descriptive sequence of words without violating the programming language syntax and grammar. Java programmers often use Camel Case, where words are identified by uppercase letters or non-alphabetic characters. In cases that multi-word identifiers use the coding conventions such as camel casing and non-alphabetic characters; splitting of the identifier into their constituent word is possible via software analysis tools that use natural language information rely on coding conventions. However, there are some cases where existing coding conventions break down (e.g. serialversionuid, ASTVisitor, GPSstate).

For any identifier, there are four possible cases to consider:

1- Character in place *i* is lowercase and character in place *j* is uppercase (e.g., getString)
2- Character of *i* is upper case and character of *j* is lower case (e.g. GPSState)
3- Both characters of *i* and *j* are lower case (e.g., newlen)
4- Both characters of *i* and *j* are upper case (e.g. FILLRECT)

Case 1 is the straightforward Camel Case without abbreviations. Case 2, follows Camel Case but, in some cases, it can cause an incorrect splitting (e.g., get MA Xstring, GP Sstate). Samurai (Enslen, Hill et al. 2009), refer to cases 3 and 4 of the previous itemize , as the same-case token splitting problem. In these two cases the programmer has not used any camel case or naming convention, thus it is not easy to find out special rules to split these kinds of identifiers.

Samurai (Enslen, Hill et al. 2009) is a technique and a tool that assumes that an identifier is composed of words used (alone) in some other parts of the system. It therefore uses words and word frequencies, mined from the source code, to determine likely splitting of identifiers. Its hypothesis is that "the strings composing multi-word tokens in a given program are most likely used elsewhere in the same program or in other programs. The words could have been used alone or as part of another token". Thus Samurai uses string frequencies in the system to determine splits in the identifier. Samurai mines string frequencies information from the source code and builds two tables of frequencies. The local table consists of the numbers of occurrence of each unique sting in the current source code and a global frequency table is built from the sets of strings extracted from a large corpus of systems. These two tables of frequencies help the algorithm of Samurai to split multi-word identifiers involving Camel Case (*e.g.,* getWSstring) and same case multi-words (*e.g.,* serialversionuid).

In its first step, for each identifier, Samurai executes the mixedCaseSplit algorithm and splits the token by special character and digits and via splitting of the lowercase to upper case characters. In this algorithm each mixed-case alphabetic substrings is tested to choose from straightforward camel case splitting before the last upper case letter (*e.g.,* "AST Visitor", "GP Sstate") or the alternate split between the last upper case letter and the first lower case letter (*e.g.,* "ASTV isitor", "GPS state"). This decision is determined via comparing the frequency of the right hand side of the split in the program under analysis and in a more global scope of a large set of programs.

The output of the mixedCaseSplit algorithm can be either (1) all lower case or (2) all upper case or (3) a single upper case followed by all lower case letters. In this step, if the identifier uses the coding conventions, such as Camel Casing or non-alphabetic characters it can be easily split. The output of the previous algorithm is then processed by the sameCaseSplit algorithm. In the sameCaseSplit algorithm each possible split are examined by comparing the score of left and right of the split point. The substrings returned from sameCaseSplit are linked together with space to construct the final split terms.

However, Samurai has the following limitations. First, it is not always possible 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 (De Lucia, Di Penta et al. 2006). Second, they do not deal with word transformations, *e.g.*, abbreviation of *pointer* into *pntr*.

Overall, the above previous works highlights the importance of carefully choosing identifiers for source code comprehension and maintainability. In this context, the application of our approach would be to map terms in source code identifiers to domain dictionary words to better assess the quality of these identifiers. Commonalities can be found with the work of Enslen et al.*(Enslen, Hill et al. 2009)* and the approach proposed in this thesis.Commonality is listed to the fact that we share with them the goal of automatically splitting identifiers into component words. Our approach is different. We do not assume the presence of Camel Casing nor of a set of known prefixes or suffixes. In addition, our approach automatically generates a thesaurus of abbreviations via transformations attempting to mimic the cognitive processes of developers when composing identifiers with abbreviated forms.

CHAPTER 3 OUR IDENTIFIER SPLITTING APPROACH

3.1 Definitions

For sake of completeness we report in the following the basic definitions already defined in the Introduction Chapter and needed to understand splitting of the identifiers We will refer to any substring in a compound identifier as a *term*, while an entry in a dictionary (*e.g.*, the English dictionary) will be referred to as a *word*. A term may or may not be a dictionary word. A term carries a single meaning in the context where it is used; while a word may have multiple meanings (upper ontologies like WordNet associate multiple meanings to words). Stemming from Deißenböck and Pizka (Deißenbock and Pizka 2005) observation on the relevance of words and terms in identifiers to drive program comprehension attempted to segment identifiers by splitting them into component terms and words.

Indeed, identifiers are often composed of terms reflecting domain concepts (Lawrie, Morrell et al. 2006), referred to as *"hard words"*. Hard words are usually concatenated to form compound identifiers, using the Camel Case naming convention, *e.g.*, *drawRectangle*, or underscore, *e.g.*, *draw _rectangle*. 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.*, *drawrect* or *pntrapplicationgid*. The component words *draw, application*, the abbreviations *rect, pntr*, and the acronym *gid* (*i.e.*, group identifier) are referred to as *"soft-words"* (Lawrie, Feild et al. 2006).

3.2 Goal

Our approach intends to segment identifiers into composing words and terms. The application of our approach would be to map terms in source code identifiers to domain dictionary words to better assess the quality of these identifiers. Our approach is inspired from speech recognition and overcomes the limitations of previous approaches, *i.e.* it is 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 (De Lucia, Di Penta et al. 2006). It also deals with word transformations, *e.g.*, abbreviation of *pointer* into *pntr*. Our approach splits identifiers composed of transformed words, regardless of the kind of separators.

3.3 Steps

Our approach is based on a modified version of the Dynamic Time Warping (DTW) algorithm proposed by Ney for connected speech recognition (Ney 1984) (*i.e.*, for recognizing sequences of words in a speech signal) and on the Levenshtein string edit-distance (Levenshtein 1966). The approach assumes that there is a limited set of (implicit and–or explicit) rules applied by developers to create identifiers. It uses words transformation rules, plus a hill climbing algorithm (Michalewicz and Fogel 2004) to deal with word abbreviation and transformation.

Our identifier splitting algorithm works as follows, as shown in Fig.1:
1) Based on the current dictionary, we (i) split the identifier using DTW that will be explained in Section 3.4.,(ii) from the previous score calculation, each word captures the score of comparison and then from these scores we 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 words with non-zero distance obtained at step (1), we randomly select one word having the 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) 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 than three characters, we go to step (a); else,
3) If the global distance is non-zero and the iteration limit was not reached, then, we go to step (1), otherwise we exit with failure.

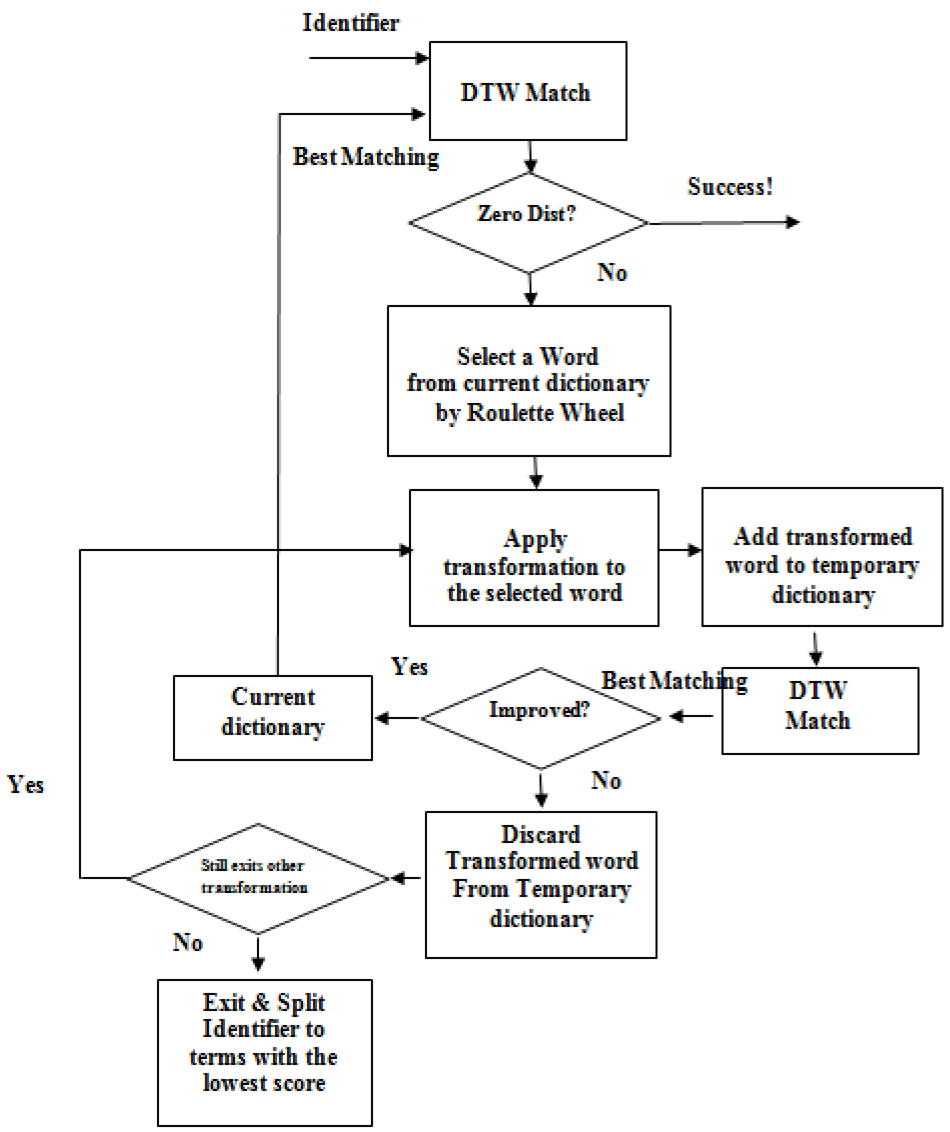

Identifier
DTW Match
Best Matching
Zero Dist?
Success!
No
Select a Word from current dictionary by Roulette Wheel
Apply transformation to the selected word
Add transformed word to temporary dictionary
Yes
Best Matching
DTW Match
Current dictionary
Improved?
Yes
No
Discard Transformed word From Temporary dictionary
Still exits other transformation
No
Exit & Split Identifier to terms with the lowest score

Figure 1 - Approach Steps

Each transformed word will be added to the dictionary if and only if it reduces the global distance. Briefly, a hill climbing algorithm 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 algorithm terminates when there is no move to nearby solutions improving the fitness. Different from traditional hill climbing algorithms, our algorithm attempts to explore as much as possible of neighboring solutions by performing word transformations. Different neighbors can be explored depending on the order of transformations.

3.4 Dynamic Programming Algorithm

Dynamic Programming (DP) is a method for efficiently solving complex problems by breaking them into simpler steps. DP is a powerful technique for solving problems where we need to find the best solutions one after another. A dynamic program is an algorithmic technique that is usually based on a recurrent formula and some starting states; a sub-solution of the problem is constructed from previously found ones following a divide and conquers strategy. If sub-problems can be nested recursively inside larger problems, then dynamic programming methods are applicable, then there is a relation between the solution value of the larger problem and the values of the sub problems. In DP, first of all we need to find a *state* for which an optimal solution is found and with the help of which we can find the optimal solution for the next *state*. A s*tate* is a way to describe a situation, sub-solution for the problem. Dynamic programming works either top-down or bottom-up:

In the top-down approach, if the solution to any problem can be formulated recursively using the solution to its sub-problems then we will save the solutions to the sub-problems in a table. When we want to solve a new sub-stproblem first , we check in this table if we have already solved this problem. If there exists a solution, then we use it directly, otherwise we solve the sub-problem and add its solution to the table. In the bottom-up approach, each time that we solve sub-problems we must use their solutions to build-on and reach to a solution to bigger sub-problems. This approach is also usually done in a tabular form by iteratively generating solutions to bigger and bigger sub-problems by using the solutions to small sub-problems.

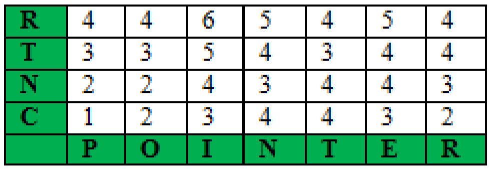
3.4.1 DTW Definition

Dynamic time-warping (DTW) studies multi-dimensional time series of different length and evaluates the similarity of two time series. The distance between two series after warping is calculated. The distance measures how well the features of a new unknown sequence match those of reference template. DTW uses a dynamic technique to compare point by point two series by building a matrix. It will build this matrix staring from bottom-left corner which is 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 top-right cell contains the distance between the two strings that has the shortest path in this matrix (Lachlan 2007).

Our approach is based on a modified version of Dynamic Time Warping. The input identifier is compared with all English word of the dictionary, character by character. Each word of the dictionary is stored in a template and this template is compared with the input identifier to find the closest match. Then the distance is found by minimizing a cost that is defined by the string edit distances between all matches. The minimum-distance of the top best match decides the identifier split.

3.4.2 DTW Template

For each word of the dictionary, we should initiate a template. In each template, the identifier is compared with one of the words from the dictionary. Let us consider words in the dictionary along the y-axis of the template. Both strings (identifier and word dictionary) , see Fig. 2, start on the bottom left on the template. In each cell, we must calculate the distance by comparing the two characters.



R	4	4	6	5	4	5	4
T	3	3	5	4	3	4	4
N	2	2	4	3	4	4	3
C	1	2	3	4	4	3	2
	P	**O**	**I**	**N**	**T**	**E**	**R**

Figure 2 - Example of DTW Template

Let us consider that we have two strings of X and Y as:

$X=x_1, x_2, x_3,.., x_n$

$Y=y_1, y_2, y_3,.., y_m$

The sequence of X and Y will be arranged in a matrix of n-by-m, where each cell, (i,j) , correspond to the difference of character x_i and y_j. There is a warping path, W, that maps the elements of X and Y, in a way that the distance between these two sequences is minimized.

After comparison of the two sequences we can find a path through the template that minimizes the total distance between the sequences. The main goal for computing the overall distance measure is to find all possible paths through the template. The number of possible paths through the template can be more than one. We must select the best path among the existing paths. We can define the dynamic time warping problem as a minimization over the warping path;

$$DTW(X,Y) = \min[\sum_{k=1}^{p} | x_i - y_j | (w_k)]$$

3.4. 3 DTW Optimizations

As the number of alignment path is exponential in the pattern length, there is a need to reduce complexity with a suitable strategy. The approach most often use is to impose constraints to the possible alignment function. This is to say: given a cell (i,j), only a limited number of other cells [(i-1,j),(i,j-1),(i-1,j-1)] can originate the path through (i,j).

The major optimizations to the DTW algorithm stems from observations on the nature of good paths through the grid. These paths characteristic outlined in Sakoe and Chiba and can be summarized as (Sakoe and Chiba 1978):

- The path will not turn back on itself, both *i* and *j* indexes either stay the same or increase, they never decrease.

- The path advances one step at a time. Both *i* and *j* can only increase by 1 at each step along the path.
- The path starts at the bottom left and ends at the top right.

3.4.4 Weighting Function

A weight function is a mathematical formula used to give some elements in a same sets more “weight” to influence some results than other elements. The weighting function is used to normalize the path.

There exist two families of weighting functions. With a symmetric function, we combine direction of *i* and *j* while the asymmetric function just consider direction of *i*. If the path move in a diagonal step then *i* and *j* will increase by 1 and we have $w_{ij}= w_{ji}$; while in the asymmetric, we have $w_{ij}\neq w_{ji}$; this is a kind of normalization for the path overall distance measures (Cassidy 2002).

Then, for any intermediate point the cost of *D(i; j)* is computed as:

$$\begin{aligned}
&c(i;j) = d(xi;yj)\\
&D(i;j) = min[w_1 D(i-1;j) + c(i;j)\,; && \text{//insertion}\\
&\qquad\qquad w_2 D(i;j-1) + c(i;j)\,; && \text{//deletion}\\
&\qquad\qquad D(i-1;j-1) + w_3 c(i;j)] && \text{//match}
\end{aligned}$$

Equation 1- Distance calculation

Where *D(i; j)* is the global distance and the *d(i,j)* is the distance found in the current cell as a local distance (*i.e.*, comparison of characters). That is, the sum of the distance between current characters and the minimum of the distance of the neighboring points. Our implementation uses a symmetric function where $w_1=w_2$.

Eq.1 computes the current value of the distance based on the previous values thus it imposes continuity constraints. Each value of the weights w_1, w_2, w_3 are problem-dependent and most of the times, w_1 is chosen equal to w_2 and the value of w_3 is often chosen to be twice of the value of w_1 and w_2. In our computation, we choose $w1 = w2 = 1$ and $w3 = 2$ as in the classic Levenshtein string edit distance (Alshraideh and Bottaci 2006).

3.4.5 Computation Process

The computation starts from the bottom-left cell of the matrix and we compute the distance D(i,j) based on the $d(x_i,y_j)$ for each cell in the distance matrix. In this part we will compare the characters of x_i and y_j and finding the local path of minimum cost between (i-1,j),(i,j),(i,j-1). This comparison can be done by columns or rows. After all cells of are computed then value of D(N,M) contains the minimum distance between $x_1; x_2; \ldots ; x_N$ and $y_1; y_2; \ldots; y_M$.

Backtracking from $(N;M)$ down to $(0; 0)$ recovers the warping path corresponding to the optimal alignment of $x_1; x_2; \ldots ; x_N$ and $y_1; y_2; \ldots; y_M$.

3.4.6 DTW for Connected Word Recognition

Identifier splitting is performed via an adaptation of the connected speech recognition algorithm proposed by Ney (Ney 1984) that, in turns, extends to connected words the isolated word DTW (Sakoe and Chiba 1978) algorithm. DTW was conceived for time series alignment and was widely applied in early speech recognition applications in the 70s and 80s. Herman Ney (Ney 1984) presents a simple approach to the pattern matching problem for connected word recognition which is based on parametrizing the time warping path with single index and by using path constraint both in the word boundaries and word interior.

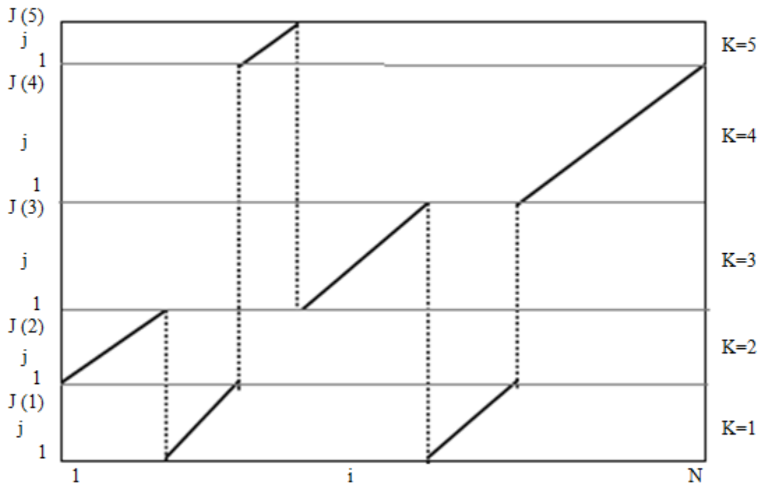


Figure 3 -Connected Word Recognition Problem (Ney 1984)

In Ney's approach, there is an input that is composed of individual words or test pattern that contains i=1,…,N time frames. Also there exist words templates which are distinguished by the index of k=1,…,k. The time frames of the template *k* are denoted as *j* = 1,. . . , *J(k),* where *J(k)* is the length of the template *k.*

The basic idea is illustrated in Fig.3. The time frames *i* of the test pattern and the time frames *j* of each template *k* define a set of grid points *(i, j, k).* There is a local distance measure of *d(i, j , k)* which indicate the dissimilarity between the corresponding acoustic events. The goal of his approach is to find the best path (i.e. time warping path) through these grid points of (i,j,k) that shows the best match between the test pattern and the unknown sequence of templates. Different constraints are applied in finding the best path. Path with minimum global distance, i.e., the sum over the local distances along a given path is the best one to select.

Two types of transition rules are applied due to concatenation of single word templates to a "super" reference pattern: (1) Within-template transition rules, *i.e.,* transitions rules in the template interior; (2) Between-template transitions rules, *i.e.,* transition rules at the template boundaries.

Herman Ney illustrates these two types of transition rules in Fig.4:

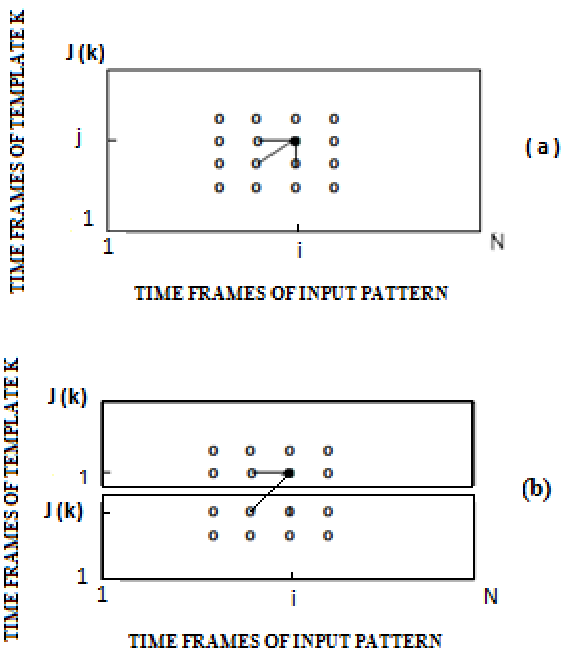

TIME FRAMES OF TEMPLATE K
J (k)
j
1
1
i
N
TIME FRAMES OF INPUT PATTERN
(a)
TIME FRAMES OF TEMPLATE K
J (k)
1
J (k)
1
1
i
N
TIME FRAMES OF INPUT PATTERN
(b)

Figure 4 – (a) Within-template Transition Rules.

(b) Illustration of Between-template (Ney 1984)

As shown in Fig.4(a), a within- template transition rule is:

if w(l) = (i,j, k), j > 1, then

$$w(l-1) \in \{(i-l, j, k), (i-1, j-l, k), (i, j-l, k)\}$$

i.e., it explains that the point (i,j,k) can be reached from one of the points of (i-1,j,k), (i,j-1,k), (i-1,j-1,k).

Also as shown in Fig.4(b), a between-template transition rules is:

if w(l) = (i, 1, k), then

$$w(l-1) \in \{(i-1, 1, k); (i-1, J(k^*), k^*): k^* = 1, \ldots, K\}.$$

Where point (i,1, k) is the beginning frame of the template *k* that it can reach from the ending frame of any template *k** including *k* itself. Ney algorithm finds the best path through these points which is the minimum distance along any path to grid point (i, j, k). To obtain the best path the algorithm uses following formula:

$$D(i,j,k) = d(i,j,k) + \min\{D(i-1,j,k),\ D(i-1,j-1,k),\ D(i,j-1,k)\}.$$

And at the template boundaries with $j = 1$, the between-template transition rules yield:

$$D(i,1,k) = d(i,1,k) + \min\{D(i-1,1,k);\ D(i-1,J(k^*),k^*) : k^* = 1;\ldots,K\}.$$

By using these two relations, the accumulated distances $D(i,j,k)$ can be recursively evaluated point by point.

Once computation is completed, backtracking step recovers the unknown sequence of words. The backtracking information is recorded during the evaluation of the dynamic programming recursion. In backtracking, Ney uses two terms of *from frame* and *from template*, which helps in keeping track of the frames along the pattern time axis from which the best path to the grid point *(i,j,k)* has come. Also *from template* helps keeping track of the respective decision about the recognized word. “It is crucial to realize that for each time frame *i,* only the best template, *i.e.,* the template with minimum accumulated distance at its ending frame, and the corresponding word boundary must be kept track of in order to be able to determine the optimal global path” (Ney 1984).

We present our custom algorithm in Fig. 5. Assume that the x axis is an identifier that could contain one or more words, these words can be the dictionary words or the abbreviation of words. In this example, we have “UserCounterPtr” as an identifier. All words of the dictionary are represented on the y axis. In Fig.5 the dictionary contains 3 words for Counter, User and Ptr. One instance for each dictionary entry as shown in Fig. 5, where at each column end, *e.g.*, column *i*, each word in the dictionary can start at the next position *i+1.* In other words, the algorithm performs a warping of each word and then identifies an optimal path among these warpings (shown by dashed arrows in Fig. 5) to match the identifier.

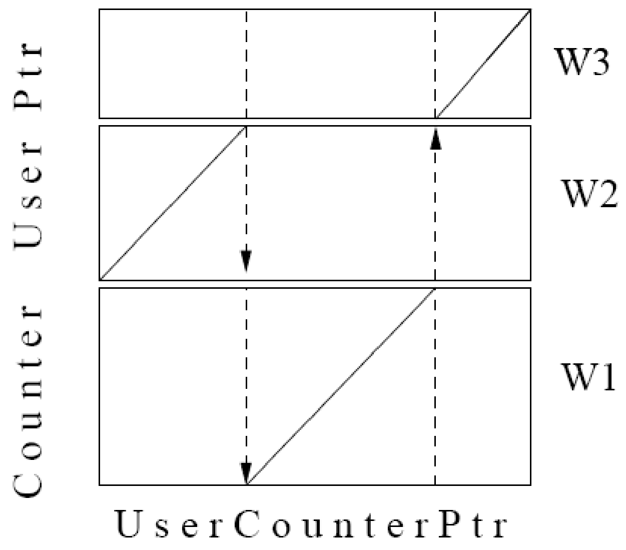


Figure 5 - Connected Word Recognition (Ney 1984)

3.4.7 Initialization Step

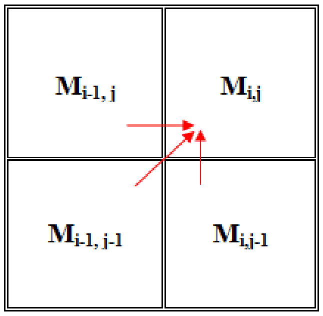
In the first step, we create matrices for all the words of the dictionary with M columns and N rows where M and N correspond to the size of the identifier and dictionary word to be matched. For each word of the dictionary, we initialize two matrixes of M by N; one matrix is filled by the distance values and another matrix is filled with integers that will guide us in back tracing step. For each word in the dictionary, we store the information of position of the word in the dictionary, its score that we calculated in the step of score calculation, so that we can go back and tag word with fitness value.

3.4.8 Distance Computation in Forwarding Template:

In the forwarding matrix we want to find the minimum global alignment score by starting in the bottom left hand corner in the matrix and finding the minimum score $D(i; j)$ for each position in the matrix. To find $D(i; j)$ for any i,j, it is necessary to know $D(i -1; j)$, $D(i; j - 1)$ and $D(i - 1; j - 1)$.

For each position, $D(i; j)$ is defined to be the minimum score at position i,j; i.e.

D(i; j) = Minimum [
$w_1D(i-1; j) + c(i; j)$; ***//insertion***
$w_2D(i; j-1) + c(i; j)$; ***//deletion***
$D(i-1; j-1) + w_3c(i; j)$] ***//match***



In the example, $M_{i-1,j-1}$ will be diagonal, $M_{i,j-1}$ will be vertical and $M_{i-1,j}$ will be horizontal. Consider example of comparison of identifier *pointercntr* with two words of *cntr* and *pointer*; there are two different matrices for these two words. One template of 11*4 for word *cntr* and one template of 11*7 for word *pointer* are created.

We start with left bottom of the template of Fig.6 where both identifier and word start; the comparison starts from this init point. Considering word pointer, the first character in both sequences is *p*. Since this example assumes there is no gap opening or gap extension penalty, the first row and first column of the matrix can be initially filled with 0.

Thus, $M_{1,1\text{-pointer}}$ is calculated as:

D(1; 1) = Minimum [
$w_1D(0; 1) + c(i; j)$;
$w_2D(1;0) + c(i; j)$;
$D(0; 0) + w_3c(i; j)$]

D(1; 1) = Minimum [*0+0* ; *0+0* ; *0+0*]=0

A value of 0 is then placed in position 1,1 of the scoring matrix.

R	6	5	4	3	2	1	0	1	2	3	4
E	5	4	3	2	1	0	1	2	3	4	5
T	4	3	2	1	0	1	2	3	4	3	4
N	3	2	1	0	1	2	3	4	3	4	5
I	2	1	0	1	2	3	4	3	4	5	6
O	1	0	1	2	3	4	3	2	3	4	5
P	0	1	2	3	4	3	2	1	2	3	4
	P	O	I	N	T	E	R	C	N	T	R

R	4	5	6	5	4	5	4	3	2	1	0
T	3	4	5	4	3	4	4	2	1	0	1
N	2	3	4	3	4	4	3	1	0	1	2
C	1	2	3	4	4	3	2	0	1	2	2
	P	O	I	N	T	E	R	C	N	T	R

Figure 6 - Distance Calculation

For word *cntr* the score at the position of 1,1, in the matrix is the comparison of character *p* and *c*: the value of $S_{1,1} = 1$, and we have:

$M_{1,1\text{-cntr}} = \text{Min}[M_{0,0} + 2*1, M_{1,0} + 1, M_{0,1} + 1] = \text{Min}\ [2, 1, 1] = 1.$

D(1; 1) = Minimum [
w_1D(0; 1) + c(i; j) ;
w_2D(1;0) + c(i; j) ;
D(0; 0) + w_3c(i; j)]

D(1; 1) = Minimum [*0+1 ; 0+1 ; 0+2*1*]=1

The same calculation is applied for cell i=2, j=2 of word *cntr* where there are two characters of *O* in identifier *pointercntr* and character *N* from word *cntr* that are compared to each other.

D(2; 2) = Minimum [
w_1D(1; 2) + c(i; j) ;
w_2D(2;1) + c(i; j) ;
D(1;1) + w_3c(i; j)]

D(1; 1) = Minimum [*2+1 ; 2+1 ; 1+1*]=2

The identifier *pointercntr*, should be split to two words of *cntr* and *pointer*. In this example, the value of matrix in a diagonal for 4 last characters *cntr* is zero while this term exists exactly in the word of the dictionary *cntr*. We gain the same result for the 7 first characters of the word *pointer*.

The value of top right in each word shows the score of that word, the lowest score indicate the best word. If an exact splitting happens, we always have zero for this score, but most of the times the identifiers consist of terms that are abbreviated and transformed words of the word and maybe does not exist in the input dictionary. By the above example we considered that word *cntr*, however it is abbreviated of word *counter* but it exists in our dictionary. If this word does not exist, then we had different score from zero for this identifier.

We now show the algorithm of filling both matrixes of forwarding and back pointer. We name the template of field for distance (DL) and the back points (BP):

```
if (t.DL[i-1][j] < t.DL[i][j - 1]  AND
                                          (t.DL[i-1][j]+1) < ( t.DL[i - 1][j - 1]+2*d)){
  // deletion
     t.DL[i][j] = t.DL[i-1][j]+1;
     t.BP[i][j] = t.BP[i - 1][j];

}else if (t.DL[i][j-1] < t.DL[i-1][j] AND
                                          (t.DL[i][j-1]+1) < ( t.DL[i - 1][j - 1]+2*d) {
// insertion
   t.DL[i][j] = t.DL[i][j-1]+1;
   t.BP[i][j] = t.BP[i][j-1];

}else{
// possible substitution
   t.DL[i][j] = t.DL[i-1][j-1]+2*d;
   t.BP[i][j] = t.BP[i-1][j-1];
}
```

Let us consider an identifier of *pointercntr* for this calculation for two words of *cntr* and *pointer*; using the above rules we can fill the backtracking matrix of these words as Fig.7:

Forwarding Matrix:

R	6	5	4	3	2	1	0	1	2	3	4
E	5	4	3	2	1	0	1	2	3	4	5
T	4	3	2	1	0	1	2	3	4	3	4
N	3	2	1	0	1	2	3	4	3	4	5
I	2	1	0	1	2	3	4	3	4	5	6
O	1	0	1	2	3	4	3	2	3	4	5
P	0	1	2	3	4	3	2	1	2	3	4
	P	O	I	N	T	E	R	C	N	T	R

Back pointer Matrix:

R	0	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0	0	0	-4	-7
T	0	0	0	0	0	0	0	0	-4	-7	-7
N	0	0	0	0	0	0	0	-4	-7	-7	-7
I	0	0	0	0	0	0	-4	-7	-7	-7	-10
O	0	0	0	0	0	-4	-6	-7	-7	-8	-10
P	0	0	0	0	-4	-5	-6	-7	-8	-9	-10
	P	O	I	N	T	E	R	C	N	T	R

R	4	5	6	5	4	5	4	3	2	1	0
T	3	4	5	4	3	4	4	2	1	0	1
N	2	3	4	3	4	4	3	1	0	1	2
C	1	2	3	4	4	3	2	0	1	2	2
	P	O	I	N	T	E	R	C	N	T	R

R	0	0	0	0	0	0	0	-7	-7	-7	-7
T	0	0	0	0	0	0	-6	-7	-7	-7	-7
N	0	0	0	0	0	-5	-6	-7	-7	-7	-7
C	0	0	0	0	-4	-5	-6	-7	-7	-7	-10
	P	O	I	N	T	E	R	C	N	T	R

Figure 7 - Forwarding and Back Pointer Matrixes

3.4.9 Trace Back Step

By tracing back the matrix of back pointer, each negative valued cell helps us to find the best points to split our string. We use information of word position in the dictionary and the value of its distance that we have stored in the previous step of forwarding calculation in this part. These data and the negative value of back pointer matrix show us the decomposing point of the identifier to proper terms with the minimum distance. In the example of Fig. 7, negative value of -7 indicates that the identifier of *pointercntr* may be split from the 7th character of the identifier into terms of *pointer* and *cntr*. This is to say the 7th character is the boundary where the path of *cntr* arrives from a different word.

R	6	5	4	3	2	1	0	1	2	3	4
E	5	4	3	2	1	0	1	2	3	4	5
T	4	3	2	1	0	1	2	3	4	3	4
N	3	2	1	0	1	2	3	4	3	4	5
I	2	1	0	1	2	3	4	3	4	5	6

O	1	0	1	2	3	4	3	2	3	4	5
P	0	1	2	3	4	3	2	1	2	3	4
	P	O	I	N	T	E	R	C	N	T	R

R	4	5	6	5	4	5	4	3	2	1	0
T	3	4	5	4	3	4	4	2	1	0	1
N	2	3	4	3	4	4	3	1	0	1	2
C	1	2	3	4	4	3	2	0	1	2	2
	P	O	I	N	T	E	R	C	N	T	R

Figure 8 - Splitting Identifiers

As mentioned above, this process of distance calculation is similar to Levenshtein Distance in determination of the similarity between the strings.

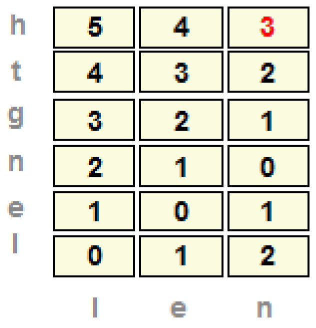
3.5 Levenshtein Edit Distance

In many applications, it is necessary to determine the similarity of two strings. A widely-used notion of string similarity is the edit distance that is the number of deletions, insertions, or substitutions required to transform one string into another string.As an example, if we can consider two strings s="counter" and t="counter", then LD(s, t) =0, because no transformations are needed and the strings are already identical. But if we have s="cntr" and t="pntr", then LD (s,t) = 2, because we need one deletion of character "*c*" and one insertion of character "*p*" to change the string s into t.

The first step in the calculation of LD is the initialization of the matrix. The two strings with *n* and *m* characters build the matrix. The matrix can be filled from the lower left to the upper right corner. Each cell of the matrix indicates the distance of the characters. Horizontal or vertical jumps correspond to an insert or a delete, respectively. The cost is normally set to 1 for each of the operations. The diagonal jump can cost either two, if the two characters in the row and column do not match or 0, if they do. Each cell always minimizes the cost locally. The number in the upper right corner is the Levenshtein distance between both words.

After initializing the matrix, we have to examine each character of both strings. If the characters are equal, there is no distance; the cost is zero unless the cost is one. Each cell of the matrix is set with minimum of its neighbors.

In Fig. 9 we have two strings *len* and *length* thus a matrix 3*6 is initialized for these two string of *len* with 3 characters and string of *length* with 6 characters. The distance between these two strings is 3;



$$\begin{array}{c|ccc}
h & 5 & 4 & 3 \\
t & 4 & 3 & 2 \\
g & 3 & 2 & 1 \\
n & 2 & 1 & 0 \\
e & 1 & 0 & 1 \\
l & 0 & 1 & 2 \\
\hline
 & l & e & n
\end{array}$$

Figure 9 - Matrix of Comparison

In general, the algorithm of Levenshtein Edit Distance works as follow:

Step 1- Matrix initialization:

1.1) Set *n* to be the length of first string of *s* and set *m* as the length of String *t*.
1.2) If *n* = 0, return m and exit and If *m* = 0, return n and exit.
1.3) Construct a matrix containing 0...*m* rows and 0...*n* columns.
1.4) Initialize the first row to 0...n and the first column to 0...m.

Step 2:

2.1) Calculate distance for each character of s (i from 1 to n).
2.2) Calculate distance for each character of t (j from 1 to m).

Step 3:

3.1) If s[i] equals t[j], the cost is 0.
3.2) If s[i] doesn't equal t[j], the cost is 1.

Step 4:

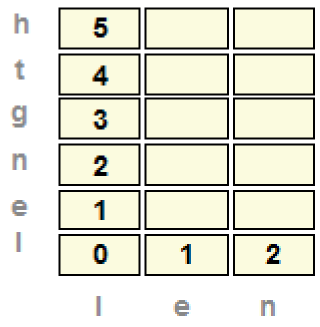
Set cell d[i,j] of the matrix equal to the minimum of:

a. The cell immediately above plus 1: d[i-1,j] + 1.
b. The cell immediately to the left plus 1: d[i,j-1] + 1.
c. The cell diagonally above and to the left plus the cost: d[i-1,j-1] + cost.

Step 5:

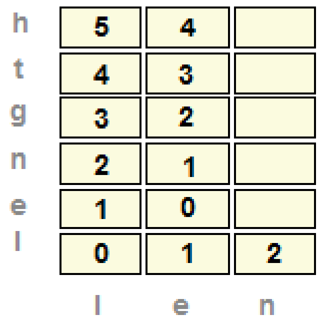
After the iteration steps (2, 3, 4) are complete, the distance is found in cell d[n,m].

As an example, considering two words of *string* and *str*, we have the following steps: Initializing the rows and columns and fill the value of cells for i=1:



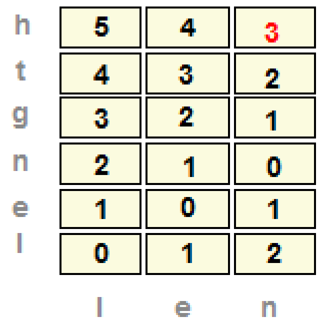
h	5		
t	4		
g	3		
n	2		
e	1		
l	0	1	2
	l	e	n

Step 2,3 and 4 for i=2:



h	5	4	
t	4	3	
g	3	2	
n	2	1	
e	1	0	
l	0	1	2
	l	e	n

Step 2,3,4 and 5 for i=3:



h	5	4	3
t	4	3	2
g	3	2	1
n	2	1	0
e	1	0	1
l	0	1	2
	l	e	n

In the last step, the distance is in the lower right hand corner of the matrix, *i.e.*, 3. This corresponds to our intuitive realization that *string* can be transformed into *str* by substituting *ing* (3 substitution = 3 changes).

Levenshtein Distance counts the differences between two strings, where we would count a difference not only when strings have different characters but also when one has a character whereas the other does not.

3.6 Dictionary Word Selection and Roulette Wheel

Some identifier substrings may not be part of the dictionary and need to be either generated from existing dictionary word or added to it. Thus we must select some words from the current dictionary to generate new words by applying different transformation rules. Selection of the word is possible using the Roulette Wheel algorithm.

Roulette wheel is used for selecting a suitable individual from a population. The most common type of genetic algorithm works as below: The population of a group of individuals is created and then the individuals in the population are evaluated. This evaluation assigns the individuals a score and one identifier is then selected based on its fitness, the higher the fitness, the higher of the chance to be selected.

Pseudo-code for a roulette wheel selection algorithm is shown below:

```
for all members of population
   sum += fitness of this individual
end for

for all members of population
   probability = fitness / sum
   sum of probabilities += probability
end for

loop until new population is full
     number = Random between 0 and 1
     for all members of population
        if number > probability but less than next probability
            then you have been selected
     end for
end loop
```

Considering a roulette wheel in which all chromosomes in the population are placed according to fitness, as in the Fig.10:

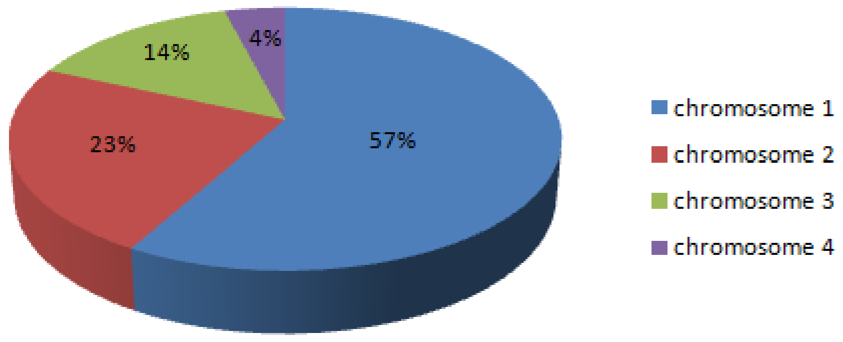


Figure 10 - Roulette Wheel

The best chromosome that has the maximum probabilty to be selected is chromosome 1 that allocate 57% of the percentages of the roulette to itself. In general in tournament selection *n* individuals are selected at random and the fittest is selected. The best chromosome is copied to the population in the next generation. The roulette method of selection will have problems when the fitnesses differs greatly. For example, if the best chromosome fitness is 90% of all the roulette wheel then the other chromosomes will have a slim chance of being selected.

In this thesis, the population of the roulette contains words of the dictionary. Based on the current dictionary, we computed the distance between the input identifier and all words contained in the dictionary and associated to each word of dictionary a fitness value based on its distance computed in the step of calculation. In the algorithm, a list of the words with theirs fitness value is created.

3.6.1 Roulette Wheel Rescaling

There is no way to guarantee that the raw *scores* will fit a certain set of parameters. Instead, we *rescale* the *scores* so that these scores will be suitable for calculating the fitness value of the words. The fitness value of the words is rescaled by subtracting the value of fitness from a fixed value of as the maximum value that should remain always a positive value, *e.g.,* for m=20, word with score of 1, will be rescaled to value of 19. We set the new fitness of the word with the new rescaled fitness.

3.6.2 Roulette Wheel Normalization

After rescaling, we have to normalize the value of the probability in the roulette. We calculate the probability of the word in the roulette by dividing the fitness with the sum of the whole fitness. The fitness is used to calculate the probability of the selection with other words in the roulette.

This fitness level is used to associate a probability of selection with each individual words. If f_i is the fitness of word i in the current dictionary, its probability of being selected is $p_i = \frac{f_i}{\sum_{n=1}^{N} f_n}$, where N is the number of words in the dictionary.

3.6.3 Roulette Wheel Selection

After rescaling and normalizing, we should prepare regions of fitness intervals by assigning probabilities. The region based on the sum of value of word's fitness beside each other. One value is selected randomly. This value is compared to the region. If the value of the random value is less than the region, that region is selected. Each region is limited to one the words in the dictionary. So the word with the biggest region has the highest probability to be selected.

For example for identifier "pntr", we consider 6 words of the dictionary; below you can find the table of fitness:

Table 1 – Fitness table

Word	rectangle	counter	pointer	information	decimal	pntr
Fitness	9.0	6.0	5.0	11.0	7.0	0.0

Related to the Table 1, we will have bellow probabilities:

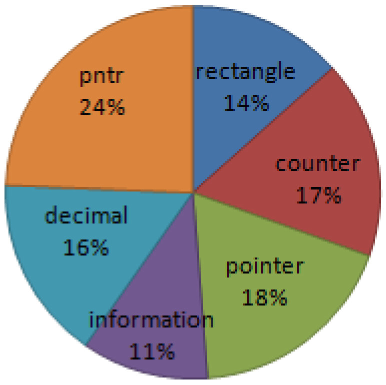
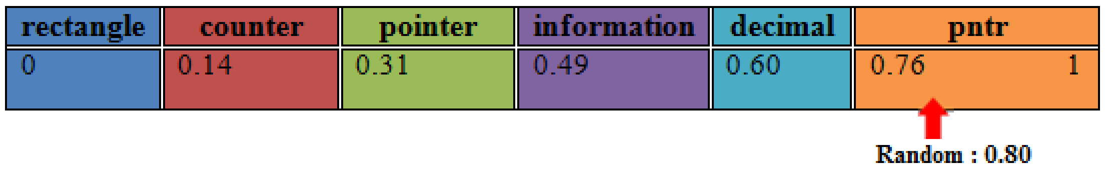


Figure 11 - Roulette Wheel Word's Probability

As we can see in Fig.11, words with lowest fitness value such as *pointer* and *pntr* allocate more places in the figure in comparison to other words of the dictionary (e.g. *pointer* 18% and *pntr* 24%). Thus there is more probability for these words to be selected in the roulette wheel.

In Table 2, we can find the word's region for above dictionary's words; the region is between **0-1** and the new region of each word is the sum of its probability with previous region value. For example word *information* is placed between region value of 0.49 and 0.60. Random number **0.80** is in the region of word *pntr*.

Table 2 – Roulette wheel word's region



rectangle	counter	pointer	information	decimal	pntr
0	0.14	0.31	0.49	0.60	0.76 1

3.7 Word Transformation Rules

Usually, identifiers are built by terms that are taken from words of the dictionary. But some identifier substrings may not be part of the dictionary and need to be either generated from existing dictionary entries or added to it. We have to match a substring of the identifier to the dictionary words. Sometimes omitting some characters of the words causes an exact matching of the substring with that word. Let us consider the identifier of *addlbl* that is built from two substrings of *add* and *lbl*. Clearly, term *add* matches the word of the dictionary if it contains that word. For the substring of *lbl* each word of the dictionary that contains these characters can match this term. If our dictionary contains word *label* then it is possible that this word exactly match the substring of *lbl* by omitting all the vowels of this word (label). For each word of the dictionary we have to determine which word has the minimum distance from the substring and we will select that word. There may exist several words in the dictionary that have the same (minimum) distance from the substring to be matched.

During matching the substring of the identifier with words of dictionary, it can happen that two or three words have the lowest distances. Thus we have to select one of the possibilities to generate the missing dictionary entry term. As an example, 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*. Thus, both words of length and lender are suitable for the substring of the identifier of *fileLen*; we should select one of the words for the modification, maybe omitting all vowels or randomly deleting a character.

To choose the most suitable word to be transformed, we use the following simple heuristic. We select the closest words, with non-zero distance, to the substring to be matched via an algorithm of Roulette wheel. In the algorithm, a list of the words with theirs fitness value is created. Word with minimum value in distance should have the highest probability to be chosen. Then, repeatedly, we modify them using a randomly chosen transformation rule among five 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;
- Delete Suffix;
- Keeping the first n characters ;
- Delete a random vowel;
- Delete a random character.

Delete all vowels:

One of the regular ways of abbreviation of the substrings of the identifier is omitting all the vowels of the word.

In this rule, a word is sent to an algorithm that walks characters by characters of the word and if it face one of the vowels “o”,”e”,”i”,”u” and”a”, it deletes that character and continues for the left characters of the word. Deletion of characters continues until the length of the string is more or equal to three. For example, if we consider the word “pointer”, by this transformation of deleting all vowels, the algorithm returns string “pntr” as a transformed word.

Delete Suffix:

Some words in English end with special suffix, some developers abbreviate identifiers by omitting these suffixes. In this rule, suffixes such as *ing, tion , ment , able , ful , less* are removed from the word.In this algorithm the word of the dictionary is taken as an input and the algorithm consider if this word ends with one of the suffixes.

As an example, we can point to the word *improvement* that is transformed to the word *improve* in this rule by omitting the suffix of *ment* from the word.

Keeping the first *n* characters:

Some developers abbreviate the long length words to the short form by keeping their first 3 or 4 characters. In this transformation, we consider keeping the *n* first characters of the word and the value of n is changeable during transformation.
For example, word *rectangle* will be abbreviated by this rule to *rect* for n=4.

Delete a random vowel:

In this transformation algorithm, one of the vowels of the word is selected randomly. The word is passed to the algorithm of removing the vowel at the selected position that is specified by the function of random. For example word *number* is transformed to *numbr* by selecting the vowel of *e* from the word.

Delete a random character:

One randomly-chosen character is omitted in this rule. For example word *pntr* is transformed to *ptr* by selecting and deleting the character *n* from the word.

The transformations are applied in the context of a hill climbing search. DTW, word transformation rules and hill climbing are the key components of our identifier segmentation algorithm.
Briefly, the algorithm works as follows:

1) Based on the current dictionary :
    1.1) We calculate the distance of each word with the input identifier. We split the identifier using DTW, as explained in Section 3.4.
    1.2) Distance of zero indicates that the substring of the word exists in the dictionary and the identifier can be split in these substrings. For non-zero distances, we will sort all the words of the dictionary by their distance(score).
    1.3) We compute the global minimum distance between the input identifier and all words contained in the dictionary. If the minimum global distance is zero, the process terminates successfully; else

2) For the roulette wheel algorithm that we explained in Section 3.6.
    2.1) We set the words of current dictionary with non-zero distance obtained from step1.1.
    2.2) We associate to each dictionary word a fitness value based on its distance computed in Step 1.1. Usually word with the lowest distance receives the best fitness in the dictionary.
    2.3) Using the fitness value that we calculated in Step 2.2 we have to select one word. The characteristic of the roulette wheel guide us to select an individual from a population with the greates probability to be selected. An individual with the best fitness value allocate the most probability to be selected.
3) In the transformation rules algorithm,
    3.1) We randomly select one transformation not violating transformation constraints, apply it to the word and add the transformed word to a temporary dictionary.
    3.2) 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)
    3.3) If there are still applicable transformations, and the string produced in step (3.1) is longer than three characters, we go to step (3.1).

4) If the global distance is non-zero and the iteration limit was not reached, then, we go to step (1), otherwise we exit with splitting the identifier with substrings that have the minimum score.

These previous steps describe a hill climbing algorithm, in which a transformed word is added to the dictionary if and only if it reduces the global distance. Briefly, a hill climbing algorithm (Michalewicz and Fogel 2004) 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 move to nearby solutions improving the fitness. Different from traditional hill climbing algorithms, in Steps (3.1) and (3.2), our algorithm attempts to explore as much as possible of the neighbors by performing word transformations. Different neighbors can be explored depending on the order of transformations.

CHAPTER 4 EXPERIMENTAL STUDY AND RESULTS

In this chapter, we report results from a preliminary experimental study carried out to analyze the proposed identifier splitting approach, with the p*urpose* of evaluating its ability to adequately identify dictionary words composing identifiers, even in presence of word transformations. In the next subsections, we describe the hypotheses, and the main experimental steps, details about the algorithmic settings, and finally, we present results and their interpretation.

The *quality focus* of the study is the precision and recall of the approach when identifying words composing 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.

Table 3 reports some relevant figures (*e.g.,* number of identifiers) about the two systems.

Table 3 – Main characteristic 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

The dictionary contains about 2,500 words extracted from a glossary found on the Internet[2], 500 most frequent English words[3], plus terms and words contained in Lynx and JHotDraw.

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

4.1 Subject Programs

The first program is a *JHotDraw*[4], which is a well-known Java two-dimensional graphics framework used in drawing 2D graphics and structured drawing editors. JHotDraw started in October 2000 with the main purpose of illustrating the use of design patterns in a real context. It provides support for a range of programs from simple paint package style editors to more complex programs that have rules about how their elements can be used and altered (*e.g.* a UML diagramming tool). It provides support for the creation of geometric and user-defined shapes, editing those shapes, creating behavioral constraints in the editor and animation.

The second program is Lynx. *Lynx*[5] is known as "the textual Web browser", *i.e.*, 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.

4.2 Research Questions

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

1) **RQ1:** *What is the percentage of identifiers correctly split by the proposed approach?* This research question investigates the overall performance of our approach, comparing the results with a manually-built oracle.

2) **RQ2:** *How does the proposed approach perform compared with a 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.

[3] http://www.world-english.org/
[4] http://www.jhotdraw.org
[5] http://lynx.isc.org/

3) **RQ3:** *What percentage of identifiers containing word abbreviations is the approach 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 the proposed approach 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.

We limit our analysis to identifiers longer than or equal to three characters: 2,348 identifiers in JHotDraw and 12,194 identifiers 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 digits.

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 follow a two-step approach. First, we execute the proposed algorithm in a single iteration mode and with no transformations. Thus, only identifiers composed of dictionary words are split with zero distance, see Fig. 13.

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, Fig.14.
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 has been performed, results are compared to 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.

Fig.13. and 14 we illustrate the two phases to answer the first research question.

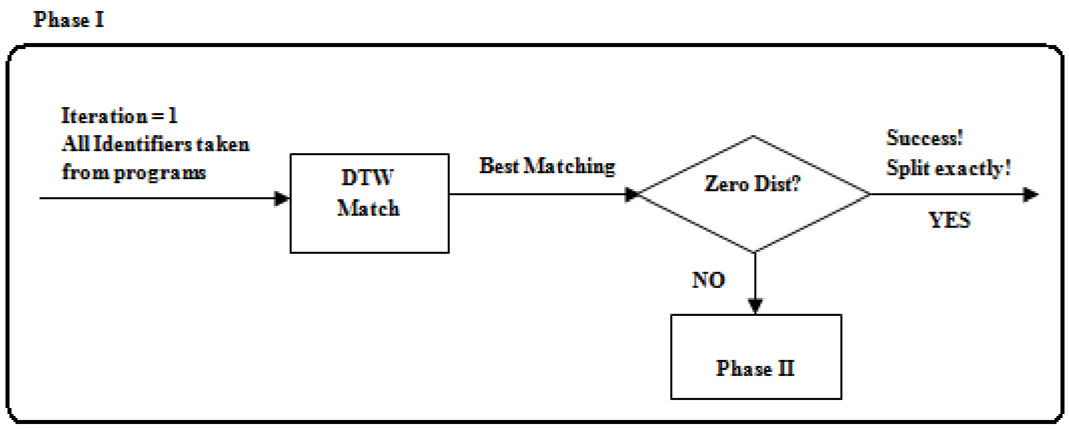


Figure 12 - Phase I of Research Question 1 (RQ1)

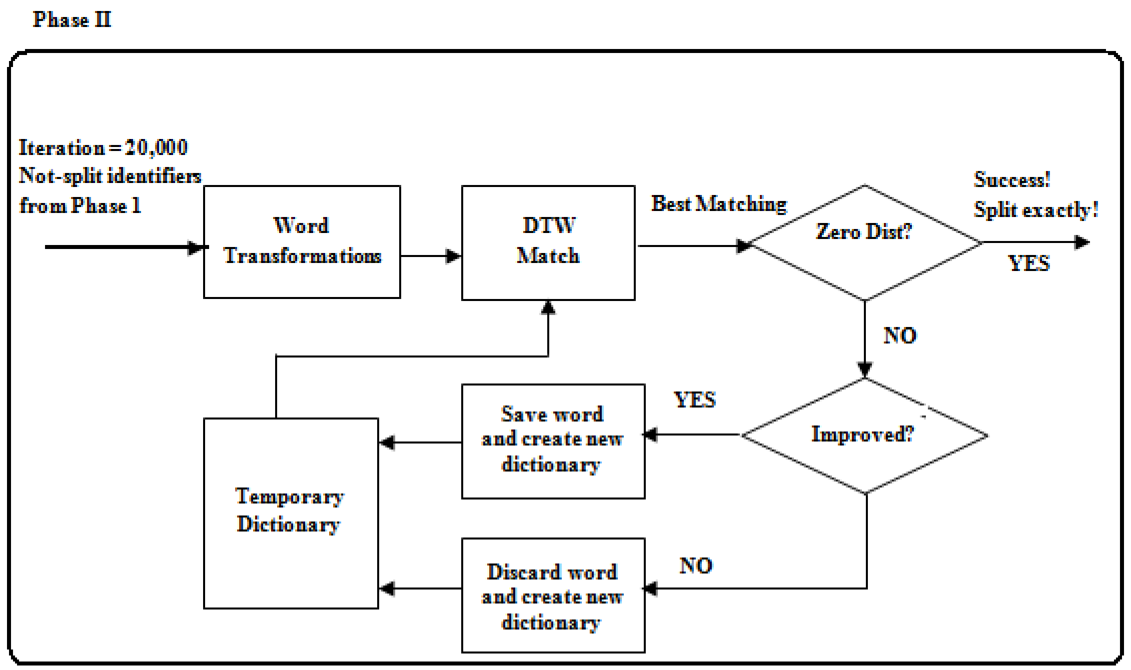


Figure 13- Phase II of Research Question 1 (RQ1)

RQ2 aims at performing a comparison of the proposed approach with a 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 (Sheskin 2007) 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) (Sheskin 2007) 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), 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): Then the odds ratio is:

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

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 our approach to split identifiers when their 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 the results of our study with the objective of addressing our research questions.

1) RQ1: What is the percentage of identifiers correctly split by the proposed approach? Table 4 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 in the first phase.

The fourth column cumulates results of the third columns with the number of composed identifiers made of dictionary words abbreviations split with zero distance within 20,000 iterations. In other words, the fourth column shows the numbers and percentages of all the correctly-split identifiers in second phase. Finally, the fifth column shows the numbers of identifiers that were not exactly split or for which the splitting did not match the oracle. There are 37 of identifiers for JHotDraw and 217 identifiers for Lynx that are split wrongly.

Table 4 – Percentage of correct classifications (RQ1)

Systems	**Identifiers in Oracle**	**Identifiers**	**Exact Splittings**		**Errors**
			Single Iteration	**Multiple Iteration**	
JHotDraw	957	2,348	891 (93%)	920 (96%)	37
Lynx	3,085	12,194	2,169 (70%)	2,901 (94%)	217

Wrong splittings are due to identifiers containing acronyms or too short abbreviations. For example, it may be 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 94% 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 **(RQ1)**.

Table 5 – Performance of the Camel Case Splitter

Systems	**Identifiers in Oracle**	**Identifiers**	**Correct Splittings**	**Errors**
JHotDraw	957	2,348	874	83
Lynx	3,085	12,194	561	2,524

2) RQ2*: How does our approach compares to the Camel Case splitter?*

Table 5 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, unsurprisingly this is not the case of Lynx, the C Web browser. Indeed, C coding standards such as the Indian Hill[6] coding standards or the GNU coding standards[7] do not enforce Camel casing.

When comparing the performances of the proposed approach (see Table 4, considering results after the second phase, *i.e.*, the third column) with those of the Camel Case splitting (see Table 5), 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.

Referring to above comparison we can therefore conclude that the proposed approach performs better than Camel Case splitter on both systems and significantly better on Lynx **(RQ2)**.

3) RQ3: What percentage of identifiers containing word abbreviations is the approach able to map to dictionary words?

Table 4 and 6 reports results aimed at addressing **RQ3**. The fourth and fifth columns of Table 4 shows that a substantial fraction of identifiers containing abbreviations can be split into dictionary words that originate such abbreviations.

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. For example , Lynx prefixes, such as *ly*, *ht*, or *hta*, make it hard to produce correct splittings

[6] http://www.chris-lott.org/resources/cstyle/
[7] http://www.gnu.org/prep/standards/

without a specialized dictionary in which such prefixes are added with, possibly, the proper expansion.

Table 6 – JHotDraw: Results and Statics for selected Identifiers in ten split 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	file 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	11,425	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	
getvadjustable	No						get bad just able	
fimagewidth	No						him age width	
fimageheight	No						him age height	
write**ref**	No						write red	

4.5 Discussion

The proposed approach has a non-deterministic aspect in the way word transformation rules are applied and in the way in which the candidate words to be transformed are selected.

Consequently, different runs of the approach may lead to different identifier splittings. Table 6 reports for a subset of JHotDraw identifiers the splittings obtained in ten runs, each run with an upper limit of 20,000 iterations. The lower part of the table shows identifiers wrongly segmented and for which the zero distance was never achieved.

Two phenomena can be observed. First, because the word *red* is part of the dictionary, the identifier *writeref* is split into *write red* with distance 1; 1 is also the minimum distance and, thus, *red* is always preferred over *reference*. This observation suggests the need for improving the heuristic to select the candidate words to be used in splitting an identifier as any word shorter than *reference* with the current simple heuristic (based on the matching distance) is preferred.

We believe that for words such as *selectionzordered,* j*hotdraw, get*v*adjustable,* f*imagewidth* and f*imageheight*, it would be impossible to compute the correct splitting and identify originating words. For example, in our dictionary the character *f* is contained in about 300 words, each of these words could generate *f* in f*imageheight*. We believe that a substantial reduction of the search space is needed to match single characters, for example, by coupling our algorithm with the approach of Enslen *et al.(Enslen, Hill et al. 2009)*, which would restrict the search to the dictionary words containing *f* to the words used in the same method, class, or package.

It should be considered that the number of authors for each application can have a large influence on our results because of idiosyncrasies and individual quality of each author. We can mention that the programmers of the Java code, e.g., authors of JHotDraw pay more attention to following the naming convention rules such as algorithm of Camel Case, however the programmers of C may not consider the Camel Case rules.

Finally, *serialversion*uid suffers of the problem of acronym expansion mentioned above. We believe that the dictionary should also be expanded with well-known acronyms, such as *afaik*, and with technical abbreviations, such as *uid*, *gid*, *smtp*. In general, the dictionary should contain as many words belonging to the application domain as possible. Requirement documents and user manuals are precious sources of such words.

The upper part of Table 6 shows another limitation of the proposed approach. Sometimes, different component words are discovered in different runs. For example, the identifier *newlen* was split in two different ways: *new length* and *new lender*. Clearly, the latter splitting is semantically wrong: even if *lender* can generate *len*, in the (intended) context of *newlen*, the term *lender* is nonsensical. We believe that the heuristic choosing the words to be transformed needs to be improved, possibly by relying on the strategy derived from(Enslen, Hill et al. 2009), *i.e.*, favoring words already used in the same context.

Finally, it is important to remark that building an oracle for this kind of approach is a difficult and challenging task. Each composed identifier must be split in component words and abbreviations expanded into English words. We have experienced that the task is non trivial: we discovered eight mistakes in the initial JHotDraw oracle, while assessing our approach output and similar errors also occurred in the first version of the Lynx oracle (both oracles were fixed after such runs and the corrected ones were used to produce the results reported in this thesis).

4.6 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 numbers 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. This problem is somehow related to guessing the developers' intent and we can only hope that, given the application 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. We are working on improving our oracle and increasing the number of manually-split identifiers. Variety in the set of developers and different code style should be considered as another threat to validity in the splitting the identifiers. By a different group of developers, projects can be developed in different programming understanding; however it is not obvious and we do not have evidence that various team or group of developers generate different identifiers with level of difficulties that are not the same. It also should be considered that a deterministic model is one in which every set of variable states is uniquely determined by parameters in the model and by sets of previous states of these variables; considering the fact that a stochastic model and variable states are not described by unique values, but rather by probability distributions; applying a roulette wheel algorithm for selection of words from the dictionary produce faster and in some cases better results in comparison of the results that we do not use this algorithm.

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 a Camel Case splitter with our approach. Furthermore, both oracles were built by the same researcher and manually verified by two other 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, and are developed by different teams, with different programming languages. We believe this choice mitigates the threats to the external validity of this study.

CHAPTER 5 CONCLUSION

Systems that are carefully designed and well documented are easier to understand, change and reuse in the future. Most of the times documents are not available or they are not up-to-dates because of time pressure or reduction of the costs. Therefore, the source code of the programs is a key means to support developers during program comprehension. Identifiers and comments in the code are rich source of information for each program. Thus, the proper choice of identifiers can help in promoting software understanding and software evolution. Often, identifiers are created by concatenating English terms and–or acronyms and abbreviated form of words identifying domain concepts. Recognizing terms composing identifiers is a nontrivial task when concatenation does not follow Camel Case rules or when abbreviations are used.

5.1 Summary

In this thesis, we presented an algorithm inspired by Ney's extension of the Dynamic Time Warping (DTW) algorithm to split identifiers into component words. We coupled the DTW extension with transformation rules and hill climbing to infer segmentation in identifiers composed of dictionary words and also of word abbreviations. We applied our approach to split the identifiers of two systems, developed with different programming languages, and belonging to different application domains: JHotDraw and Lynx.

The segmentation process is done in one iteration if the terms contained in the identifiers include words the dictionary. Otherwise, we use transformations rules: we apply a set of successive transformations on dictionary words based on an algorithm of Hill Climbing to match the terms forming the identifier and the created word whose origin comes from words of the dictionary. The exact matching is achieved if the distance between a dictionary word and the term is zero.

Results have been obtained by comparing the obtained splittings with manually-built oracles. They showed that the proposed approach 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.

The usage of techniques inspired from speech recognition is not the only way of splitting identifiers into words. Clearly, when Camel Case separators (or other separators, such as the underscore) are being used, there is no need for complex splitting techniques. However: In some situations, the Camel Case separator or other explicit separators are not used, thus other approaches must be used. A possible alternative approach is the one by Enslen *et al. (Enslen, Hill et al. 2009)*;

The DTW algorithm 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*. It accepts match by identifying the dictionary words closest to identifier substrings; The DTW algorithm is able to perform 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*; thus, it is better than, for example, applying only the Levenshtein edit distance.

The DTW algorithm assigns a distance to matched substrings. Thus, in the identifier of *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 dictionary can be sorted so that the approach favors matching longest words with respect to multiple words composing the longest one. 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.

5.2 Limitations

DTW is a powerful technique but it has also some disadvantages. The first disadvantage is the intrinsic quadratic complexity of a single match with a cubic cost. Furthermore, sentence syntax and semantics are not involved as matching is done at the character level. Going back to the *fileLen* example, *length* should be preferred over *lender*, however DTW cannot choose between the two.

Finally, developers are able to disambiguate complex situations leading to optimal non-zero distance split when DTW cannot. Consider the identifier *imagEdges*; it is immediate to recognize the component words *image* and *edges*. 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.

Our 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, it will correctly split *imagEdges* into *image* and *edges*.

5.3 Future Work

Future work should be devoted to extend the evaluation of our approach to other systems and other splitting algorithms, e.g., Lawrie et al.'s (Lawrie, Feild et al. 2006). In our thesis, we used hill climbing as an algorithm of search-based techniques; this algorithm may be improved in the future. For example, it is possible to introduce enhanced heuristics for term selection and word transformations, with the aim of improving the current performances.

Also,contextualizing our search approach by coupling our algorithm with the approach of Enslen et al. (Enslen, Hill et al. 2009) , which restrict the search to the words used in the same method, class or package, will improve this approach.

Finally, we published our work at the 14th European conference on software maintenance and reengineering in March 2010. The article published is entitled "Recognizing Words from Source Code Identifiers using Speech Recognition Techniques", by Nioosha Madani, Latifa Guerrouj, Massimiliano di Penta, Yann-Gaël Guéhéneuc and Giuliano Antoniol . This paper received the Best Paper award during the conference.

REFERENCES

Alshraideh, M. and L. Bottaci (2006). "Search-based software test data generation for string data using program-specific search operators: Research articles." Softw. Test. Verif. Reliab. **vol. 16, no. 3**: pp. 175–203.

Anquetil, N. and T. Lethbridge (1998). "Assessing the relevance of identifier names in a legacy software system." in Proceedings of CASCON: pp. 213–222.

Antoniol, G., G. Canfora, et al. (2002). "Recovering traceability links between code and documentation." Software Engineering, IEEE Transactions on **28**(10): 970-983.

Biggerstaff, T., B. Mitbander, et al. (1993). The concept assignment problem in program understanding. . Proceedings of the International Conference on Software Engineering (ICSE). Los Alamitos CA, U.S.A., IEEE Computer Society. **482-498**.

Caprile, B. and P. Tonella (2000). Restructuring program identifier names. International Conference on Software Maintenance, 2000.

Caprile, C. and P. Tonella (1999). Nomen est omen: analyzing the language of function identifiers. Sixth Working Conference on Reverse Engineering, 1999.

Cassidy, S. (2002). from http://web.science.mq.edu.au/~cassidy/comp449/html/.

Corbi, T. A. (1989). "Program understanding : challenge for the 1990's. IBM Syst. ." 289--294.

De Lucia, A., M. Di Penta, et al. (2006). Improving Comprehensibility of Source Code via Traceability Information: a Controlled Experiment. 14th IEEE International Conference on Program Comprehension, 2006. ICPC 2006.

Deißenbock, F. and M. Pizka (2005). "Concise and Consistent Naming." Proc. of the International Workshop on Program Comprehension (IWPC) **14**: 261--282.

Demeyer, D., S. Ducasse, et al. (2000). Finding refactorings via change metrics. Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. Minneapolis, Minnesota, United States, ACM**:** 166-177.

Enslen, E., E. Hill, et al. (2009). Mining source code to automatically split identifiers for software analysis. 6th IEEE International Working Conference on Mining Software Repositories, 2009.

Falkenauer, E. (1998). Genetic Algorithms and Grouping Problems, John Wiley \&amp; Sons, Inc.

Fluri, B., M. Wursch, et al. (2007). Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on.

Goldberg, D. E. (1989). Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley Longman Publishing Co., Inc.

Jiang, Z. M. and A. E. Hassan (2006). "Examining the evolution of code comments in PostgreSQL,." in Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006,: ,pp. 179–180.

Lachlan, R. (2007). from http://luscinia.sourceforge.net/page26/page16/page16.html.

Lawrie, D., H. Feild, et al. (2006). Syntactic Identifier Conciseness and Consistency. Source Code Analysis and Manipulation, 2006. SCAM '06. Sixth IEEE International Workshop on.

Lawrie, D., C. Morrell, et al. (2006). What's in a name? A Study of Identifiers. Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on.

Lawrie, D., C. Morrell, et al. (2007). "Effective identifier names for comprehension and memory." Innovations in Systems and Software Engineering **vol. 3, no. 4,** : pp. 303--318.

Levenshtein, V. L. (1966). "Binary codes capable of correcting deletions, insertions, and reversals." Cybernetics and Control Theory **no.10**: 707–710.

Maletic, J. l., G. Antoniol, et al. (2005). 3rd international workshop on traceability in emerging formsof software engineering (tefse 2005). ASE.

Marcus, A. and J. I. Maletic (2003). Recovering documentation-to-source-code traceability links using latent semantic indexing. Software Engineering, 2003. Proceedings. 25th International Conference on.

Marcus, A., D. Poshyvanyk, et al. (2008). "Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems." Software Engineering, IEEE Transactions on **34**(2): 287-300.

Merlo, E., l. McAdam, et al. (2003). "Feed-forward and recurrent neural networks for source code informal information analysis,." Journal of Software Maintenance, **vol. 15, no. 4,**: pp. 205–244.

Michalewicz, Z. and D. B. Fogel (2004). "How to Solve It: Modern Heuristics - (2nd edition)." Berlin Germany: Springer-Verlag.

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

Pigoski, T. M. (1996). Practical Software Maintenance :. Best Practices for Managing Your Software Investment. Wiley, New York, .

Poshyvanyk, D. and A. Marcus (2006). The Conceptual Coupling Metrics for Object-Oriented Systems. Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on.

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

Sheskin, D. J. (2007). Handbook of Parametric and Nonparametric Statistical Procedures, Chapman \& Hall/CRC.

Sim, S. E. and R. C. Holt (1998). "The ramp-up problem in software projects: a case study of how software immigrants naturalize,." in ICSE '98: Proceedings of the 20th international conference on Software engineering. Washington DC USA: IEEE Computer Society, : pp. 361–370.

Sommerville, I. (2000). "Software Engineering. Addison-Wesley, sixth edition."

Storey, M. A. (2006). "Theories, tools and research methods in program comprehension: past, present and future." Software Quality Journal **Volume 14**(3): 187--208.

Sulaiman S., I. N. B., and Sahibuddin S. (2002). Production and Maintenance of System Documentation: What, Why, When and How Tools Should Support the Practice.

Proceedings of the 9th Asia Pacific Software Engineering Conference (APSEC'2002), IEEE Computer Society Press. USA. **pp. 558-567**.

Swanson, B. E. (1976). The dimensions of maintenance. . In Intl. Conf. on Software Engineering,. San Francisco, California,, IEEE Computer Society.**:** 492--497.

Takang, A., P. Grubb , et al. (1996). "The effects of comments and identifier names on program comprehensibility: An experiential study." Journal of Program Languages.