PhD Dissertation

International Doctorate School in Information and Communication Technologies

DISI - University of Trento

Supporting Concept Extraction and Identifier Quality Improvement through Programmers' Lexicon Analysis

Surafel Lemma Abebe

Advisor:

Prof. Paolo Tonella

Software Engineering Unit, Fondazione Bruno Kessler

April 2013

To my family.

Acknowledgement

It gives me great pleasure to acknowledge the people who have supported me and made their contributions in realizing this thesis. First and foremost, I would like to extend my sincere gratitude to my advisor Prof. Paolo Tonella for his constant guidance, scholarly inputs and support throughout my research work. He has always made himself available to clarify my doubts despite his busy schedules and has helped me develop my research skills.

In my PhD journey, thanks to my advisor, I have got a number of opportunities to collaborate with professors and researchers in different labs. The collaborations have helped me to develop my communication skills and were great experiences from which I learnt a lot. This thesis is also the result of the contributions of my collaborators, Prof. Andrian Marcus, Prof. Giuliano Antoniol, Prof. Anna Corazza, Sonia Haiduc, Anita Alicante, and Venera Arnaoudova, and I do hereby like to acknowledge and extend my special thanks to all of them. I would also like to express my gratitude to Prof. Anna Perini who gave me invaluable feedback during the early stage of my research.

I would like to sincerely thank Prof. Anna Corazza, Prof. Paolo Giorgini, and Prof. Lori L. Pollock for accepting to be in my examining committee, and spending their valuable time in evaluating my thesis.

My colleagues and friends have also played an important role in my endeavor. I thank Angelo Susi, Mariano Ceccato, Cu Nguyen Duy, and Andrea Avancini for their comments during my seminars, and informative and relaxing conversations during our lunch and coffee breaks, and Mirko Morandini for additionally organizing wonderful trips and social occasions which are always lively and memorable. I am highly indebted to my friend

and colleague Fitsum Meshesha who is always there for me when I need him the most and helped me to think out of the box. I would like to thank all my friends who gave me moral support and encouraged me to move forward. My special thanks goes to Birhanu Mekuria for giving me his thoughtful feedback on my work, Ephrem Teshale and Gashaw Tadesse for listening to my ideas and helping me balance my life outside work, and Tibebe Akalu for accommodating my time constraints in his program and letting me share his important life mile stone.

I would like to extend my profound thanks to my parents and sisters who have always been by my side despite the distance. Their wholehearted love, consistent advice, and encouragement have given me the strength to go through my difficult times. Had it not been for them, I could not have reached where I am now.

Abstract

Identifiers play an important role in communicating the intentions associated with the program entities they represent. The information captured in identifiers support programmers to (re-)build the "mental model" of the software and facilitates understanding. (Re-)building the "mental model" and understanding large software, however, is difficult and expensive. Besides, the effort involved in the process heavily depends on the quality of the programmers' lexicon used to construct the identifiers.

This thesis addresses the problem of program understanding focusing on (i) concept extraction, and (ii) quality of the lexicon used in identifiers. To address the first problem (concept extraction), two ontology extraction approaches exploiting the natural language information captured in identifiers and structural information of the source code are proposed and evaluated. We have also proposed a method to automatically train a natural language analyzer for identifiers. The trained analyzer is used for concept extraction. The evaluation was conducted on a program understanding task, concept location. Results show that the extracted concepts increase the effectiveness of concept location queries. Besides extracting concepts from the source code, we have investigated information retrieval (IR) based techniques to filter domain concepts from implementation concepts.

To address the second problem (quality of the lexicon used in identifiers), we have defined a publicly available catalog of lexicon bad smells (LBS)

and developed a suite of tools to automatically detect them. LBS indicate some potential lexicon construction problems that can be addressed through refactoring. The impact of LBS on concept location and the contribution they can give to fault prediction have been studied empirically. Results indicate that LBS refactoring has a significant positive impact on IR-based concept location task and contributes to improve fault prediction, when used in conjunction with structural metrics. In addition to detecting LBS in identifiers, we try also to fix them. We have proposed an approach which uses the concepts extracted from the source code to suggest names which can be used to complete or replace an identifier. The evaluation of the approach shows that it provides useful suggestions, which can effectively support programmers to write consistent names.

Keywords

Concept extraction, lexicon bad smells, identifier parsing, domain concept filtering

Contents

List of Tables

List of Figures

Chapter 1

Introduction

To understand and maintain a software, developers try to explore and gather information from various software artifacts such as the source code, the design, and requirement documents. Artifacts such as design and requirement documents however are often not available or up-to-date. Hence, developers opt to rely on the source code as their primary source of information [104].

The source code is a formalized representation of a solution to a given requirement in a domain. Despite the formalization, the source code contains a lot of textual information which is not strictly formal. In fact, approximately 70% of the source code is composed of identifiers [39], which are freely chosen by developers to communicate their intentions [98]. Understanding the source code requires a developer to acquire the intention embedded in the identifiers and (re-)building a "mental model"of the system/domain.

Understanding the source code written by others or written some time ago however is usually a difficult and time consuming activity. The difficulty stems from the fact that a solution of a problem in a domain can be formalized in a number of ways and different developers might use a different lexicon to express their intention through identifiers. A study

conducted by Furnas *et al.* [46] has shown that the probability of two people naming the same object with the same name is between 7% and 18%. This characteristics apply also to developers naming a concept and results in identifiers which are not consistent and concise in representing a given concept. Often there is no way of knowing how developers represent their intention other than by reading the code. Reading the code of a large program and building a "mental model"on the other hand is an expensive activity, highly influenced by the quality of the identifiers [107].

Quality is subjective and its definition is relative to the person receiving the final product [107]. Developers usually agree on some quality attributes of the source code and use them by convention. For example some companies and open source communities have adopted coding conventions such as the Java coding conventions[1]. Despite the effort and agreements on conventions, some source codes contain poor quality identifiers. For instance, in some legacy systems Sneed [106] has observed that programmers often choose to name procedures after their girlfriends or favorite sportsmen.

If the quality of identifiers is poor, it hinders the process of reading and understanding the source code which has a negative impact on maintenance and evolution of the software. Realizing the difficulty associated with poor quality identifiers on reading and understanding the source code, some developers rename identifiers to meaningless names to obfuscate their source code, and hence protect their intellectual property from being copied [109]. While such obfuscation is done before distributing the software, the quality of identifiers in the source code on which the developers are working is supposed to be good. This is often true for open source software, where developers located at different places collaborate to maintain and evolve a common code base.

The *theory of broken windows* [113] states that if a window in a building

[1] http://www.oracle.com/technetwork/java/codeconv-138413.html

is broken and is *left unrepaired*, all the rest of the windows will soon be broken. Like wise in the source code being unable to easily identify developer's intentions and to re-build her "mental model", and having poor quality identifiers could be an indication and a driving factor for other serious problems. Hence, we believe that addressing these problems have a twofold advantage: (a) it prevents developers from misunderstanding the code functionality and introducing new problems and (b) it improves or maintains the quality of the source code over time. In this thesis, we present techniques and methodologies which help developers in extracting the knowledge embedded in the source code through reverse engineering, in locating problems related to the quality of identifiers and in preventing them from being introduced.

1.1 Research problem

Problem 1: ***Concept extraction.*** Understanding a program involves learning the concepts implemented in the source code and the relations among them. When programmers are given a task, they have to decide how to structure and implement the knowledge they have about the solution domain. This knowledge is encoded in program syntax, comments, but most of all in identifiers. For maintainers and even for the first developers of the software, after some time, (re-)acquiring the encoded knowledge from the source code and understanding the program may be a difficult and expensive task, especially for large systems. In fact different programmers model the concepts of a domain, and the relations among them in various ways, and represent them in the source code differently. We intend to address the problem of extracting the concepts from the source code to help program understanding.

Problem 2: ***Improving the quality of the lexicon used in iden-***

tifiers. The effort put to read and understand a program depends on the quality of the lexicon used to represent the concepts in the identifier. Despite the fact that programmers are free to use any lexicon to give a name to a concept in their mind, they try to follow commonly adopted conventions to make the name meaningful and consistent. However, due to various factors, the lexicon selected to give names to different program elements is not always consistent and concise in conveying the intended meaning. In addition, different programmers might also follow different naming patterns, which create confusion to the reader of the program. Such a misunderstanding could also introduce ambiguous representations of concepts during maintenance. Hence, it adds more difficulties to the already difficult problem of program understanding. In this regard, we intend to identify and suggest ways to improve the programmer's lexicon used in identifiers, which may compromise the quality of the source code.

1.2 Contribution

The main contributions of this thesis are techniques and methodologies for extracting concepts and inter-concept relations from the source code and for improving the quality of identifiers.

Concept extraction. We have defined a natural language based methodology to extract concepts and relations among concepts. The methodology uses natural language analyzers to identify concepts and inter-concept relations from phrases constructed by splitting identifiers. We have also proposed and investigated different techniques to adapt and use natural language analyzers with phrases constructed from splitted identifiers. Besides the natural language based methodology, we have also formulated an approach which exploits structural information in the source code to extract concepts and inter-concept relations.

In addition to extracting concepts used in source code, we have investigated information retrieval (IR) based approaches to separate (filter) domain concepts from implementation concepts.

Improving the quality of the lexicon used in identifiers. We have introduced the notion of "lexicon bad smell", which indicates some bad practices on the choice of the lexicon and on the construction of identifiers. We have created a catalog of lexicon bad smells and developed a publicly available suite of detectors to locate them. We have also proposed suggestions which can be used to refactor the identifiers with a bad smell and to improve their quality.

Identifier suggestion. To support programmers in writing good quality identifiers, we have formalized a new approach to suggest identifiers. The suggestion can be used to replace an existing identifier or to complete a new identifier. The approach exploits the concepts extracted from the source code following our natural language based approach and ranks the suggested names taking into account the context in which the identifier is written.

1.3 Structure of the thesis

An overview of the thesis structure is shown in Figure 1.1. The approaches we proposed to extract concepts from the source code and their evaluations are presented in Chapter 2. Chapter 3 presents the approaches investigated and evaluated to filter domain concepts from the concepts extracted following the approaches discussed in Chapter 2. The description of the catalog defined to identify bad practices in naming identifiers, the corresponding detectors, and evaluations are presented in Chapter 4. In Chapter 5 an approach, relying on the methodology described in Chapter 2, which provides a ranked list of name suggestions, to replace an existing or complete

a new identifier is presented. Chapter 6 discusses the related works in concept extraction from the source code and methods proposed to improve the quality of identifiers. The conclusion of the thesis and future works are presented in Chapter 7.

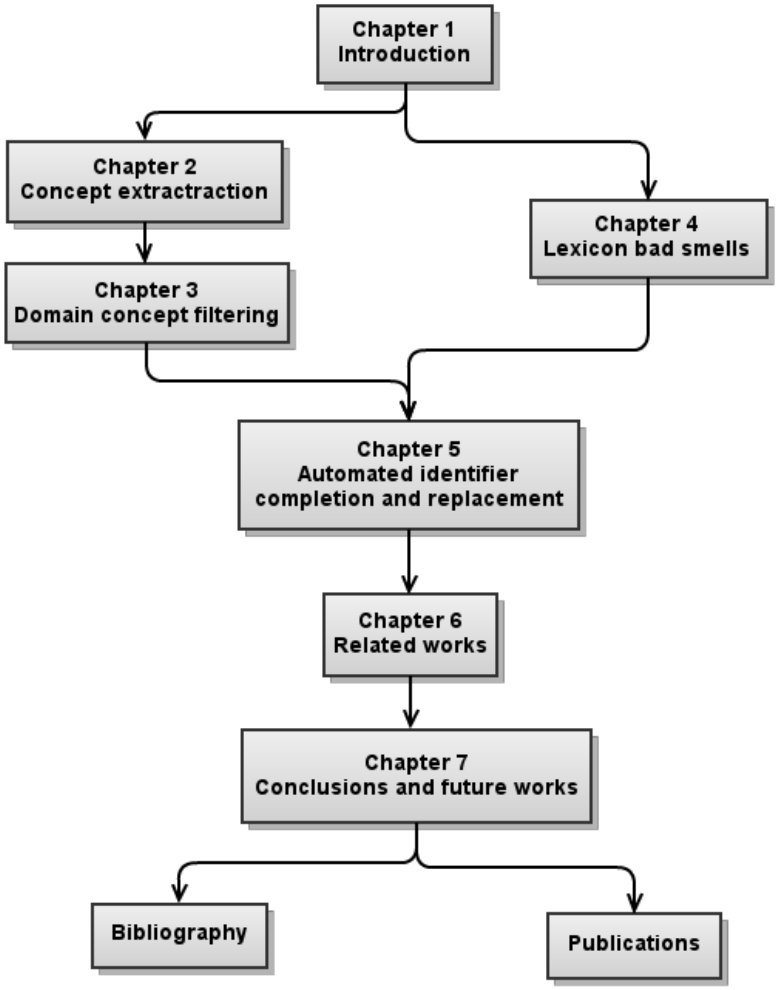

Figure 1.1: Structure of the thesis

Chapter 2

Concept extraction

Program understanding involves (re-)building the mental model of the knowledge captured in a program. Rebuilding the mental model of a program requires identifying the concepts in the source code and the relations among them. However, this task may be difficult for programmers having different perceptions and representations of the concepts introduced during the development and evolution of the source code.

One way for programmers to represent and communicate the intended meaning of program elements is through identifiers [98] which constitute approximately 70% of the source code [39]. Different approaches have been proposed to analyze and exploit the information captured in identifiers to support program understanding. Ratiu *et al.* [102] have developed an approach to automatically extract a domain ontology from different domain specific APIs that target the same domain. The extracted ontology is composed of all prevalent domain concepts in the APIs. In other works [83, 97, 105, 47] the authors have proposed approaches which minimize the effort required to go through the source code and understand parts of the source code relevant to a concept.

In this chapter, we present two approaches to extract concepts from the source code and build an ontology, which supports program understand-

ing. The first approach exploits the natural language information captured in the identifiers [4], while the second approach exploits the structural relations among source code elements. Based on the level of formality, an ontology can vary from a simple taxonomy with almost no formalization, to one which uses a rigorously formalized theory [110]. Ontology in this context is a "lightweight ontology" which is in between these two extremes and does not include axioms supporting formal reasoning, but only considers concepts and relations connecting the concepts. Lightweight ontology which is built using only concepts and relations connecting the concepts without any formalization is sometimes referred to as "concept map". Here after we refer to such lightweight ontology as ontology.

To exploit the natural language information captured in the identifiers, we use natural language analyzers. A natural language analyzer is a natural language tool which takes an input sentence, that is a string of words, and returns its syntactic analysis. To carry out syntactic analysis on identifiers and exploit the information captured in the identifiers, we have investigated an approach to train natural language analyzers for use with identifiers. We present our proposed approach which adapts natural language analyzers to identifiers in Section 2.1. Section 2.2 and 2.3 discuss the approaches we propose to extract concepts and build an ontology from the identifiers and the structure of the source code, respectively. The comparisons of the different types of analyzers, and the two types of concept extraction approaches are presented in Section 2.5.

2.1 Identifier parsing

Natural language analyzers are mainly conceived to work with full sentences. The term lists which are obtained by splitting identifiers however are different from sentences. We present some heuristics to convert an iden-

tifier term list into a sentence [4], so that it can be handled by a general purpose parser which we call Standard English Analyzer (SEA). We also investigate an analyzer constructed to directly work on the identifier term list.

In the following sub-sections, we describe the syntactic analysis approaches followed to parse identifiers, including the analyzers and the corresponding training sets. The steps involved in the construction of the sentences which are used by SEA are also described below.

2.1.1 Syntactic analysis

In NLP it is well known that a syntactic analysis is necessary to reconstruct the meaning of the input sentence. In our case, the relations between the entities (concepts) included in the identifier term list depend on the syntactic and semantic role of each token. A syntactic parse is therefore necessary for further processing.

The construction of the syntactic analysis for an input identifier can be performed in different ways and requires several steps. The first step in all cases, however, is *tokenization*. *Tokenization* is the process of splitting a text into words or linguistic elements called *tokens* or *terms*. Identifiers are composed of one or more terms. In order to identify the composing terms and tokenize identifiers, we take advantage of the commonly used term separators, such as camel casing (*e.g.*, *FileItem*) and underscore (*e.g.*, *file_item*). This can also be achieved using more sophisticated techniques proposed by Lawrie *et al.* [72], and Corazza *et al.* [36]. When the terms used to construct the identifiers are abbreviations or contractions, they can be expanded using existing approaches [70, 73, 57, 71, 36]. For example, by tokenizing the identifier *fileItem*, we get the term list <file, item>.

The syntactic analysis includes two modules: *PoS tagging* and *parsing*.

The former assigns a label corresponding to the function of the word in the sentence, such as *noun, verb*, and so on, to each token, while the latter constructs a syntactic analysis of the whole sentence. The syntactic analysis can be hierarchically organized phrases, if a constituency based approach is adopted, or a set of dependencies between word pairs composing a directed graph in case of a dependency based approach. In the latter case, by *dependency* relationship we mean an asymmetric binary relationship between a token called *head*, and another token called *modifier* (see Lin [78]). In our case we consider a *dependency parser* where the analysis is formed of dependencies between pairs of input words. The *dependency parser* is chosen over the *constituency parser* because it allows a more direct reconstruction of relations between concepts.

The two modules can be organized to work in a *pipeline* or *integrated*. While in the pipeline analyzer PoS tagging is completed before the syntactic parse is built, in the integrated schema a list of possible PoS tags is associated to each token in the input by consulting the lexicon, and the choice of the best PoS tags is performed during parsing. In other words, in the integrated analyzer the parser is also involved in the decision of the PoS tag which is more likely in the considered sentence.

Some NLP systems use a data-driven natural language parser which requires a training phase to learn how to process the input tokens. The advantage of such data-driven parsers is that they can easily learn how to parse different (novel) languages and their variants from a collection of suitable parse trees, called *treebank*.

We consider three NLP systems. The first NLP system, which is applied to the token sequence, is similar to the analyzer of the standard English, SEA. Its two analyzer modules are trained on a largely employed English treebank, namely the PennTreebank (see Marcus *et al.* [84]). Both remaining NLP systems involve retraining the two analyzer modules to adapt

them to the identifier language, and only differ in the architecture, being *pipeline* or *integrated*.

Training is performed on an annotated set which should be as similar as possible to the actual input set, hence, in our case, to tokenized identifiers. Indeed, the string of tokens extracted from an identifier is very different from a natural language sentence as identifiers usually do not correspond to complete sentences. In addition to that, they also have a different structure depending on their function: method names are more likely to describe actions and therefore their structure resembles the Verbal Phrases (VPs), while attribute and class names usually aim at indicating things, in a way similar to Noun Phrases (NPs). This is, for example, the case of the two examples reported in Figure 2.1 which shows the analysis of two identifiers: *TextPanel* and *removeFile*. The former corresponds to a class identifier while the latter to a method name. Such distinction is expected to affect the syntactic analysis, but not the PoS tagging. Therefore, we will consider a unique PoS tagger, but a different parser for each of the two classes of identifiers, namely a VP-parser for method names and an NP-parser for all the others (classes and attributes). Consistently, we construct two different training treebanks, one for each parser. In Section 2.1.4, we present details of the training set construction.

All in all we therefore consider four NLP systems:

1. *Untrained Integrated Analyzer (UIA)*: consists of an SEA integrating PoS tagging and dependency analysis, applied to complete sentences built by padding the token sequence produced by the tokenizer. The system architecture is shown in Figure 2.2;

2. *Untrained Pipeline Analyzer (UPA)*: to overcome the need for complete English sentences, a pipeline composed of standard English PoS tagger and parser is directly applied to the token sequence extracted

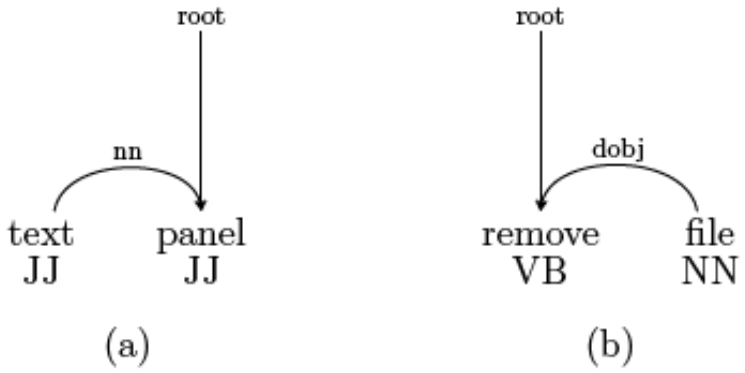

Figure 2.1: Examples of dependency analysis for two identifiers: (a) the class name composed by two tokens, **text panel**, and (b) the method name formed by two tokens, **remove file**. Each graph node is labeled by one token of the identifier and the PoS tag assigned to the token. The edge labels are dependency relationships extracted by the analyzer, namely **dobj** - direct object and **nn** - noun-noun specifier.

from the input identifier: the analysis in this case is obviously more difficult than in the previous case and a more accurate analyzer is required. The pipeline architecture is depicted in Figure 2.3;

3. *Trained Pipeline Analyzer (TPA)*: both syntactic analysis modules, namely PoS tagger and parser, are retrained to adapt them to the token language and then they are combined in a pipeline and directly applied to the tokenizer output (see Figure 2.4). Note that in this case two different parsers (*i.e.*, VP and NP) are used, depending on the function of the input identifier;

4. *Trained Integrated Analyzer (TIA)*: this system is identical to the previous one, except that the two syntactic analysis modules are integrated together to improve robustness towards PoS tagging errors, as depicted in Figure 2.5.

The sentence construction module is adopted only in the UIA, while in all other cases the analyzer is modified to directly process the tokenizer

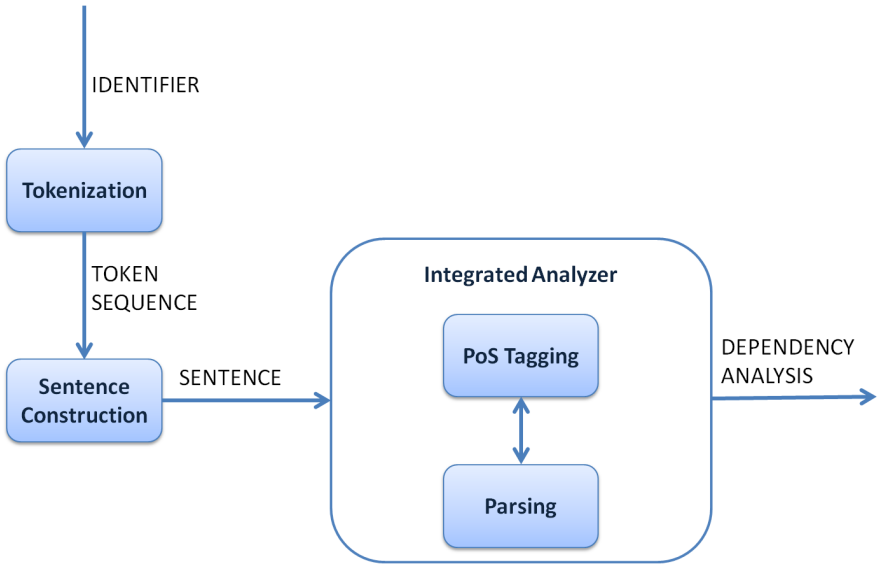

Figure 2.2: Architecture of Untrained Integrated Analyzer (UIA).

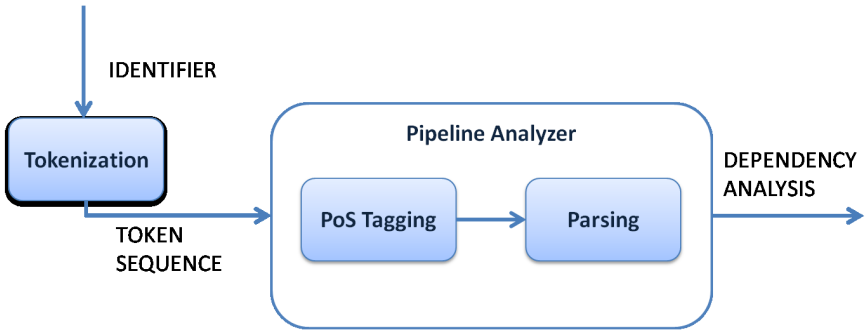

Figure 2.3: Architecture of Untrained Pipeline Analyzer (UPA): the two analyzer modules are applied as distributed, without retraining, and in a pipeline.

output.

2.1.2 Sentence construction in UIA

The sentence construction step in the UIA system (see Figure 2.2) aims at constructing a sentence which is used as an input to the integrated analyzer.

To generate a sentence from an identifier term list, we have formulated different rules which are shown in Table 2.1. The rules are defined for the

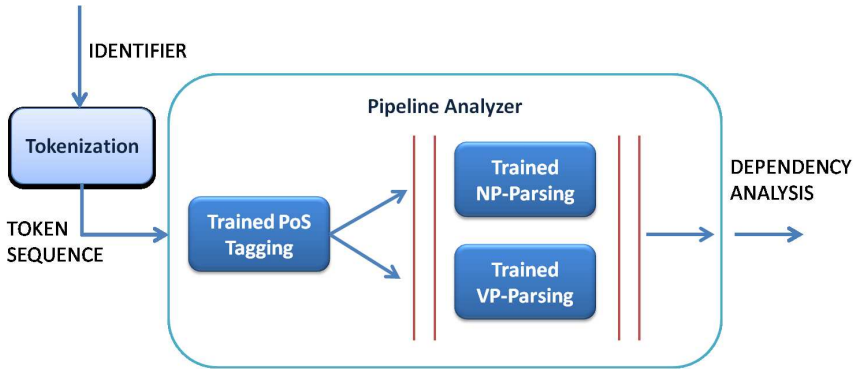

Figure 2.4: Architecture of Trained Pipeline Analyzer (TPA): a pipeline of the two re-trained modules is directly applied to the tokenizer output.

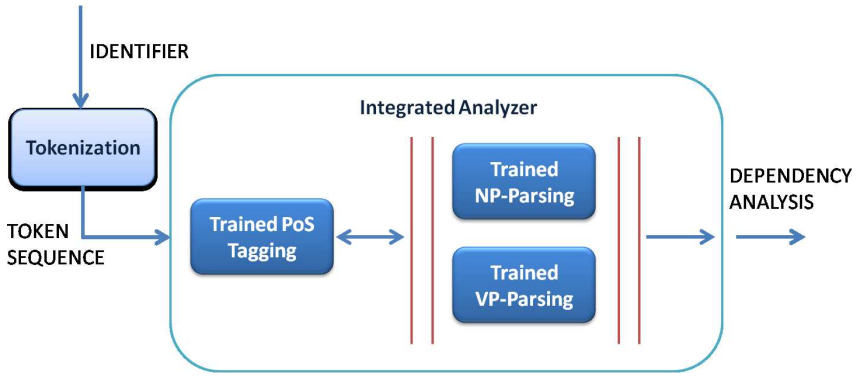

Figure 2.5: Architecture of Trained Integrated Analyzer (TIA): the two modules are retrained and applied in an integrated modality.

three main identifier types: *class*, *method*, and *attribute*. A rule is applied to a given type of identifier term list when the corresponding constraint is satisfied. To know the parts of speech of the terms and see if a constraint is satisfied, we have used WordNet [89, 44]. If none of the constraints are satisfied, the list of terms is used as it is. Based on the rules we defined, for a given term list, at most two candidate sentences are generated. Of the two candidate sentences one is selected for further processing.

To elaborate the steps involved in the sentence construction, we use the code snippet shown in Figure 2.6 as our running example.

Table 2.1: Rules to generate sentences from term lists.

Rule	**Class term list**	**Generated sentence**	**Constraint**
CR1	$C = \langle T_1 \rangle$	T_1 *"is a thing"*	T_1 is a noun or an adjective
CR2	$C = \langle T_1 \rangle$	$T_1 er$ *"is a thing"*	T_1 is a verb
CR3	$C = \langle T_1, T_2, \ldots \rangle$	$T_1 T_2 \ldots$ *"is a thing"*	T_1 is a noun or an adjective
CR4	$C = \langle T_1, T_2, \ldots \rangle$	$T_1 ing\ T2 \ldots$ *"is a thing"*	T_1 is a verb
Rule	**Method term list**	**Generated sentence**	**Constraint**
MR1	$M = \langle T_1 \rangle$	*"Subjects"* T_1 *"object"*	T_1 is a verb
MR2	$M = \langle T_1 \rangle$	*"Subjects get"* T_1	T_1 is a noun
MR3	$M = \langle T_1, T_2, \ldots \rangle$	*"Subjects"* $T_1 T_2 \ldots$	T_1 is a verb
MR4	$M = \langle T_1, T_2, \ldots \rangle$	*"Subjects get"* $T_1 T_2 \ldots$	T_1 is a noun or an adjective
MR5	$M = \langle T_1, T_2, \ldots \rangle$	*"Subjects handle"* $T_2 \ldots$	T_1 is the preposition "on"
MR6	$M = \langle T_1, T_2, \ldots \rangle$	*"Subjects convert"* $T_2 \ldots$	T_1 is the preposition "to"
Rule	**Attribute term list**	**Generated sentence**	**Constraint**
AR1	$A = \langle T_1 \rangle$	T_1 *"is a thing"*	T_1 is a noun or an adjective
AR2	$A = \langle T_1 \rangle$	$T_1 er$ *"is a thing"*	T_1 is a verb which is not a past participle, or T_1 is a past participle verb and A is not of boolean type
AR3	$A = \langle T_1 \rangle$	T_1 *"subjects are things"*	T_1 is a past participle verb and A has a boolean type
AR4	$A = \langle T_1, T_2, \ldots \rangle$	$T_1 T_2 \ldots$ *"is a thing"*	T_1 is a noun or an adjective
AR5	$A = \langle T_1, T_2, \ldots \rangle$	$T_1 ing\ T_2 \ldots$ *"is a thing"*	T_1 is a verb

Generating candidate sentences

Class term list: A class term list is converted to sentences using the rules shown in Table 2.1 (top). Class identifiers are usually constructed from a noun, multiple nouns or adjectives followed by nouns. The rules take advantage of this general nature of class identifiers to suggest the formulation of sentences. To construct a sentence from the list of terms produced from a class name, we append "is a thing" at the end of the list. When the first term in the sequence can (also) be used as a verb, we add either "er" or "ing" to it. "er" is appended if the phrase has only one term while "ing" is appended when there is more than one term.

If, for example, the class term list contains only a single term (*e.g.*, the super-class in our running example, <*account*>), by applying CR1 and

```
public class CurrentAccount extends Account
                    implements DigitalCheque {
   private double balance;
   private boolean closed;
   public double getBalance(){
      return balance;
   }
   public double transactionPayment(double amount){
      ServiceCharge sc = new ServiceCharge("currentAccount");
      double charge = sc.calculate();
      ...
   }
   public void withdraw (Money m){...}
   public void close () {...}
   public boolean isClosed(){...}
   public double calculateInterest(){...}
   ...
}
-----------------------------------------------------------------
public class BankSystem{
...
   public void main(){
      ....
      CurrentAccount ca = new CurrentAccount();
      ...
      ca.withdraw(new Money("10", "Euro"));
      ...
      ca.close()
   }
...
   private void authenticate (User user){
      Login login = new Login();
      ...
      login.authenticate(user);
      ...
   }
}
```

Figure 2.6: Running example: A fragment code of a *Bank System*.

CR2, we get "account is a thing" and "accounter is a thing" respectively, since *account* can be used as both a noun and a verb. If the class list of terms has more than one term, for example <*current, account*> (as in Figure 2.6), we will have *"current account is a thing"* by applying CR3. As *current* can only be used as an adjective or a noun, rule CR4 is not applied.

Method term list: Method identifiers often consist of a single term which can be a verb or a noun, or multiple terms which start with a verb, followed by nouns, which usually serve as the object of the verb. In addition to these, we have included two special types of method name term lists which are commonly used to describe methods that deal with events and conversion. The former type of methods usually starts with the preposition "on" while the latter starts with "to". The possible sentences generated for these term lists are shown in Table 2.1 (middle).

If a method name is constructed from a single term which can be used as a verb or a noun, we apply rules MR1 and MR2. When the term is considered as a *noun* and a missing verb is anticipated (MR2) "get" is used to construct the sentence. For example, if we take the term <*close*> extracted from the method name of our running example (see Figure 2.6), two candidate sentences are generated by applying rules MR1 and MR2: *"Subjects close object"* and *"Subjects get close"*, since the term *close* can be used as both a verb and a noun. For term lists containing two or more terms, we apply rule MR3 if the first term is a verb and rule MR4 if it is a noun. For MR3, if the first term in the term list is the verb *is*, we use *Subject* instead of *Subjects* while constructing the sentence. In our running example, for the method term list <*transaction, payment*>, the sentence *"Subjects get transaction payment"* is generated by applying MR3. No alternative candidate sentence is generated for this term list as the term "transaction" can only be used as a noun. In addition to these rules, we have two special rules, MR5 and MR6, for the two most commonly used prepositions in method names. If the leading term of a method name is the preposition "on" or "to", we replace them with the equivalent verbs "handle" and "convert", respectively, during sentence construction. If for example we have the method term list <*on, click*> and <*to, euro*>, the sentences *"Subjects handle click"* and *"Subjects convert*

euro" are generated using rules MR5 and MR6, respectively.

Attribute term list: Attribute identifiers are usually similar in nature to class identifier. They are mostly constructed from a noun, multiple nouns, adjectives followed by nouns, or a verb which is in its past participle form and has a boolean return type. To construct a sentence from the list of terms created from attribute identifiers, we follow similar construction techniques as for the class identifiers. The sentences are generated by appending "is a thing" or "subjects are things" at the end of each term list. The latter sentence is used when we have an attribute term list composed of a single term which is in its past participle form and has a boolean type (AR3). AR3 is a special case where the verb becomes a modifier of the subject in the clause "subjects are things". The first term in the list is also modified by adding "er" or "ing" when we have a verb that is not in its past participle form and the associated attribute has a boolean type. "er" is appended if the term list is constructed from only one term, while "ing" is appended when there is more than one term. The summary of these rules is shown in Table 2.1 (bottom).

If we take attribute name <*balance*> as an example from our running example, we generate two candidate sentences "balance is a thing" and "balancer is a thing" using rules AR1 and AR2, since the term *balance* can be used as both a noun and a verb. By applying AR3, the boolean attribute <*activated*> will generate the candidate sentence "activated subjects are things".

Candidate sentence selection

The rules shown in Table 2.1 and described above generate one or two sentences. When we have two sentences, we need to select one sentence which will be used as an input to the following steps. Prerequisite to

the selection of a sentence is the generation of dependency trees for the initial candidate sentences using a SEA (*e.g.*, Minipar). Once parse trees are available for the candidate sentences, we apply the following selection criteria, in the given order (the first match is applied, without considering the next ones). If we have only one candidate sentence, it is automatically selected.

a. If only one of the sentences is correctly parsed, select the sentence whose parse tree is correct. When SEA is not able to identify a term in a sentence and parse the sentence correctly, *Unknown* (*U*, for short) is reported. Hence, if just one of the two sentences has a *U*, the sentence without *U* is selected.

b. If both sentences do not have a *U* and the source of the terms is a method, the method name is checked against the attributes of the enclosing class. If a match is found, the sentence with the verb *get* is selected.

c. If both sentences do not have a *U*, the sentence satisfying the following priority rules is selected: If the two sentences have been generated for a method term list, the sentence constructed using either rule MR1 or MR3 is selected. For sentences generated from a class term list, the sentences generated following either rule CR1 or CR3 is given priority, while for attribute term lists, either rule AR1 or AR4 has priority.

d. If both sentences have a *U*, selection criterion *c* is applied.

For the method term list <*close*> of our running example, two candidate sentences are generated using rules MR1 and MR2: S_1 = *"Subjects close object"* and S_2 = *"Subjects get close"*. These two sentences are parsed correctly (with no *U* in the parse trees). The term *close* does not appear as

attribute name in the containing class. Hence, based on the third criterion, sentence S_1 is selected for further analysis.

2.1.3 Syntactic analyzers

As our analyzers, we use two tools, namely *Minipar* which has an integrated PoS tagger and the *Malt* parser, which we employ together with the *SVMTool* PoS tagger. Minipar is used in UIA while Malt/SVMTool are used in UPA, TPA, and TIA. Minipar is quite robust with respect to natural language variability but is available as is, and can not be adapted in any way to new tasks. Since we aim at adapting the analyzer to identifier analysis, we consider the combination of the latter two state-of-the-art tools: Malt parser and SVMTool. Malt parser and SVMTool are based on data-driven NLP approaches. We have applied them both in their standard English version, referred to as *untrained*, and after re-training on a text which is similar to the token sequences generated from identifiers. Details of these tools are presented in the following sub-sections.

Minipar

Minipar[1] is a broad-coverage principle based parser for the English language (see Lin [77]), in which the grammar is represented as a network. It adopts an integrated strategy: a list of possible PoS tags is associated to each word in the lexicon and the resulting tag is chosen during parsing. The lexicon used by Minipar contains 130,000 entries which are composed of the lexicon from *WordNet* [89, 44] and additional proper names. Its frequency and possible PoS tags are associated to each lexicon entry. After parsing a sentence, Minipar outputs information about the individual components of the sentence and the structural relations between such components, including their mutual dependencies. In addition to specifying the

[1] http://webdocs.cs.ualberta.ca/ lindek/minipar.htm

relationship between terms, Minipar labels each term with one of the PoS based on its role in the sentence.

The PoS which are of interest to us, to extract concepts and relations, and to build the ontology, are nouns (N), verbs (V) and adjectives (A). Minipar generates a list of tuples. Each tuple provides information about the term, w, represented by the node, its category (N, V, A, etc.), the head (*root*) term it modifies, and the dependency relationship between the modified term (the head) and w (see Lin [77]). In this study we are mainly interested in the dependency relations between verbs and their respective objects, and the nouns and their modifiers. The former dependency relation is referred to as object relation (*obj*) in Minipar, while the latter as a modifier (*mod*) or noun-noun specifier (*NN*) relation. Figure 2.7 shows a graphical representation of the tuples generated by Minipar for the sentence *Subjects get size*.

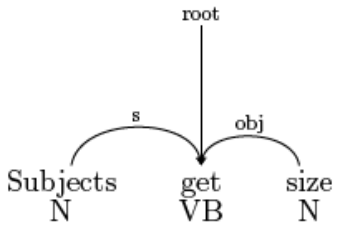

Figure 2.7: Parse tree for ***Subjects get size*** which is generated using SEA (Minipar).

SVMTool PoS tagger

The PoS tagger SVMTool presented in Giménez and Màrquez [49] is based on Support Vector Machines (SVMs) by Vapnik [111], a machine learning approach largely adopted because of its good performance on a large set of tasks. The SVMTool tagger for standard English achieves a very competitive accuracy of 97.2%, as reported by Giménez and Màrquez [48]. The system can also be efficiently trained to be adapted to different languages.

In fact, SVMTool is composed of two modules: SVMLearn and SVMTagger. The former is used to train the models and it is based on SVM^{light}, a library for SVM implemented by Thorsten Joachims [63]. The latter is used to tag the input sentences, and in our case it is applied to tokenized sentences. In the UPA NLP system, the standard English tagger is applied to the token sequence output by the tokenizer, while in the TPA and TIA systems it is applied after training the models on the training set, as discussed in Section 2.1.4.

Malt parser

Malt parser[2] [92, 91] is a data-driven dependency parser, which, similarly to SVMTool, can be applied either with a standard English model or trained on a training set. In this case, however, the training set is represented by a collection of sentences annotated with the corresponding analysis, called *treebank*. While Minipar also includes a PoS tagger, the input to Malt parser must be tagged. The final output of Malt parser is quite similar to that of Minipar.

2.1.4 Training

In NLP tools based on machine learning, a model is learned from a training set and then it is applied to the input. This is the means we employ in this work to adapt generic natural language tools to process the list of tokens extracted from an identifier. The crucial point in this approach, however, is the construction of a training set which can describe the task at hand. Each training set should be collected from a domain as similar as possible to the one considered, and then annotated with the information necessary for the model. In our case, annotation should include both PoS tagging

[2]Malt parser can be freely downloaded from `http://maltparser.org/download.html`

and dependency analysis. We looked for collections of identifiers available together with their analysis (PoS tags and parse trees). The closest data we got is the class identifier data set built by Butler *et al.* [27]. It is a treebank which has been employed for class and attribute names. This data set contains 120,000 class identifiers extracted from 60 Java open source projects. It has been used to understand the Java class identifier naming conventions used in practice [27]. Since the first step of training is identifying grammatical patterns in names based on a part of speech (PoS) tagging, we use the identifiers of this data set, which are already tokenized and tagged using the Stanford Log-linear PoS tagger[3].

As larger training sets are usually better than smaller ones, a good training set is obtained as a trade-off between the need for a large amount of data and the requirement that such data accurately describe the task at hand. Unfortunately, manual annotation is a very expensive process, and therefore it is very difficult to obtain large training sets for tokenized identifiers. Furthermore no large collection of program identifiers annotated with PoS tags and associated with the respective parse trees is publicly available. We have therefore designed an automatic procedure to construct the necessary annotations without manual intervention. We used natural language texts available from the documentation of the considered software projects to build our training sets. Such documentation typically includes comments extracted from the source code, user manuals, system documentation, and FAQs describing *howtos* of the system.

As already mentioned above, while only PoS annotations are needed to train the PoS tagger, for the parsers we need a *treebank*. In a similar way to syntactic parsers, treebanks can also follow the *constituency* or *dependency* framework. The latter suits well ontology construction, as it builds dependency relations between words. On the other hand, the constituency

[3]http://nlp.stanford.edu/software/tagger.shtml

approach produces a sentence parse tree, which is easier to transform to obtain the kind of simplified sentences corresponding to identifiers. We have used both approaches during the construction of the training sets for both the PoS tagger and the dependency parser. In fact, the transformations necessary to build a potential identifier from a natural language sentence are more intuitively expressed in the constituency framework. Eventually, the so obtained constituency treebank has been transformed into an equivalent dependency one, necessary to train Malt parser.

The construction of the training set by means of transformations applied to a natural language treebank has also the advantage of allowing a stronger adaptation to the considered software system. Indeed, it can be automatically applied to any natural language description of the system, such as comments and documentation. While designing the transformations to be applied to these texts in order to simulate identifiers, we have considered the fact that identifiers have different structures depending on their function. For example, method names are more likely to describe actions and therefore their structure resembles VPs, while attribute and class names usually aim at indicating things, in a way similar to NPs. Such distinction is expected to affect the syntactic analysis, but not the PoS tagging. Therefore, a unique training set including all sentences is considered to train the PoS tagger, while two different, disjoint training sets (VP-like sentences for method names and NP-like sentences for class/attribute names) are considered to train two parsers, a VP-parser and an NP-parser.

First of all, the natural language sentences available from the project documentation are PoS tagged using SVMTool with the language model for the standard English distributed with the tagger. Afterwards a constituency parser, namely the Stanford parser discussed later in Section 2.1.4, is employed to build the constituency parse trees of each sentence. Although automatic PoS tagging and parsing can introduce errors, it is very

cheap and the error rates of both tools are low enough to be sure that the introduced errors will not deteriorate too much the resulting treebank. All determiners are then deleted from the treebank, since no determiner is usually included in identifiers. Although a similar transformation could also be applied to other PoS tags, only this one resulted to be effective in some preliminary tests. This is probably due to the fact that the other infrequent PoS tags are nearly absent from the simplified text which we use for training the parser. However, the deletion must be performed in such a way that a consistent parse tree is produced even after the transformation. An important property of parse trees is that only leafs are labeled with PoS tags. Therefore, after deletion, every internal node must still have one or more children.

As noted before, we aim at obtaining two different parsers, namely the VP-parser to apply to method identifiers and the NP-parser for class/attribute identifiers. Therefore, we need two different training sets, one containing only parse trees of VP's and the other of NP's. As identifier structures are usually quite simple, we also impose that all NP's and VP's composing our training treebanks are minimal in the sense that they do not contain any other subtree with the same root. Such trees are called *non-recursive*. Then, all non-recursive VP subtrees are collected from the parsed project documentation to form the VP training set, while all non-recursive NP subtrees form the NP training set. Eventually, the parse trees are converted into equivalent dependency graphs, used to train the data-driven dependency parser.

We use the whole treebank to train SVMTool, while each of the two treebanks is used to train the VP-parser and the NP-parser respectively. The so obtained modules are then introduced in the two trained NLP systems, namely TPA and TIA, as shown in Figures 2.4 and 2.5, respectively.

Stanford parser

The constituency parser used for the construction of the training treebank is the Stanford parser [67, 68] with the English grammar distributed together with the software. It is based on probabilistic context-free grammars whose probabilities are estimated during training and used during parsing to output the most probable derivation with the Viterbi algorithm. This package is implemented in Java and can be freely downloaded from `http://nlp.stanford.edu/software/lex-parser.shtml`. To build the training treebanks for NP-parser and VP-parser, we use another tool distributed by the Stanford lab, namely the Tsurgeon[4], a tree transformation tool which maintains the consistency of parse trees when non-recursive subtrees are extracted.

2.2 NLP based concept extraction

The concepts which are used in building the ontology are derived mainly from the nouns and adjectives found in the term lists. The ontological relations are obtained by mapping the linguistic relations in the dependency tree produced by the analyzers to ontological relations. Additional ontological relations are obtained from verbs.

The resulting ontological relations are:

a. ***isA:*** a relation between a general and more specific concept.

b. ***<verb>:*** a context specific relation between a concept, usually the doer, and the object on which the verb acts.

c. ***hasProperty:*** a relation between a concept and its properties.

d. ***hasState:*** a relation between a concept and its state.

[4]The tool can be freely download from `http://nlp.stanford.edu/software/stanford-tregex-2012-07-09.tgz`

The details of how these ontological relations are extracted by mapping the linguistic relations (dependencies) is presented in the remaining of this section. To graphically demonstrate the mappings we have used the linguistic relations produced by UIA (Minipar).

2.2.1 The *isA* ontological relation

An *isA* ontological relation is mapped to *nn* and *mod* linguistic relations produced by UIA (Minipar) as shown in Figure 2.8. If the analyzer used is UPA, TPA, or TIA (Malt parser), it is mapped to *nn*, *amod* (*adjectival modifier*) or *partmod* (*participial modifier*) natural language dependencies. The *isA* ontological relation sub-tree is obtained by first taking the root noun (the most general concept) which is modified/specified in the sentence parse tree and the descendant (more specialized) sub-concepts are obtained by incrementally adding all specifiers/modifiers down the sub-tree.

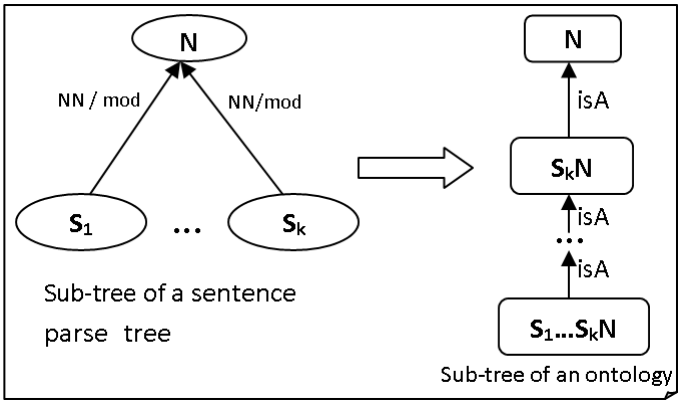

Figure 2.8: Mapping rule for *NN*-specifier or *mod* relation to an *isA* relation, S is a specifier/modifier.

If we take the sentence generated using rule CR3 for the class identifier *currentAccount* of our running example, *"current account is a thing"*, *current* is identified as a *nn*-specifier of *account* (see Figure 2.9). Hence, the ontological relation *isA(current account, account)* is extracted.

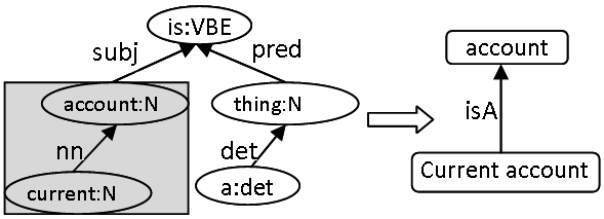

Figure 2.9: Concept and relation extraction from the NLP parse tree of the sentence *"current account is a thing"*. The shaded box shows the terms taken from the class name of the running example and the linguistic relation among them.

2.2.2 The <*verb* > ontological relation

The <*verb* > relation is a context specific relation between the doer concept, usually the class name, and the object (another concept in the relation) on which the verb acts (see Figure 2.10a). The object and relation are identified and extracted from the verb phrase dependency tree of the sentence constructed for method names. The <*verb*> relation is mapped to the verb of the verb phrase while the object is identified by looking at the object of the verb, which is connected to it using the NLP dependency *obj* produced by UIA (Minipar), or the NLP dependency *dobj* (*direct object*) or *pobj* (*preposition object*) produced by UPA, TPA and TIA. For example, from the parse tree of the sentence *Subjects calculate interest* which is constructed for the method term list <*calculate, interest*> created from the method name *calculateInterest* in the class *CurrentAccount*, we can extract the ontological relation *calculate(current account, interest)* (see Figure 2.11).

This ontological relation is not generated if the verb in the sentence is an *accessor* (*get* or *set*). When there is no *obj* NLP dependency in the verb phrase, *e.g.*, due to a method name constructed from only one verb, we use the following steps to identify the possible object of the verb.

i. If the method has one or more formal parameters, we take the type of the first formal parameter as an object, if such a type is a user defined

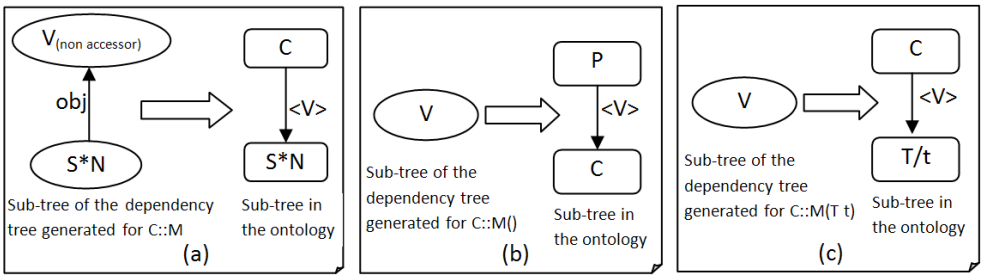

Figure 2.10: Mapping rules for $<verb>$ ontological relation. C is the concept related to the class containing the method M for which the sub-parse tree is shown, P is the concept representing the program and S is a specifier/modifier. S* means zero or more repetitions of S. T is the type of the formal parameter t of method M.

type (see Figure 2.10(c)).

ii. If the method has one or more formal parameters and the type of the first formal parameter is not a user defined type, we take the parameter name as an object (see Figure 2.10(c)).

iii. If the method does not have any formal parameter, the class name is considered as the object. In this case the doer concept is the concept represented by the program name (see Figure 2.10(b)).

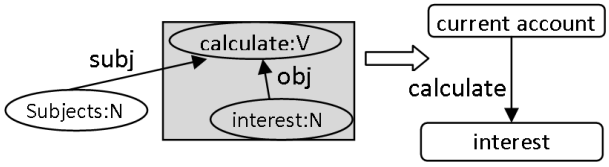

Figure 2.11: The NLP parse tree of the sentence *Subjects calculate interest* and the corresponding ontological concepts and relation extracted. The shaded box shows the terms taken from the method name of the running example and the linguistic relation among them.

2.2.3 The *hasProperty* ontological relation

The *hasProperty* ontological relation is extracted in a similar way as the $<verb>$ relation. This ontological relation, however, is extracted when the

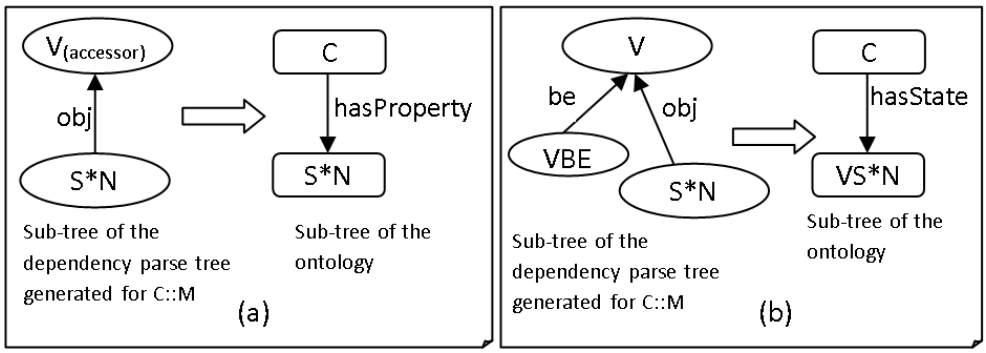

Figure 2.12: Mapping rules for *hasProperty*, (a), and *hasState*, (b), ontological relations. C is the concept related to the class containing the method M for which the sub-parse tree is shown and S is a modifier/specifier. S* means zero or more repetitions of S.

verb in the verb phrase of the sentence is either of the two *access verbs*, *get* or *set*. The concepts involved in this relation are those associated with the class name and the object in the verb phrase which represents the property (see Figure 2.12(a)). An example of such an extraction is shown in Figure 2.13(a) for the statement *Subjects get balance*. The ontological relation extracted from the corresponding parse tree is *hasProperty(current account, balance)*.

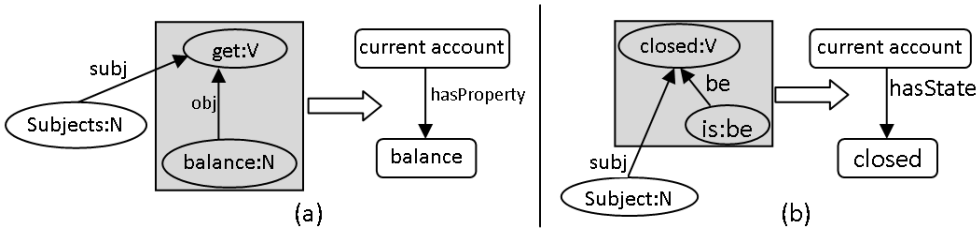

Figure 2.13: NLP parse trees and the corresponding ontological concepts and relations extracted for the sentences *Subjects get balance*, (a), and *Subject is closed*, (b).

2.2.4 The *hasState* ontological relation

The *hasState* ontological relation is a relation generated from a parse tree of a sentence when the *verb to be* is found in both the parse tree and the method term list from which the sentence is constructed. The predicate

verb and the corresponding object (when available) are used as concepts that represent the state of the concept associated with the class containing the method (see Figure 2.12(b)). Figure 2.13(b) shows the ontological relation and concepts extracted, *hasState(current account, closed)*, for the sentence *"Subject is closed"* which is generated for the method term list $<is, closed>$.

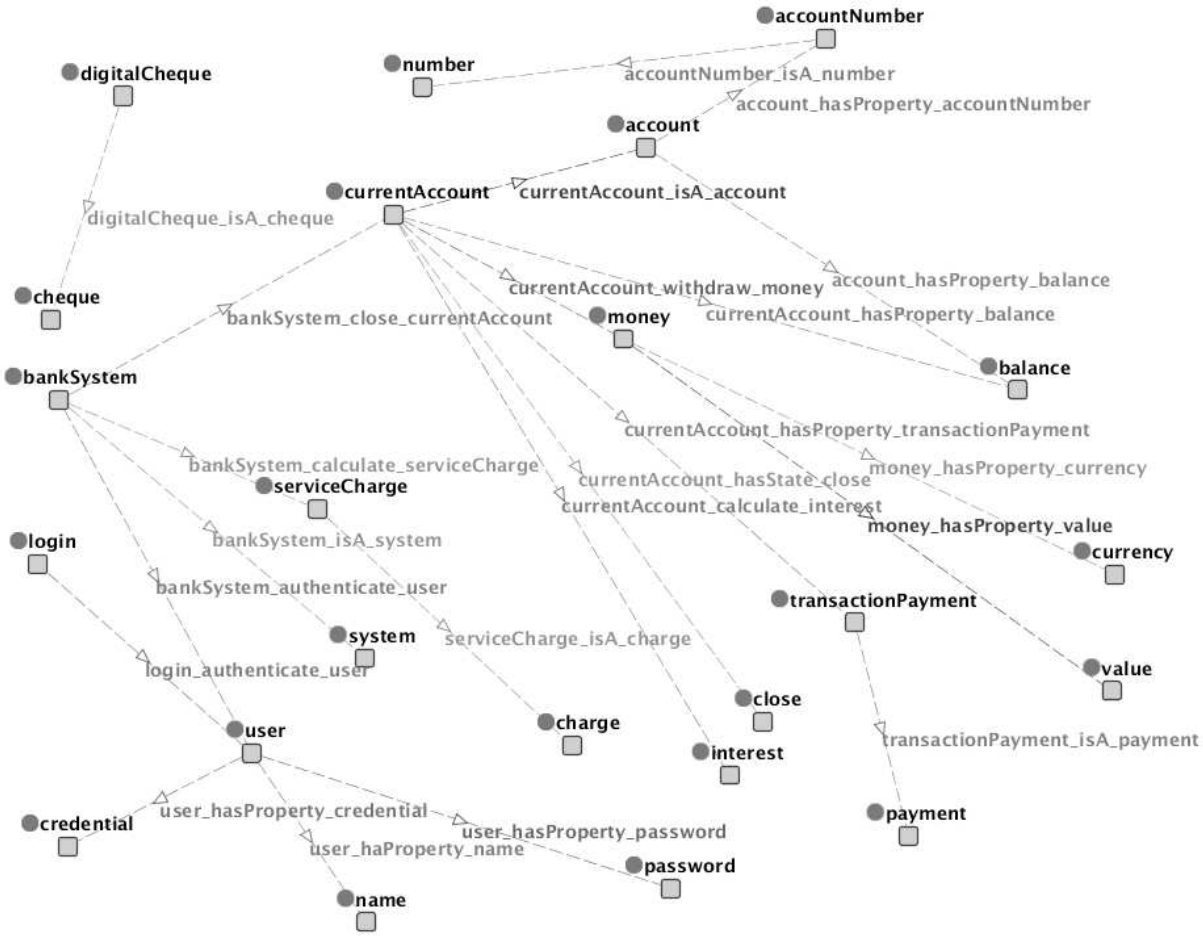

Figure 2.14: An ontology extracted for the running example code fragment shown in Figure 2.6 using the NLP based approach.

An example ontology

Figure 2.14 depicts the ontology [5] generated for the code fragment shown in Figure 2.6, following the steps described above. We have concepts connected using the *isA* relation such as *isA(service charge, charge)*, *isA(digital cheque, cheque)*, and *isA(current account, account)* where one of the three is also captured in the source code syntax (`class CurrentAccount extends class Account`).

In addition, the ontology provides information about the properties and actions of a concept. For example, the concept *current account*, has *balance* as its property and *calculates interest* as an action it can perform. The ontology also captures states of a concept through its *hasState* relation (*e.g.*, *hasState(current account, closed)*).

2.3 Structural based concept extraction

Programmers who use the object oriented paradigm to write their code follow certain structural rules and design guidelines supported by the paradigm. This, in turn, influences the way they think and model their knowledge of the solution during the implementation. In this section, we describe how we use the rules and structure associated with the object oriented paradigm to extract an ontology which models the knowledge captured in the source code structure. The summary of the mapping rules from the structure of the source code to the ontology concepts and relations is presented in Table 2.2 for the Java programming language. Similar rules can be easily defined for other object oriented languages (*e.g.*, C++).

The first syntactic relation listed in Table 2.2 is *extends*. *Extends* is used to introduce an inheritance relationship between two classes. The corresponding ontological relation which captures this characteristic is *isA* (S1).

[5] Protégé (http://protege.stanford.edu/) was used to visualize the ontology.

Table 2.2: Rules for extracting structural ontology from object oriented (Java) source code. (C=class, M=method, A=attribute, T=type, I=interface, p=formal parameter, UDC= user defined class)

Rule	**Structural**			**Ontological**	
	Source	**Relation**	**Target**	**Concepts**	**Relation**
S1	C_1	extends	C_2	C_1, C_2	isA(C_1, C_2)
S2	C_1	implements	I	C_1, I	isA(C_1, I)
S3	C_1	has attribute	$A : T$, $T \neq$ boolean	C_1, A	hasProperty(C_1, A)
S4	C_1	has attribute	$A : T$, $T =$ boolean	C_1, A	hasState(C_1, A)
S5	$C_1 :: M_1$	Calls	$C_2 :: M_2()$	C_1, C_2	$<M_2>(C_1, C_2)$
S6	$C_1 :: M_1$	Calls	$C_2 :: M_2(Tp, ...)$	C_1, T, p	$<M_2>(C_1, T)$, if $T \epsilon UDC$ $<M_2>(C_1, p)$, otherwise
S7	C_1	has get/set method	M_2	C_1, m_2; m_2 is M_2 without get/set prefix	hasProperty(C_1, m_2)

In some programming languages, such as Java, multiple inheritance is not allowed. However, Java provides multiple sub-typing using the *implements* construct. We map such a relation to the *isA* ontological relation as shown in *Rule S2*. In the ontological relation, the class which is extended or the interface which is implemented is taken as the general concept, while the class which extends or implements it is the more specific concept. In our running example, shown in Figure 2.6, there is an *extends* relation between the child class *CurrentAccount* and the parent class *Account*, and there is an *implements* relation between the class *CurrentAccount* and the interface *DigitalCheque*. These two syntactic relations are used to extract the *isA(current account, account)* and *isA(current account, digital cheque)* ontological relations, respectively.

Attributes are used to represent properties and states of a class. As shown in *Rules S3* and *S4*, we use the *hasProperty* and *hasState* ontological

relations to capture them in the ontology. The concepts involved in this relation are the attribute and the class containing the attribute. If the attribute is of type boolean, the ontological relation used is *hasState* while when it is not, *hasProperty* is used. For example, following *Rule S3* the ontological relation *hasProperty(current account, balance)* is extracted for our running example from class *CurrentAccount* and its attribute *balance* (see Figure 2.6). Properties of a class which are the result of a computation are usually represented using accessor methods. We capture also such properties using the *hasProperty* ontological relation (S7). In such cases, the concepts involved in the relation are the class containing the accessor method and the method name without the prefix *get* or *set*. For example, if we have a class *Rectangle* with a method *getArea*, we extract the ontological relation *hasProperty(rectangle, area)* using *Rule S7*.

Methods are used as a means of communication and interaction with other classes. Such interaction is captured in our ontology through the method name involved in the call relation between two classes (see *Rules S5* and *S6*). When the called method does not take any argument, we create a relation between the concepts represented by the caller class and the type of the object on which the method is invoked (*Rule S5*). The relation connecting the two concepts is a context specific relation represented by the called method name. In cases where the called method has a parameter, we create a relation between the concepts associated with the caller class and the type of the first parameter, if it is a user defined type. Otherwise the relation will be between the concept representing the caller class and the name of the first parameter (*Rule S6*). In both cases the relation used to connect the concepts is a context specific relation represented by the called method name, like in *Rule S5*. For example, from the method call *calculate()* on object *sc* in method *transactionPayment* of our running example, we can extract the ontological relation *calculate(current account,*

service charge) using *Rule S5*. From the method call *withdraw(...)* on the object *ca*, the ontological relation *withdraw(bank system, money)* is obtained following *Rule S6*.

Object oriented languages support polymorphic calls. As a consequence, the declared type of the target object involved in a method call may be different from its actual type (as determined at object creation time). This might result in some degree of imprecision, when extracting C_2 according to *Rules S5* and *S6*, which only consider the declared type of the object on which a method is invoked or the declared type of the first method parameter. The ontology extractor might be improved by resorting to points-to analysis [88] or to the object flow graph [108].

An example ontology

By applying the rules described above to the code fragment shown in Figure 2.6, we obtain the ontology shown in Figure 2.15. The ontology shows the concepts and relations among them, as captured by the structural relations in the code. For example, the inheritance relationship shown between *currentAccount* and *account* is also present in the ontology, where it is represented as an *isA* relation. The ontology captures also the attributes of each class through the *hasProperty* relation (*e.g.*, *hasProperty(currentAccount, balance)*). Context specific relations, such as *close(bankSystem, currentAccount)*, capture the communication relations appearing in the code as method calls.

2.4 Concept location

Concept location is part of a program comprehension activity where the programmer searches the source code to identify a specific part which implements a given concept [100, 99]. During software evolution, it is used

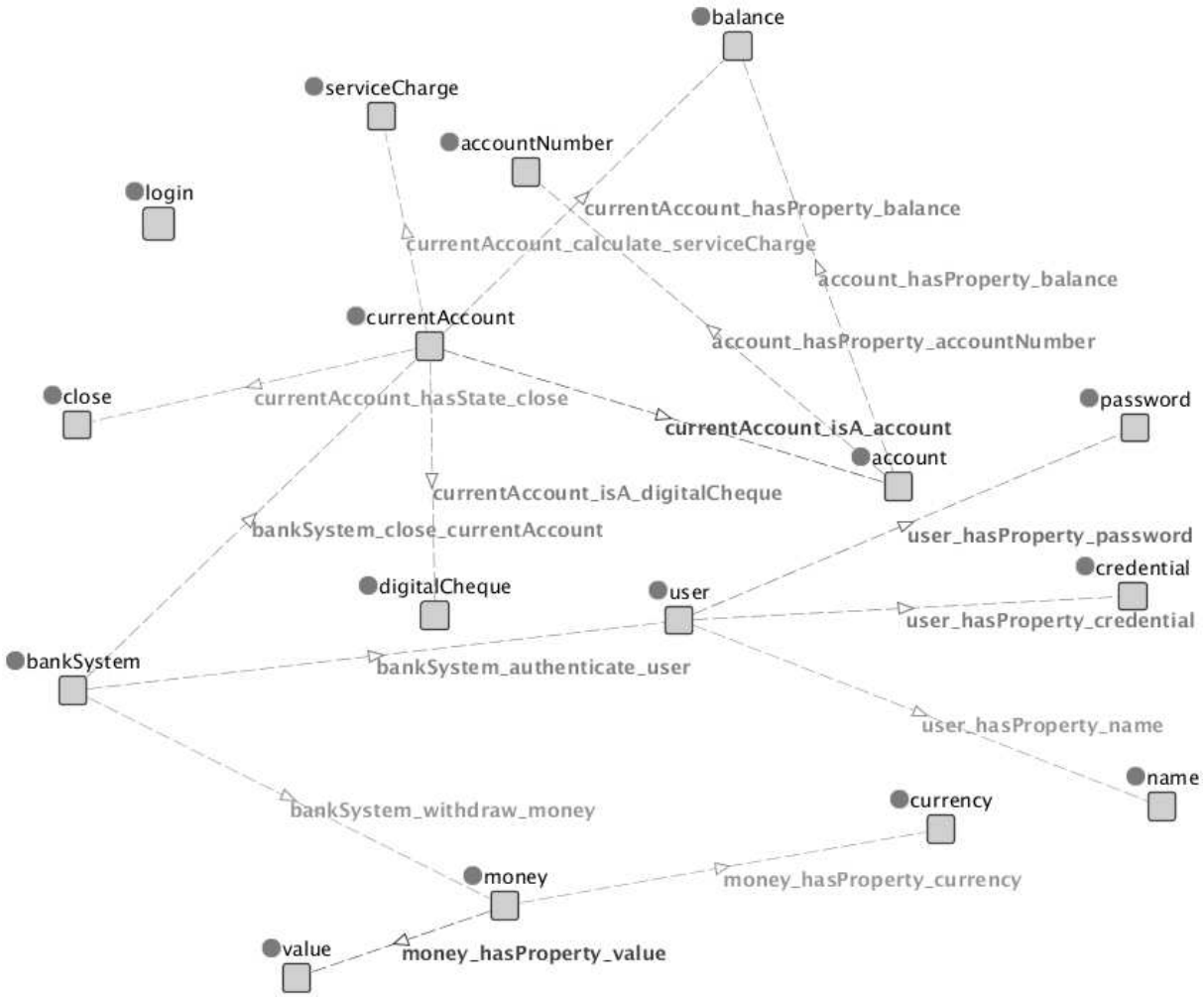

Figure 2.15: An ontology extracted for the running example code fragment shown in Figure 2.6 using the structural approach

to identify the location where a change is to start in response to a change request, such as, a bug report or a new feature request. It involves formulating a query composed of one or more keywords which a programmer thinks are related or refer to the concept to be searched. While formulating a query, the programmer resorts to her prior knowledge, as well as any information associated with the concept to be searched.

After querying the code base with the initially formulated query, the programmer will analyze the returned results. If she is not satisfied with

the result, based on the newly acquired knowledge, she may decide to reformulate the query or to further filter the results (see Petrenko *et al.* [94] and Hill *et al.* [58]). Successive filtering and reformulation of the query continues until the programmer is satisfied with the result.

To carry out a concept location task, the developer can employ various approaches which exploit textual and dynamic information. In this section, we discuss two text based approaches which are used in our studies: information retrieval (IR) and regular expression matching.

2.4.1 Information retrieval

IR-based approaches treat the source code as a document corpus and use methods such as latent semantic indexing (LSI) to index the corpus [83, 97, 95, 47]. A document corpus which corresponds to the source code is created by extracting the identifiers and comments at a developer-defined granularity level (classes, methods, etc.). In the corpus, one document is mapped to a code entity at the chosen granularity level. While creating the corpus, identifiers are split to their composing terms. For example, *userName* is split to *user* and *name*. In some cases, common English words are also filtered from the corpus.

The document corpus is then transformed to the corresponding mathematical representation and indexed using techniques such as LSI and Lucene[6]. To transform the corpus to the corresponding mathematical representation, scoring methods such as term frequency and term frequency inverse document frequency (TF-IDF) are used. The indexed corpus is then searched using queries formulated by developers.

In response to a query, IR-based approaches return a ranked list of documents which correspond to the entities at the selected granularity level. The rank to each document is given based on the similarity of the doc-

[6]http://lucene.apache.org/

ument to the query. Similarity between the query and the documents is computed using similarity measures such as *cosine similarity*. The documents ranked close to the top are those relatively similar to the query, and they are considered relevant to the query.

2.4.2 Regular expression matching

In regular expression matching the query formulated by the developers is directly matched against the content of the files in the code base. The matching is conducted using tools such as `grep`[7]. The result of the query, in this case, is a set of files which contain all the terms (keywords) in the query.

2.5 Evaluation

2.5.1 Comparison of natural language analyzers

In Section 2.1, we have presented four types of analyzers which can be used to syntactically analyze, and extract the information captured in the identifiers to build ontologies. In this section, we asses the impact of using different analyzers to generate ontologies. The assessment is conducted using a case study in the context of a program understanding task, namely concept location.

One of the applications in which concept location is widely used is bug fixing. When users of a program encounter a problem, they communicate it to the developers of the program by filing a bug report. The bug report contains several data, among which a title, a bug description and (optionally) a set of keywords. It is reasonable to assume that developers will use this information to formulate a query, used to retrieve files which are

[7]http://www.gnu.org/software/grep/

relevant for the bug to be fixed. In our study, we call such queries *basic queries*.

In our previous work [4], we have proposed to enhance basic queries using concepts taken from the ontology extracted from the corresponding source code. The enhancement of the queries is carried out by expanding the set of keywords used for formulating the basic queries with concept names taken from the ontology. The concept names are selected by first matching each keyword to the concepts in the ontology and taking the neighboring concepts of the matched concept. A match is found when the name of a concept is the same as the keyword. A neighboring concept of a given concept is any concept that is exactly one edge away from the matched concept, where by edge we mean any ontological relation (*isA*, *hasProperty*, etc.). For example, given the keyword *balance* and the ontology shown in Figure 2.14, the neighboring concepts *current account* and *account* will be considered as additional keywords to be used in formulating the query. In the following, we refer to queries formulated in this way as *enhanced queries*. In our approach, the *enhanced queries* can be formulated from the ontologies built using the parse trees of either UIA, UPA, TPA, or TIA. We call the enhanced queries formulated using concepts taken from the ontology built using parse trees of UIA/UPA/TPA/TIA respectively as *UIA/UPA/TPA/TIA enhanced queries*.

The relation between concepts in the ontology are derived from the natural language dependencies. Since these often represent a semantic relation between the terms they connect, we argue that the concepts connected in the ontology are also closely related. Consequently we conjecture that the expansion of the query with the additional closely related concepts could potentially improve the quality of the query, which may have a positive impact on concept location.

Research questions

In this case study, we address the following research questions.

- ***RQ1. Query effectiveness:*** *Do the extracted ontology concepts contribute to increasing the effectiveness of basic queries formulated for concept location?*
- ***RQ2. Ontology comparison:*** *Do the ontologies produced by different analyzers differ between each other?*
- ***RQ3. Analyzer impact:*** *Does the choice of the analyzer impact the effectiveness of concept location?*

The effort programmers have to put in a concept location activity depends on the effectiveness of the queries they formulate. If a query is effective, it will either rank a relevant document at the top of the ranked list of documents (if IR-based approach is used) or it will find all relevant files (if regular expression matching is used). Hence, a good query can reduce the effort and time developers have to put in the task for which the query is formulated. The first research question, RQ1, compares the basic and enhanced queries in terms of their effectiveness, and investigates the following null/alternative hypotheses:

> $\mathbf{H_{0-RQ1}}$: There is no statistically significant difference between the effectiveness of basic queries and the effectiveness of UIA, UPA, TPA, or TIA enhanced queries formulated by expert or average programmers while using either grep-based or LSI-based approach.
>
> $\mathbf{H_{1-RQ1}}$: There is statistically significant difference between the effectiveness of basic queries and the effectiveness of UIA, UPA, TPA, or TIA enhanced queries formulated by expert or average programmers while using either grep-based or LSI-based approach.

We can measure the effectiveness of a query in two ways depending on the approach used to query the source code: reciprocal rank, for IR-based approaches, and precision and completeness, for regular expression matching. *Reciprocal Rank* of a query is computed as *1/rank of the top relevant document*. A query evaluated using reciprocal rank is considered effective if the computed metric value is close to 1 (*i.e.*, if the relevant document is ranked close to the top). The *Mean Reciprocal Rank (MRR)* is computed as the average of the top relevant document reciprocal rank over all bugs considered for a system.

A query is *precise* if it reduces the amount of effort required by a developer to identify the relevant source code entities, among those reported when the query is executed. It can be measured by computing the *Precision (P)* of the query. Precision is defined as the ratio of *number of relevant source code files retrieved* to *total number of source code files retrieved*. If precision is low, a developer has to inspect many files to identify those which are actually relevant for the task at hand.

A query is *complete* if it identifies all the source code files which are relevant for the task at hand. It is measured using *Recall (R)* which is defined as the ratio of *number of relevant source code files retrieved* to *total number of relevant source code files*. A recall value of 1 indicates that all of the relevant files (*e.g.*, to be modified to address a given change request) are retrieved, while a value of 0 indicates that none of the relevant files are retrieved by the query. To combine the two inversely related measures (precision and recall) and simplify the comparison on query effectiveness, we use the *F-measure (F)*. F-measure (F) is computed as the harmonic mean of precision and recall ($F = (2 * P * R)/(P + R)$). High F-measure indicates that the query is effective.

A query can be formulated by programmers with different expertise. In our study, we simulate the query formulation activity of programmers

with two levels of expertise: expert programmer and average programmer. To carry out the simulation, we assume that an expert programmer will formulate the queries which give the highest reciprocal rank or the highest F-measure and that an average programmer will formulate queries which give the median reciprocal rank or the median F-measure, among all possible queries. Here after we refer to the queries which give the highest reciprocal rank or the highest F-measure as *best queries* while those queries which give the median reciprocal rank or the median F-measure are referred to as *average queries*.

To test the null hypothesis, we have applied the two-sided, pair-wise Wilcoxon signed-rank test between basic queries, and UIA, UPA, TPA, and TIA enhanced queries, considering reciprocal highest and median ranks, and highest and median F-measures. The computed pair-wise tests are summarized in Table 2.3 (top).

Table 2.3: Summary of pairs used to answer RQ1 and RQ3 hypotheses. The analyzer names used in the table correspond to the enhanced queries formulated using the ontology built from the respective analyzer.

Search approach	**LSI-based**		**Grep-based**	
Query	**Best**	**Average**	**Best**	**Average**
H_{0-RQ1}	Basic vs. UIA	Basic vs. UIA	Basic vs. UIA	Basic vs. UIA
	Basic vs. UPA	Basic vs. UPA	Basic vs. UPA	Basic vs. UPA
	Basic vs. TPA	Basic vs. TPA	Basic vs. TPA	Basic vs. TPA
	Basic vs. TIA	Basic vs. TIA	Basic vs. TIA	Basic vs. TIA
H_{0-RQ3}	UIA vs. UPA	UIA vs. UPA	UIA vs. UPA	UIA vs. UPA
	UIA vs. TPA	UIA vs. TPA	UIA vs. TPA	UIA vs. TPA
	UIA vs. TIA	UIA vs. TIA	UIA vs. TIA	UIA vs. TIA
	UPA vs. TPA	UPA vs. TPA	UPA vs. TPA	UPA vs. TPA
	UPA vs. TIA	UPA vs. TIA	UPA vs. TIA	UPA vs. TIA
	TPA vs. TIA	TPA vs. TIA	TPA vs. TIA	TPA vs. TIA

The concepts used in the enhanced queries are retrieved from different ontologies which are generated using different analyzers. In RQ2, we compare four ontologies, generated using the analyzers UIA, UPA, TPA and TIA. To compare the ontologies, we compute the Jaccard index ($|A \cap B|/|A \cup B|$) between them, to see how similar they are, and the ratio of unique concepts each ontology has to their union ($|A \setminus B|/|A \cup B|$, $|B \setminus A|/|A \cup B|$).

The enhanced queries use concepts taken from different ontologies which are based on different parse trees of identifiers. The different parse trees are produced using UIA, UPA, TPA and TIA. In our last research question, RQ3, we investigate whether the choice of the analyzer impacts the effectiveness of the enhanced queries in concept location and we test if the impact is statistically significant. The investigation is conducted by comparing the effectiveness of the enhanced queries formulated using concepts taken from the ontologies produced using the outputs of different analyzers. To carry out the test, we have formulated the following null/alternative hypotheses:

> $\mathbf{H_{0-RQ3}}$: There is no statistically significant difference between the effectiveness of UIA, UPA, TPA, and TIA enhanced queries formulated by expert or average programmers while using either grep-based or LSI-based approach.
>
> $\mathbf{H_{1-RQ3}}$: There is a statistically significant difference between the effectiveness of UIA, UPA, TPA, and TIA enhanced queries formulated by expert or average programmers while using either grep-based or LSI-based approach.

To test the hypotheses, we have conducted a two-sided, pair-wise Wilcoxon signed-rank test. The pair-wise tests computed to test these hypotheses are summarized in Table 2.3 (bottom).

In our study, we have conducted multiple tests on the hypotheses formulated for two of our research questions (see Table 2.3). To control the false discovery rate and correct for multiple comparison, we have adjusted the p-values using the Benjamini and Hochberg (BH) [15] correction.

Procedure

Our case study has three main steps, *identifier parsing*, *ontology extraction* and *concept location*. Next, we describe each step in detail.

Identifier parsing

Before parsing identifier names using the analyzers, they have to be tokenized. If the identifiers are composed of more than one term, we regard camel casing and underscore as separators and use them to split identifier names into their composing tokens. Before splitting an identifier into its composing tokens, prefixes, such as those associated with the Hungarian notation (*e.g.*, $m\_$ for data members and C for class names) are removed.

Sometimes an identifier's token is not a word. In such cases we consult a predefined list of "known abbreviations and contractions" to identify a possible expansion for the token. If the token is not in the predefined list, we use the longest common sub-string (LCS) technique to find the most similar expanded form for the token. According to this technique, an available dictionary of words is accessed to find the most similar word (*i.e.*, the one with largest LCS with the given token). For example, the token "remot" is replaced by "remote" after applying the LCS algorithm. The expansion of a token to its respective word can also be carried out using the techniques described in existing works on the topic [70, 73, 57, 71, 36]. In the following, we assume that the tokenization step has produced a sequence of valid words.

To automate the tokenization step, we have developed a tool which

automatically collects and produces a tokenized list of class, attribute and method identifiers, following the procedure described above. In addition to this, the tool produces a *fact file*, which is used for further processing in the next step (see Section 2.5.1). The fact file contains information about all classes in the system and their members. The inputs to our tool are XML representations of the source code files, produced by the *src2srcml* tool [35] and a configuration file which contains options and file path information for the files containing the list of "known abbreviations and contractions" and identifier naming conventions (*e.g.*, Hungarian notation in use).

The tokenized list of class, attribute and method names are passed as an input to three types of syntactic analyzers, namely UPA, TPA and TIA (SVMTool/MaltParser). The trained analyzers are obtained following the approach presented in Section 2.1.4. Furthermore the sentences constructed using the same tokenized list is the input of UIA (Minipar) and they are constructed using the steps described in Section 2.1.2. The output of the four analyzers is a set of dependency parse trees which are used as an input of the following step.

Ontology extraction

In this step, we build four types of ontologies using the information captured in the parse trees generated by UIA, UPA, TPA and TIA. To build the ontologies from the parse trees, we have used the natural language dependency to ontological relation mappings described in Section 2.2. For some of the mappings, such as for the *hasProperty* ontological relation, we have to map the identifier containing the ontological relation to the source code where the identifier is defined (*e.g.*, to determine the containing class). To automate this step and make it work with all analyzers, we have modified the tool developed in a previous study [4].

To generate ontologies, our tool takes the parse trees produced for all

identifiers and the fact file generated in a previous step (see Section 2.5.1) as an input. The fact file is used to identify the containing classes of a given identifier which is reconstructed from its parse tree. The containing classes are required to create some ontological relations, as described above. For each set of parse trees generated by UIA, UPA, TPA and TIA, our tool produces four ontologies, which are named after the corresponding analyzers: *UIA ontology*, *UPA ontology*, *TPA ontology* and *TIA ontology*. The concepts in the ontologies are stemmed using *Porter stemmer*[8] to avoid different representations of a concept due to inflections of the corresponding word.

Concept location

To carry out the concept location task, one of the researchers involved in the study has played the role of the programmer and has manually collected keywords from each bug title. Bug titles usually serve as the summary for the problem described in the corresponding bug report and hence are good sources of keywords. To avoid any bias, the collection of keywords was conducted prior to computing any results. These keywords are used in formulating the basic queries and selecting the concepts to be added when formulating the enhanced queries.

To query the source code, we have used two different approaches described in Section 2.4: information retrieval (IR) and regular expression matching. The IR-based approach uses latent semantic indexing (LSI) to index the document corpus [83]. A document in the corpus corresponds to a class in the code and is composed of class, method, and attribute name terms. We have used underscore and camel case to split the names into their composing terms. LSI takes as input the number of dimensions (k-value) to which the vector space should be reduced during the Singular

[8]http://tartarus.org/ martin/PorterStemmer/

Value Decomposition (SVD) and the weight to be used when scoring input documents. For our study, we have conducted a preliminary study using different k-values, and based on the results we have selected k-value to be 10. To score documents we have used term frequency as the weight. In our previous studies [1, 5], we have observed that term frequency is associated with core domain concepts. To rank the documents in the indexed corpus, similarity between the query and every document in the indexed corpus is computed. If the result of the similarity measure is high, the document is ranked closer to the top. To compute the similarity between the query and the documents, we have used *cosine similarity*. Cosine similarity is the most widely used measure while dealing with vector-based representation of documents.

For the second approach, we have used the widely used, yet simple method `grep`. `Grep` performs a pattern matching of the query against the content of the files in the code base and returns all the files which contain all keywords in the query. Hereafter we refer to this approach as *grep*-based while we refer to the former, IR-based, approach as *LSI*-based approach.

Developers are supposed to analyze the results of the LSI-based approach sequentially starting from those documents having the highest similarities to the initially formulated query. After analyzing each source code document, they decide if it is relevant to the task at hand or not. If it is found relevant, the search succeeds. Otherwise, they move to the next top ranked document, or reformulate the query by adding or removing keywords and recompute the rankings of the documents. As we do not involve developers in our study, we simulate their activity using a tool. The tool simulates the query (re-)formulation by considering all combinations of one or two keywords (for basic query), and keywords and concepts (for enhanced query). After querying the indexed document corpus with each query, the tool determines the median/highest reciprocal rank for the

relevant documents and returns the related query.

When developers use the grep-based approach to query the code base, they usually analyze the returned result and if it is found unsuccessful, they may decide to either reformulate the query by choosing another keyword or further filter the results. Successive filtering of the query continues until the developer is satisfied with the result. In our study, we have developed a tool to automatically simulate the developers' query filtering activity. The tool simulates this activity by considering four or less possible combinations of keywords (for basic query), and keywords and concepts (for enhanced query), to formulate a compound query, *i.e.*, a query possibly consisting of at least one and at most four keywords. As we are resorting to an automated simulation, it is not possible to identify a keyword a developer would initially select and reformulate. Hence, our tool considers all possible combinations of keywords of maximum length four. The tool returns the query with the median/highest F-measure. If the selected query contains two or more keywords, we interpret it as the developer applying one or more filters after the initial query.

Subjects

In our case study, we considered three medium size open source systems, FileZilla client[9], JEdit[10] and WinMerge[11]. FileZilla client is a GUI based FTP client which is mainly used to upload and download files from an FTP server. WinMerge is a merging and differencing utility for Windows, while JEdit is a cross platform text editor mainly developed for programmers. FileZilla and WinMerge are written in C++ while JEdit is written in Java. The summary of the three systems is shown in Table 2.4 (LOT means Lines Of Text).

[9] http://filezilla-project.org/
[10] http://jedit.org/
[11] http://winmerge.org/

All systems have a bug tracking system from which we collected closed bug reports with patch files (see Table 2.4). From the patches we have collected the names of the classes and files which are actually modified to fix the bugs. These classes and files are used as our reference to compute reciprocal rank, precision, and recall (*i.e.*, these are the correct program entities to be retrieved by means of both basic and enhanced queries).

Table 2.4: Summary of systems (LOT = Lines of Text).

	Systems			
	FileZilla	**JEdit**	**WinMerge**	
Version	3.0.0	4.2	2.12.2	2.11.1.8
No. of Classes	208	639	146	145
No. of Files	264	224	257	255
No. of LOT	89080	79198	67643	67327
No. of Bugs	28	12	20	

Results

RQ1. Query effectiveness

The results of LSI and Grep are shown in Tables 2.5, 2.6 for best and average queries respectively. In all cases the enhanced queries are found to be more effective than the basic queries irrespective of the approach followed. If we average over all types of enhanced queries used with the LSI-based approach, the reciprocal rank of enhanced queries has improved over the basic queries by 26% in FileZilla, 10% in JEdit, and 29% in WinMerge. A similar analysis on the F-measures of the grep-base approach result shows that the F-measure of basic queries is increased by 127% in FileZilla, 50% in JEdit and 106% in WinMerge.

To investigate if the observed differences are statistically significant, we have formulated the hypothesis stated in H_{0-RQ1} and applied two-sided,

Table 2.5: ***RQ1: Basic vs. enhanced queries; best queries.*** Top rank MRR and average best F-measures of enhanced queries with the corresponding delta percentages over basic queries top rank MRR and average best F-measures. The p-values are adjusted for multiple tests.

	Enhanced query		FileZilla	JEdit	WinMerge
LSI-based	**MRR**	**UIA** (Δ%)	0.568 (86.18)	0.529 (23.02)	0.78 (130.62)
		UPA (Δ%)	0.557 (82.58)	0.529 (23.03)	0.597 (76.38)
		TPA (Δ%)	0.548 (79.52)	0.541 (25.89)	0.623 (84.15)
		TIA (Δ%)	0.569 (86.38)	0.529 (23.03)	0.53 (56.53)
	P**-values**	**UIA**	***0.006***	0.483	***0.009***
		UPA	***0.008***	0.309	***0.024***
		TPA	***0.003***	0.309	***0.024***
		TIA	***0.003***	0.309	***0.021***
Grep-based	**Precision**	**UIA** (Δ%)	0.560 (156)	0.638 (59.7)	0.623 (156)
		UPA (Δ%)	0.576 (163)	0.596 (49.1)	0.634 (161)
		TPA (Δ%)	0.563 (157)	0.668 (67.3)	0.634 (161)
		TIA (Δ%)	0.598 (173)	0.598 (49.7)	0.631 (159)
	Recall	**UIA** (Δ%)	0.878 (4.18)	0.958 (1.47)	0.952 (7.73)
		UPA (Δ%)	0.905 (7.36)	0.958 (1.47)	0.927 (4.91)
		TPA (Δ%)	0.887 (5.24)	0.958 (1.47)	0.927 (4.91)
		TIA (Δ%)	0.878 (4.18)	1.000 (5.89)	0.927 (4.91)
	F-measure	**UIA** (Δ%)	0.600 (124)	0.677 (48.6)	0.668 (109)
		UPA (Δ%)	0.620 (131)	0.656 (43.9)	0.656 (106)
		TPA (Δ%)	0.602 (124)	0.721 (58.2)	0.656 (106)
		TIA (Δ%)	0.614 (129)	0.674 (47.8)	0.651 (104)
	P**-values (F)**	**UIA**	***0.002***	0.309	***0.009***
		UPA	***0.003***	0.309	***0.009***
		TPA	***0.002***	0.309	***0.009***
		TIA	***0.002***	0.309	***0.009***

pair-wise Wilcoxon signed-rank test between basic queries and enhanced (UIA, UPA, TPA, and TIA) queries reciprocal ranks and F-measures of all bugs considered for each system (see Table 2.3, H_{0-RQ1}, LSI-based and grep-based approaches, best columns). The results are shown in Table 2.5.

The p-values of the two-sided, pair-wise signed Wilcoxon test are significant (at $\alpha = 0.05$) for all types of enhanced queries used in FileZilla and WinMerge, irrespective of the approach used to query the code base. For JEdit, none of the results are statistically significant. From these results, we can reject the null hypothesis for the two systems and hence conclude that for them there is a statistically significant difference between the effectiveness of best basic queries and best enhanced queries.

The results for the LSI-based approach (see in Table 2.6) show that the effectiveness of average enhanced queries are worse than the corresponding average basic queries. For the grep-based approach, however, the average enhanced queries are found to be more effective than the average basic queries. The average of the F-measure delta percentage improvement over all types of average enhanced queries is 64% in FileZilla, 31% in JEdit and 67% in WinMerge.

To assess if the difference observed between the two types of average queries is statistically significant, we have defined the hypothesis stated in H_{0-RQ1} and conducted a two-sided, pair-wise Wilcoxon signed-rank test (see Table 2.3, H_{0-RQ1}, LSI-based and grep-based approaches, average columns). The results are shown in Table 2.6. The results obtained for the LSI-based approach show that the difference observed is not significant for all systems. For the grep-based approach, however, the difference is statistically significant for all cases of FileZilla and two cases (UPA, TPA) of WinMerge, while for JEdit the difference is not statistically significant in all cases. Hence, we can reject the null hypothesis in half of the cases for the grep based approach, while for the LSI-based approach we cannot

Table 2.6: ***RQ1: Basic vs. enhanced queries; average queries.*** Median rank MRR and average median F-measures of enhanced queries with the corresponding delta percentages over basic queries median rank MRR and average median F-measures. The p-values are adjusted for multiple tests.

	Enhanced query		FileZilla	JEdit	WinMerge
LSI-based	**MRR**	**UIA** (Δ%)	0.116 (-32.95)	0.047 (-53.3)	0.091 (-40.46)
		UPA (Δ%)	0.142 (-17.92)	0.048 (-52.3)	0.099 (-34.74)
		TPA (Δ%)	0.162 (-6.36)	0.048 (-51.9)	0.099 (-34.87)
		TIA (Δ%)	0.122 (-29.48)	0.049 (-50.6)	0.091 (-40.39)
	P-value	**UIA**	0.495	0.309	0.279
		UPA	0.989	0.309	0.279
		TPA	0.989	0.309	0.278
		TIA	0.989	0.309	0.341
Grep-based	**Precision**	**UIA** (Δ%)	0.323 (48.8)	0.408 (34.4)	0.318 (81.3)
		UPA (Δ%)	0.379 (74.6)	0.422 (39.2)	0.393 (124.0)
		TPA (Δ%)	0.372 (71.4)	0.377 (24.3)	0.367 (109.0)
		TIA (Δ%)	0.404 (86.0)	0.403 (32.6)	0.330 (87.9)
	Recall	**UIA** (Δ%)	0.915 (-1.89)	0.861 (0)	0.772 (-6.65)
		UPA (Δ%)	0.910 (-2.35)	0.861 (0)	0.780 (-5.64)
		TPA (Δ%)	0.885 (-5.11)	0.861 (0)	0.780 (-5.64)
		TIA (Δ%)	0.915 (-1.89)	0.861 (0)	0.763 (-7.66)
	F-measure	**UIA** (Δ%)	0.383 (47.7)	0.474 (33.3)	0.354 (57.7)
		UPA (Δ%)	0.444 (71.6)	0.456 (28.2)	0.400 (77.9)
		TPA (Δ%)	0.424 (63.9)	0.463 (30.2)	0.392 (74.2)
		TIA (Δ%)	0.447 (72.5)	0.468 (31.4)	0.359 (59.8)
	P-Value (F)	**UIA**	***0.025***	0.309	0.111
		UPA	***0.011***	0.309	***0.021***
		TPA	***0.008***	0.691	***0.021***
		TIA	***0.006***	0.483	0.111

reject the null hypothesis in any case.

From the results obtained for both best and average queries, we can conclude that using the ontology concepts extracted based on NLP for *grep-based* concept location has increased the effectiveness of queries. The observed improvement is statistically significant in the majority of the cases. Hence, for the grep-based approach, we can answer RQ1 positively for both best and average queries. For *LSI-based* approach, however, the extracted ontology concepts did not improve average queries and hence we can answer RQ1 positively only for the best queries, *i.e.*, when expert developers are involved.

We performed a qualitative analysis of the query enhancement results. We have selected two bugs from two of the systems considered in our case study and we have analyzed in depth the effectiveness of the queries for both querying techniques (LSI and Grep). The examples of queries and their associated effectiveness metrics are shown in Table 2.7. In the upper part of the table, the arrow (->) indicates filtering, *i.e.*, the results obtained using the query indicated before the arrow are re-queried using the query indicated after the arrow.

When using Grep on FileZilla, the basic query can be improved by filtering its results with a query that exploits a neighboring concept present in all ontologies: the *ContextMenu* concept. Filtering with this concept, which was unavailable to the basic query, improves precision from 0.2 to 0.5. TPA and TIA extract another neighboring concept, *FileZilla*, which is not a neighboring concept in the UIA and UPA ontologies. Using this concept in the filter chain, precision can be further increased to its maximum, 1.

On JEdit's bug reported in Table 2.7 and with Grep, concept *EnhanceButton* is crucial to achieve maximum precision. However, only the TPA analyzer is capable of extracting the relations in the ontology that makes this concept a neighboring concept of the terms used for the basic query.

Table 2.7: Examples of best queries, with the corresponding results taken from two of our case studies. -> in grep based queries indicating a filtering relationship (P=Precision, R=Recall, F=F-measure, RR=Reciprocal Rank).

	Bug id	Grep	Basic	UIA	UPA	TPA	TIA
FileZilla	**3348**	**Query**	Remote.* tree.*view	Remote.* tree.*view ->context .*Menu	Remote.* tree.*view ->context .*Menu	Remote.* tree.*view -> file.*Zilla -> context.*Menu	Remote.* tree.*view-> file.*Zilla -> context.*Menu
		P	(1/5) 0.2	(1/2) 0.5	(1/2) 0.5	(1/1) 1.0	(1/1) 1.0
		R	(1/1) 1.0	(1/1) 1.0	(1/1) 1.0	(1/1) 1.0	(1/1) 1.0
		F	0.333	0.667	0.667	1	1
JEdit	**1275607**	**Query**	find -> focus	find -> focus	find -> focus	find -> enhance .*Button	find -> focus
		P	(1/8) 0.125	(1/8) 0.125	(1/8) 0.125	(1/1) 1.0	(1/8) 0.125
		R	(1/1) 1.0	(1/1) 1.0	(1/1) 1.0	(1/1) 1.0	(1/1) 1.0
		F	0.222	0.222	0.222	1	0.222

	Bug id	LSI	Basic	UIA	UPA	TPA	TIA
FileZilla	**3348**	**Query**	Remote tree view	drag, menu Download	item, image List	drag, menu Download	drag, menu Download
		RR	0.083	1	1	1	1
JEdit	**1275607**	**Query**	find, focus	find,focus	find, vf File Name Field	my J Radio Button, enhance Button	find, vf File Name Field
		RR	0.007	0.007	0.008	0.143	0.008

Hence, the TPA query is the only one with precision and F-measure equal to 1.

When LSI is used on FileZilla, a reciprocal rank of 1 is reached by all enhanced queries (see Table 2.7), while the basic query has a reciprocal rank of 0.083. This means that a developer using any of the extracted ontologies would find the correct answer to the query in first position, while without enhancing the query by means of the ontologies the developer would have to scroll the list of query answers up to position 12. It is interesting to notice that different concepts can be used to obtain a reciprocal rank of 1 (*e.g.*, *Drag, MenuDownload* vs. *Item, ImageList*), which explains why different ontologies (see answer to RQ2 below) can be equally good at improving the effectiveness of basic queries.

On JEdit, the results of LSI can be improved if concepts *MyJRadioButton* and *EnhanceButton* are included in the query. However, these concepts are available as neighboring concepts only in the ontology produced by TPA.

Table 2.8: Average number of keywords in the most effective queries used with Grep.

Systems	**Query types**				
	basic	**UIA**	**UPA**	**TPA**	**TIA**
FileZilla	1.32	2.12	1.70	2.04	2.08
JEdit	1.67	1.75	1.92	1.92	1.67
WinMerge	1.60	1.80	1.75	1.75	1.75

We have computed the average number of keywords used in queries to see if the improvement achieved by the best enhanced queries is through a sequence of successive filtering, when Grep is used (see Table 2.8). Results show that the average number of keywords in the enhanced queries is approximately two and that it is increased with respect to the basic query. This indicates that filtering plays an important role in improving the re-

sults of enhanced queries. On the other hand, an average of two means also that on average a developer has to filter the results only once, which does not require a lot of effort.

RQ2. Ontology comparison

To answer the second research question, we have computed the Jaccard index between each pair of ontologies and the ratio of unique concepts and relations each ontology has to their union (see Tables 2.9 and 2.10). The comparison of the paired ontologies is done by considering the union of all concepts, the union of all relations and the union of all paired concepts.

While the first comparison considers only concepts, the other two focus on pairs of concepts. In the comparisons which focus on pairs of concepts, the union of concept *relations* deals with named relations. On the contrary, the union of *paired concepts* is computed irrespective of the name of the relation connecting the two concepts. Hence, while two relations match if both concepts at the end of each relation and the relation names match, relation names are not taken into consideration while matching paired concepts. In short, concept relations are named while concept pairs are unnamed.

The Jaccard index computed for all union types of the paired ontologies show that there is some degree of similarity between the respective ontologies (see Tables 2.9 and 2.10). From the results, it is also apparent that each type of ontology is characterized by a peculiar set of concepts and relations. Hence, we can say that none of the ontologies subsume any of the other types of ontologies nor they are exactly the same.

The concepts and relations appearing in some but not all ontologies are due to the different parse trees generated by the different analyzers. For example, from the parse trees generated by UPA and TPA for the method name *findMatchingBracket* in class *TextUtilities* (see Figure 2.16), we get

Table 2.9: **RQ2: Pair wise comparison** between UIA ontology and the remaining three types of ontologies (UPA, TPA and TIA).

		Ontology			
		UIA ⊔ UPA	**Only in**		**Common (Ratio)**
			UIA	**UPA**	
FileZilla	**Concepts**	1940	780(0.402)	524(0.27)	636(0.328)
	Relations	2676	1254(0.469)	804(0.300)	618(0.231)
	Paired cpts	2446			848(0.347)
JEdit	**Concepts**	2911	602(0.207)	562(0.193)	1747(0.600)
	Relations	4163	1081(0.260)	1217(0.292)	1865(0.448)
	Paired cpts	3875			2153(0.556)
WinMerge	**Concepts**	2648	592(0.224)	809(0.306)	1247(0.471)
	Relations	3712	996(0.268)	1305(0.352)	1411(0.380)
	Paired cpts	3508			1615(0.460)
		UIA ⊔ TPA	**Only in**		**Common (Ratio)**
			UIA	**TPA**	
FileZilla	**Concepts**	1957	637(0.325)	541(0.276)	779(0.398)
	Relations	2718	1042(0.383)	846(0.311)	830(0.305)
	Paired cpts	2470			1078(0.436)
JEdit	**Concepts**	2966	564(0.190)	617(0.208)	1785(0.602)
	Relations	4267	1081(0.253)	1321(0.310)	1865(0.437)
	Paired cpts	3980			2152(0.541)
WinMerge	**Concepts**	2591	594(0.229)	752(0.290)	1245(0.481)
	Relations	3583	1061(0.296)	1176(0.328)	1346(0.376)
	Paired cpts	3381			1548(0.458)
		UIA ⊔ TIA	**Only in**		**Common (Ratio)**
			UIA	**TIA**	
FileZilla	**Concepts**	1618	610(0.377)	202(0.125)	806(0.498)
	Relations	2672	889(0.333)	800(0.299)	983(0.368)
	Paired cpts	2351			1304(0.555)
JEdit	**Concepts**	2662	970(0.364)	313(0.118)	1379(0.518)
	Relations	4417	1303(0.295)	1471(0.333)	1643(0.372)
	Paired cpts	4080			1980(0.485)
WinMerge	**Concepts**	2253	844(0.375)	414(0.184)	995(0.442)
	Relations	3409	1078(0.316)	1002(0.294)	1329(0.390)
	Paired cpts	3108			1630(0.524)

Table 2.10: **RQ2: Pair wise comparison** between UPA, TPA and TIA ontologies.

		Ontology			
		UPA ⊔ TPA	**Only in**		**Common (Ratio)**
			UPA	**TPA**	
FileZilla	**Concepts**	1582	262(0.166)	422(0.267)	898(0.568)
	Relations	2096	420(0.200)	674(0.322)	1002(0.478)
	Paired cpts	1952			1146(0.587)
JEdit	**Concepts**	2583	181(0.070)	274(0.106)	2128(0.824)
	Relations	3558	372(0.105)	476(0.134)	2710(0.762)
	Paired cpts	3192			3076(0.964)
WinMerge	**Concepts**	2143	146(0.068)	87(0.041)	1910(0.891)
	Relations	2851	329(0.115)	135(0.047)	2387(0.837)
	Paired cpts	2758			2480(0.899)
		UPA ⊔ TIA	**Only in**		**Common (Ratio)**
			UPA	**TIA**	
FileZilla	**Concepts**	1499	491(0.328)	339(0.226)	669(0.446)
	Relations	2365	582(0.246)	943(0.399)	840(0.355)
	Paired cpts	2126			1079(0.508)
JEdit	**Concepts**	2468	776(0.314)	159(0.064)	1533(0.621)
	Relations	4121	1007(0.244)	1039(0.252)	2075(0.504)
	Paired cpts	3473			2723(0.784)
WinMerge	**Concepts**	2279	870(0.382)	223(0.098)	1186(0.520)
	Relations	3608	1277(0.354)	892(0.247)	1439(0.399)
	Paired cpts	3395			1652(0.487)
		TPA ⊔ TIA	**Only in**		**Common (Ratio)**
			TPA	**TIA**	
FileZilla	**Concepts**	1557	549(0.353)	237(0.152)	771(0.495)
	Relations	2424	641(0.264)	748(0.309)	1035(0.427)
	Paired cpts	2169			1290(0.595)
JEdit	**Concepts**	2581	889(0.344)	179(0.069)	1513(0.586)
	Relations	4291	1177(0.274)	1105(0.258)	2009(0.468)
	Paired cpts	3680			2620(0.712)
WinMerge	**Concepts**	2218	809(0.365)	221(0.100)	1188(0.536)
	Relations	3437	1106(0.322)	915(0.266)	1416(0.412)
	Paired cpts	3213			1640(0.510)

different concepts and relations. UPA identifies *matching* as *xcomp* (clausal complement) which is not mapped to any of the ontological relations we defined. It has considered *Bracket* as the direct object of *<verb> matching*, which is mapped to the ontological relation *matching(TextUtilities, Bracket)*. TPA, on the other hand, has identified the *NN* (noun-noun specifier) natural language dependency between *Matching* and *Bracket*, and considered *MatchingBracket* as the direct object of *find*. As compared to the ontological relations produced by UPA, this results in two different ontological relations: *isA(MatchingBracket, Bracket)* and *find(TextUtilities, MatchingBracket)*.

If we consider the concepts produced by the two analyzers UPA and TPA, two of them are common, *Bracket* and *TextUtilities*. Concept *MatchingBracket* is extracted only by TPA, which, differently from UPA, correctly identifies the specifier dependency relationship between *matching* and *bracket*. In this case, training is crucial in order for the analyzer to be able to recognize the specifier dependency which is instead missed by the general purpose, untrained analyzer UPA. UPA misses one, potentially relevant concept, as compared to TPA.

The relations between the concepts identified by UPA and TPA are completely disjoint. While UPA identifies a *matching* relation between *TextUtilities* and *Bracket*, no such relation is reported by TPA, which, instead identifies two other relations, *isA* and *find*, connecting different pairs of concepts, which also means that the two analyzers do not identify any common paired concepts. When the extracted relations are used to determine the neighboring concepts that are used to enhance a query, the two analyzers may report different concepts, because of the difference in the extracted relations. In turn, this might affect the effectiveness of the enhanced query.

```
<root>
  <VB id="1" pos="0" role="null" phrase="find">
    <VBG id="2" pos="1" role="xcomp" phrase="matching">
      <NN id="3" pos="2" role="dobj" phrase="bracket">
      </NN>
    </VBG>
  </VB>
</root>
```

```
<root>
  <VB id="1" pos="0" role="null" phrase="find">
    <NN id="3" pos="1" role="dobj" phrase="bracket">
      <NN id="2" pos="3" role="nn" phrase="matching">
      <NN/>
    </NN>
  </VB>
</root>
```

Figure 2.16: Parse trees generated by UPA (top) and TPA (bottom) for the JEdit method name ***findMatchingBracket*** in class *TextUtilities*.

RQ3. Analyzer impact

The different types of ontologies used in this study are built using dependency parse trees generated by UIA, UPA, TPA and TIA. From the comparison of the ontologies (RQ2), we have seen that they are not exactly the same. RQ3 investigates if this difference has impacted the effectiveness of the enhanced queries used in concept location. To answer this research question, we have computed the net improvement achieved by each type of enhanced query over the other for both best and average, LSI and grep-based queries (see Tables 2.11 and 2.13 for the net improvement and Tables 2.12, 2.14 for a detailed comparison).

Values (except for the p-values) indicate the number of cases in which the enhanced query indicated in each column improves the enhanced query indicated in each row. A negative value indicates that it is the query in the row that improves the query in the column.

For the LSI-based approach, the net improvement of the top rank of one type of enhanced query over the other is marginal for all systems except WinMerge (see Table 2.11). The highest net improvements for WinMerge

Table 2.11: ***RQ3: Enhanced vs. enhanced queries; best queries.*** Net improvement of paired enhanced queries and the corresponding p-values as computed using their top ranks and best F-measures with the corresponding precision and recall measures. The p-values computed over values of MRR and F-measure are adjusted for multiple tests.

	System		**FileZilla**			**JEdit**			**WinMerge**		
	Enhanced query		**UPA**	**TPA**	**TIA**	**UPA**	**TPA**	**TIA**	**UPA**	**TPA**	**TIA**
LSI-based	**Top Ranks**	**UIA**	0	0	1	1	1	1	-6	-5	-7
		UPA		2	2		1	0		1	-1
		TPA			2			-1			-2
	P-value	**UIA**	0.99	0.93	1.00	1.00	1.00	1.00	0.11	0.16	0.07
		UPA		0.99	0.99		1.00	NaN		1.00	0.58
		TPA			0.99			1.00			0.35
Grep-based	**Precision**	**UIA**	-4	1	3	0	1	1	0	0	0
		UPA		6	6		1	-1		0	-1
		TPA			0			-2			-1
	Recall	**UIA**	0	-1	0	0	0	1	-2	-2	-2
		UPA		-1	0		0	1		0	0
		TPA			1			1			0
	F-measure	**UIA**	-4	1	3	0	1	1	0	0	0
		UPA		6	6		1	-1		0	-1
		TPA			0			-2			-1
	P-value (F)	**UIA**	0.99	0.99	0.93	1.00	1.00	1.00	0.57	0.57	0.57
		UPA		0.99	0.99		1.00	1.00		NaN	0.94
		TPA			0.99			1.00			0.94

Table 2.12: **RQ3: Detailed comparison of enhanced vs. enhanced queries; best queries.** Enhanced queries are compared on top ranks and best F-measure.

System	Enhanced query	Top ranks			Best F-measures		
		UIA			UIA		
		Better	Less	Equal	Better	Less	Equal
FileZilla	UPA	4	4	20	4	8	16
	TPA	3	3	22	5	4	19
	TIA	2	1	25	4	1	23
JEdit	UPA	1	0	10	2	2	8
	TPA	1	0	10	3	2	7
	TIA	1	0	10	2	1	9
WinMerge	UPA	0	6	13	3	3	14
	TPA	0	5	14	3	3	14
	TIA	0	7	12	3	3	14

System	Enhanced query	UPA			UPA		
		Better	Less	Equal	Better	Less	Equal
FileZilla	TPA	4	2	22	9	3	16
	TIA	5	3	20	10	4	14
JEdit	TPA	1	0	10	1	0	11
	TIA	0	0	11	1	2	9
WinMerge	TPA	1	0	18	0	0	20
	TIA	1	2	16	1	2	17

System	Enhanced query	TPA			TPA		
		Better	Less	Equal	Better	Less	Equal
FileZilla	TIA	4	2	22	4	4	20
JEdit	TIA	0	1	10	1	3	8
WinMerge	TIA	1	3	15	1	2	17

Table 2.13: ***RQ3: Enhanced vs. enhanced queries; average queries.*** Net improvement of paired enhanced queries and the corresponding p-values as computed using their median ranks and median F-measures with the corresponding median precision and recall measures.The p-values computed over values of MRR and F-measure are adjusted for multiple tests.

	System		**FileZilla**			**JEdit**			**WinMerge**		
	Enhanced query		**UPA**	**TPA**	**TIA**	**UPA**	**TPA**	**TIA**	**UPA**	**TPA**	**TIA**
LSI-based	**Median rank**	**UIA**	3	1	-1	1	1	1	1	-1	0
		UPA		-1	2		2	1		-2	2
		TPA			2			1			4
	P-value	**UIA**	0.99	0.99	0.99	1.00	1.00	1.00	0.94	1.00	0.96
		UPA		0.99	0.99		0.50	1.00		0.57	0.94
		TPA			0.99			1.00			0.75
Grep-based	**Precision**	**UIA**	2	4	5	0	-1	-1	7	6	2
		UPA		4	3		-1	-3		-2	-4
		TPA			2			-2			-2
	Recall	**UIA**	0	-2	0	0	0	0	0	0	-1
		UPA		-2	0		0	0		0	-1
		TPA			2			0			-1
	F-measure	**UIA**	4	6	4	0	-1	-1	6	5	1
		UPA		5	0		-1	-3		-1	-3
		TPA			-1			-2			-2
	P-value (F)	**UIA**	0.78	0.25	0.26	1.00	1.00	1.00	0.20	0.28	0.99
		UPA		0.99	0.99		1.00	1.00		1.00	0.29
		TPA			0.99			1.00			0.57

Table 2.14: **RQ3: Detailed comparison of enhanced vs. enhanced queries; average queries.** Comparison of enhanced queries on median ranks and median F-measure.

System	Enhanced query	Median Rank			Median F-measure		
		UIA			UIA		
		Better	Less	Equal	Better	Less	Equal
FileZilla	**UPA**	11	8	9	10	6	9
	TPA	10	9	9	8	2	15
	TIA	7	8	13	5	1	19
JEdit	**UPA**	5	4	2	2	2	8
	TPA	5	4	2	2	3	7
	TIA	5	4	2	1	2	9
WinMerge	**UPA**	8	7	4	9	3	8
	TPA	7	8	4	8	3	9
	TIA	6	6	7	6	5	9
System	**Enhanced query**	**UPA**			**UPA**		
		Better	**Less**	**Equal**	**Better**	**Less**	**Equal**
FileZilla	**TPA**	8	9	11	8	3	14
	TIA	11	9	8	7	7	11
JEdit	**TPA**	3	1	7	1	2	9
	TIA	4	3	4	1	4	7
WinMerge	**TPA**	0	2	17	0	1	19
	TIA	7	5	7	2	5	13
System	**Enhanced query**	**TPA**			**TPA**		
		Better	**Less**	**Equal**	**Better**	**Less**	**Equal**
FileZilla	**TIA**	10	8	10	4	5	16
JEdit	**TIA**	4	3	4	2	4	6
WinMerge	**TIA**	8	4	7	3	5	12

are observed when UIA is compared with UPA, TPA, and TIA, with net improvements of 6, 5, and 7, respectively. The pair-wise comparison result of the median ranks is also marginal for most cases while using LSI-based approach (see Table 2.13). The highest net improvement in this case is 4; and it is observed for WinMerge when comparing TIA with TPA. The details of the number of times one type of enhanced query is better, less than or equal to the other in terms of top and median ranks are shown in Tables 2.12 and 2.14.

The net improvements of the best F-measures of one type of enhanced query over the others while using grep-based approach are also marginal in all pairs except for FileZilla (see Table 2.11). In FileZilla, TPA and TIA enhanced queries are found more effective than both UIA and UPA enhanced queries. The highest net improvement, 6, is observed when comparing the highest F-measures of TPA and TIA with UPA. Like the best F-measures, the net improvements of the median F-measures are marginal for all pairs except for some cases of FileZilla and WinMerge (see Table 2.13). The highest net improvement, 6, is observed for FileZilla when comparing TPA with UIA, and for WinMerge when comparing UPA with UIA. The details of the number of times one type of enhanced query is better, less than or equal to the other in terms of effectiveness (F-measure) are shown in Tables 2.12 and 2.14. The results show that in the majority of the cases all pairs have performed almost equally.

To further analyze if the differences observed are statistically significant, we have formulated the hypothesis stated in H_{0-RQ3} and we have conducted a two-sided, pair-wise Wilcoxon signed-rank test (see Table 2.3, H_{0-RQ3}, LSI-based and grep-based approaches). The results are shown in Tables 2.11 and 2.13. The p-values in the tables indicate that the observed differences are not statistically significant at $\alpha = 0.05$ in all the cases.

From the results, we can conclude that the difference in the analyzers

used to build the ontologies has little or no impact on the effectiveness of the respective enhanced queries in concept location. An example of the results obtained for the different types of enhanced queries is shown in Table 2.7, where query improvements are achieved when the same neighboring concept is extracted by all ontologies (this is the case, *e.g.*, of concept *ContextMenu* on the FileZilla bug), but also when different, but equally useful concepts are extracted by different ontologies (*e.g.*, concepts *Drag, MenuDownload* extracted by UIA, TPA and TIA are equally effective in improving the basic query as concepts *Item, ImageList* extracted by UPA). This shows that ontologies that are remarkably different (see answer to RQ2) can be equally good at improving the basic queries and none of them is superior to the others when a concept location task is performed.

Discussion

To carry out a concept location task, developers can use either LSI or grep-based approach to query the code base. Results of RQ1 show that expert developers who can formulate best queries and use either of the approaches to query the code base benefit from using ontologies extracted employing the analyzers (see Table 2.5). Average programmers who usually formulate less effective queries, on the other hand, benefit from using the ontologies if they use the grep-based approach to query the code base (see Table 2.6). Using the LSI-based approach with enhanced queries formulated by average developers did not show any improvement over using the basic queries.

When using the LSI-based approach, the enhanced queries formulated by expert developers result in ranking relevant documents closer to the top than the basic queries. The improvement in ranking allows developers to go through a lower number of documents before finding the relevant document. The effectiveness improvement observed while using the grep-based approach is the result of the improvement in both precision (P) and

completeness (R) of the enhanced queries formulated by expert developers. The improvements in precision indicates that the developer has to explore fewer files to identify those relevant for the task at hand, while the improvements in the completeness of the enhanced queries indicates that more relevant files are incorporated in the list of files to be explored than what can be retrieved using the basic queries.

The effectiveness improvement observed in the median F-measures of enhanced queries while using the grep-based approach is mostly due to the improvement in precision (see Table 2.6). Though the enhanced queries completeness slightly decreased, the increase in precision compensates for such decrease. The higher precision is associated with a lower effort required to locate a relevant file. Once a relevant file is located, the remaining parts of the system affected by the change can be identified using impact analysis and change propagation.

While comparing the basic and enhanced queries, we hold all variables constant except the queries. Hence, the observed improvements are due to the concepts taken from the ontologies extracted from the source code and used to formulate the enhanced queries. The result shows that concepts which are found in the ontology, and are related to the concepts in the bug description improve the ranking of relevant documents and narrow down the search space.

In our study, we have used four different types of analyzers to generate ontologies. The comparison of the resulting ontologies show that they are different and are sensitive to the type of analyzer used (see Table 2.9). The observed difference is due to different parse trees generated by the analyzers. The analyzers use data-driven natural language parser and differ in architecture (see Section 2.1). The difference, however, did not exhibit any significant impact on the concept location task, which exploits the ontologies to enhance queries (see Tables 2.11 and 2.13).

Based on our findings, an expert developer can use any of the analyzers to extract an ontology from the identifiers and explore the ontology to find possibly related concepts which improve the effectiveness of concept location queries. For example, the addition of concepts taken from the ontologies and used to formulate UIA, UPA, TPA, and TIA enhanced queries have improved the effectiveness of the basic query formulated for FileZilla bug id *3348* (see Table 2.7) by a large amount (reaching $F = 1$ with TPA, TIA and Grep, and reaching $RR = 1$ with all analyzers and LSI). For an expert developer the improvement on the effectiveness of a query is not dependent on either of the approaches (LSI vs. grep-based) used to query the code base. For an average developer, however, a substantial improvement is achieved only when Grep is used.

2.5.2 Comparison of NLP vs. structural based concept extraction

In this chapter, we have proposed two approaches to extract concepts from the source code and build an ontology, which can support program understanding. The approaches are based on the natural language information captured in identifiers (see Section 2.2) and the structure of the source code (see Section 2.3). We call the ontology built following the former approach as *NLP ontology* while we refer to the ontology built following the latter approach as *structural ontology*. In this section, we investigate if the two approaches result in two different ontologies, and, if the difference exists, we study the impact on the support they give to a program understanding task, concept location. In the previous section, we have shown that the ontology built following the NLP-based approach improves the effectiveness of queries formulated to locate a concept. Here, we carry out a similar study using the structural ontology and the union of the structural and NLP ontologies. In particular, we address the following two research

questions:

- ***RQ1. Structural vs. NLP ontology:*** *Is there any difference between structural and NLP ontologies?*
- ***RQ2. Support for concept location:*** *Do the structural ontology and the union of structural and NLP ontologies increase the effectiveness of programmer's queries formulated for concept location?*

NLP and structural based ontologies are generated by exploiting two different aspects of the source code. In RQ1, we investigate if this difference results in ontologies which are composed of different sets of concepts. To explore the differences and similarities, we compare each ontology to the union of the two ontologies, and compute unique concepts ratio ($|A \setminus B|/|A \cup B|$, $|B \setminus A|/|A \cup B|$) and Jaccard index ($|A \cap B|/|A \cup B|$). In RQ2, we further analyze the impact of the difference on the support the ontologies provide to concept location (see Section 2.4).

Like in Section 2.5.1, the evaluation of RQ2 is conducted by comparing the effectiveness of queries formulated using only keywords taken from the bug descriptions which we refer to as *basic queries* with *enhanced queries* [4] that use also concepts from either of the ontologies in addition to the keywords from the bug descriptions. As defined in Section 2.5.1, the effectiveness of the queries is evaluated by computing *F-measure (F)*.

For RQ2, we have formulated the following null/alternative hypotheses to investigate if the differences between the effectiveness of the two types of queries is statistically significant.

> $\mathbf{H_0}$: There is no statistically significant difference between the effectiveness of basic queries and the effectiveness of enhanced queries.
>
> $\mathbf{H_1}$: There is statistically significant difference between the effectiveness of basic queries and the effectiveness of enhanced queries.

To conduct the statistical test we have used two-sided, paired Wilcoxon signed rank test. To control the false discovery rate and correct for multiple comparison, we have adjusted the p-values using the Benjamini and Hochberg (BH) [15] correction.

Procedure

To conduct our experiment we have developed two tools which implement the approaches described in Sections 2.2 and 2.3. The tools operate on the XML representation of the source code which is generated using src2srcml [35]. They respectively produce the NLP based and the structural ontology (see examples in Figures 2.14 and 2.15) for the input system. The tool which is used to extract the NLP based ontology uses the output of UIA (Minipar) (see Section 2.1) as an input to generate the NLP based ontology.

To automatically reenact concept location, we have collected bug reports which are closed and have patch files in the associated bug tracking system. In Table 2.15, we list the number of bugs which have been collected for each system. The patch files are used to identify the files which are actually changed to fix the reported bug. The names of these files are used to finally evaluate the results of our experiments. Keywords which are deemed relevant for the concept location task have been manually collected from the titles of each bug description. These keywords are used in formulating *basic query*. The *enhanced queries* are formulated using concepts taken from the ontologies and keywords [4]. The selection of concepts to be used in the enhanced queries is done by first matching keywords to concepts in the ontology and taking the neighboring concepts that are one edge away.

We have applied the queries on the source code files. To query the source code files, we resort to a very simple (yet widely used) method, namely

`grep`[12]. In Section 2.5.1, we have also used the state-of-the-art approach, LSI, to query a code base, and we have shown that for an expert user who can formulate effective queries both approaches give similar results while for the average user who formulates less effective queries grep is better. Hence, in this study we consider only grep-based queries. The activities involved in the grep-based approach are discussed in Section 2.5.1.

Subjects

To conduct the study we have used five open source programs: ADempiere, FileZilla client, JEdit, OpenOffice, and ThunderBird. The summary of the systems is shown in Table 2.15.

Table 2.15: Summary of systems.

System	Version	Files	Classes	Lines of text	No. of Bugs
ADempiere	3.1.0	1833	1917	482094	10
FileZilla	3.0.0	264	208	89080	28
JEdit	4.2	224	639	79198	12
OpenOffice	1.0.0	12761	12112	4666417	18
ThunderBird	2.0.0.0	11019	5949	3548012	12

ADempiere[13] is an enterprise resource planning software, while FileZilla client[14] is a cross-platform, graphical FTP, FTPS, and SFTP client. JEdit[15] is a programmer's editor which provides syntax highlighting for over 2000 file formats. OpenOffice[16] is an office software suite for word processing, spreadsheets, presentations, graphics and databases, while ThunderBird[17] is an email and news client developed by Mozilla foundation. In all

[12]http://www.gnu.org/software/grep/
[13]http://www.adempiere.com/
[14]http://filezilla-project.org/
[15]http://www.jedit.org/
[16]http://www.openoffice.org/
[17]http://www.mozilla.org/en-US/thunderbird/

these systems, most components are developed using the object oriented paradigm. Two of the systems, JEdit and ADempiere, are developed using Java, while the other three are developed using C++.

Results

RQ1: Structural vs. NLP ontology

To investigate the differences and commonalities of the structural and NLP ontologies, we have compared each ontology's concepts and relations to their union (see Table 2.16). For each system, we have created three types of unions: concepts, relations and paired concepts (see Section 2.5.1, RQ2).

Table 2.16: Comparison of the union of concepts and relations extracted using NLP and structural approach with the individual approaches.

Systems	Ontology	NLP ∪ Str.	Only in NLP.(Ratio)	Only in Str.(Ratio)	Common (Ratio)
ADempiere	**Concepts**	12346	3052(0.247)	1424(0.115)	7870(0.637)
	Relations	45890	11850(0.258)	25552(0.557)	8488(0.185)
	Paired cpts.	42747			11631(0.272)
FileZilla	**Concepts**	1441	537(0.373)	98(0.068)	806(0.559)
	Relations	3339	1517(0.454)	1539(0.461)	283(0.085)
	Paired cpts.	3157			465(0.147)
JEdit	**Concepts**	2535	592(0.234)	304(0.12)	1639(0.647)
	Relations	5740	2230(0.389)	2922(0.509)	588(0.102)
	Paired cpts.	5419			909(0.168)
OpenOffice	**Concepts**	75763	24234(0.32)	4384(0.058)	47145(0.622)
	Relations	233271	89933(0.386)	113496(0.487)	29842(0.128)
	Paired cpts.	222865			40248(0.181)
ThunderBird	**Concepts**	34819	13092(0.376)	3221(0.093)	18506(0.531)
	Relations	91466	39721(0.434)	43071(0.471)	8674(0.095)
	Paired cpts.	86944			13196(0.152)

The Jaccard index (shown in the last column of Table 2.16 within brackets) indicates that the two types of ontologies have some parts in common, but they are not exactly the same. The similarity between the two ontolo-

gies is higher for concepts than for relations. When considering the unique concepts of the respective ontologies, the NLP based ontologies have more unique concepts than the structural based ontologies. This, however, is reversed in all systems when considering unique relations. Hence, to get a more complete set of concepts and relations in a program, the union ontology is recommended.

RQ2: Support for concept location

Table 2.17 shows the average precision, recall and F-measures of the concept location task for the enhanced query and the corresponding percentage delta over the basic queries. The values are computed for structural ontology and NLPStr ontology, which is the union of NLP and structural ontologies.

Table 2.17: Average precision(P), recall (R) and F-measure (F) of concept location using the enhanced queries (with delta percentage within brackets); The adjusted P-values indicate the statistical significance of the difference.

Ontology	System	Enhanced query			P-value
		Avg. P(Δ%)	Avg. R(Δ%)	Avg. F(Δ%)	
Str.	ADempiere	0.676(346)	0.9(0)	0.717(210)	***0.0039***
	FileZilla	0.483(91)	0.844(0.148)	0.516(71.4)	***0.0025***
	JEdit	0.553(38.5)	0.958(1.47)	0.596(30.7)	0.0591
	OpenOffice	0.346(1031)	0.878(-7.33)	0.357(541)	***0.0025***
	ThunderBird	0.173(38.2)	0.833(0)	0.202(56.2)	0.1000
NLPStr	ADempiere	0.744(391)	0.9(0)	0.77(233)	***0.0039***
	FileZilla	0.55(118)	0.842(-0.0551)	0.601(99.7)	***0.0005***
	JEdit	0.652(63.2)	0.958(1.47)	0.691(51.6)	0.0591
	OpenOffice	0.448(1362)	0.818(-13.7)	0.456(719)	***0.0017***
	ThunderBird	0.324(159)	0.833(0)	0.363(180)	***0.0215***

In all the systems, the average effectiveness (F-measure) of the enhanced query is higher than the basic query. Negative delta percentage occurs only for the recall of OpenOffice and FileZilla. However, these negative values

are not reflected in the overall effectiveness as there is a large average improvement of the precision. The improvement in the precision indicates that there is a reduction of the search space, which also reduces the effort of the developer who executes the concept location task. To see if the average effectiveness improvement is statistically significant, we have formulated the hypothesis stated in H_0 and computed the two-sided, paired Wilcoxon signed rank test.

For the union of NLP and structural ontologies, the observed improvement on the effectiveness of enhanced queries is found statistically significant (at $\alpha = 0.05$) in four of the five systems, while for the structural ontologies the observed difference is statistically significant in three of the five systems. Hence, for majority of the systems, we can reject the null hypothesis.

The effectiveness of programmer's queries has improved for all systems when using the ontology concepts and this is found statistically significant in more than half of the systems. Hence, we can answer RQ2 affirmatively.

Discussion

The comparisons of NLP and structural based ontologies show that the two ontologies are complementary (see Table 2.16). The unique concepts found in each of the ontologies could be due to the different aspects of the source code which are exploited when extracting them. Hence, we think that using the two ontologies in combination (i.e., computing their union) could be useful for developers.

In RQ2, we have investigated the support of the structural ontology and the union of the structural and NLP ontologies to enhance concept location queries. The results show that both types of ontologies are practically helpful to enhanced queries in all systems (see Table 2.17, F-measure delta percentage). However, among the two, the F-measure delta percentages

obtained for the NLPStr ontology are greater than the structural ontology. The average of the F-measure delta percentages for the structural ontology is 181.9%, while for the NLPStr ontology it is 256.7%. The NLPStr ontology also has a higher average F-measure delta percentage when compared with the average F-measure delta percentage of the NLP ontology, 236.3%. Besides, these observed improvements are found statistically significant in four of the five systems for the union ontology (see Table 2.17, p-values). From these results we can see that the effectiveness of the enhanced queries is better if a developer uses the union of NLP and structural ontologies to enhance concept location queries.

2.5.3 Threats to validity

The main threats to the validity that can affect the results of the evaluations in Sections 2.5.1 and 2.5.2 can be of four different types [114]: construct, internal, conclusion, and external validity threats. In the following we discuss them together with the strategies adopted to minimize their effect.

Construct validity threats concern the relationship between theory and observation. The main point to consider involves the adopted effectiveness measures, which in our case depend on whether the results are ranked or not. In the former case (see Section 2.5.1), the Reciprocal Rank aims at measuring the effort necessary to get the correct answer by considering results in the list starting from the top. In the latter (see Sections 2.5.1 and 2.5.2), we considered the usual measures of precision, recall and F-measure.

However, to be more precise, the effort required from the user to process the output crucially depends on the choice of the basic query and on the following filtering performed by the developer. Performing this activity manually would have caused all threats to validity connected to human involvement [114], including the possibility that the subject learns how the

system behaves or tends to unconsciously alter the system performance. We therefore preferred an automatic strategy, which simulates developers activity and also makes replicability easier.

Internal validity threats concern additional factors that may affect an independent variable. First of all, one threat can derive from the fact that the basic queries employed in the evaluation have been created by one of the researchers involved in the study from bug report titles. In both studies, to minimize this problem we have decided a-priori a standard strategy for query creation. We considered all the keywords as initial query and then filtered the results. Even more importantly, the researcher has carried out query creation prior to computing any result.

Another factor which can influence the system effectiveness in the study conducted in Section 2.5.1 is the size of data available for training the natural language parsers in TPA and TIA cases. It is largely known that for all machine learning approaches, the combination of quantity and quality of training data is crucial for effectiveness. In our case, we constructed the training set by the union of documentation internal to the project and general data sets, based on the documentation commonly available in a software project. While such documentation is not required to be fully aligned with the implementation for our purposes, a completely misaligned or outdated documentation might impair the training phase.

Conclusion validity threats concern the relationship between the treatment and the outcome. The validity of the results strongly depends on the choice of the baseline system. We considered the LSI-approach, which at the moment is largely regarded as the state of the art in concept location (see Section 2.5.1) and have shown that the results are consistent with the more traditional and widely used grep-based approach for effective queries.

Another point regards the statistical significance of the results. We adopted the Wilcoxon signed-rank test to compute the p-value for each pair

of considered systems. However, in our case such pair-wise comparisons must be combined to obtain a global ranking of all the four proposed approaches and the baseline, hence involving multiple tests. To take such multiple tests into account, we have applied the Benjamini and Hochberg (BH) [15] adjustment of the p-values.

External validity concerns the generalization of the findings. The study conducted on comparison of analyzers (see Section 2.5.1) involved three open source software applications, two written in C++ and one in Java, while the study conducted on the comparison of the two concept extraction approaches (see Section 2.5.2) involved five systems of which two are written in Java and the rest in C++. Developers in open source projects usually are very careful in producing code which can be easily modified by someone else. This implies that they try to exploit at best natural language semantics, and this can help text mining approaches as the one we are proposing. We feel that the approach could be effective also on commercial systems, but this should be verified by further assessment.

In general, we hypothesized that the programming language is object-oriented and considered class, attribute and method names. The approach could be easily extended to other kinds of languages, and we are confident that the resulting trend would not change, because the techniques we use exploit the natural language properties rather than the programming language ones. However, such generalization would require further experimentation. On the other hand, although English is largely employed in software comments and identifiers, cross-language adaptation would be necessary if other languages are adopted in addition to or in substitution of English.

Last but not least, we adopted the available bug fixing patches to assess the results of concept location, as usually done in the concept location literature. However, bug fixing could require intervention only in a proper

subset of all the places where the concept occurs and the number of gold positives could be larger than the one considered.

2.6 Conclusion

In this chapter, we have presented four types of natural language analyzers to parse identifiers and two approaches to extract concepts and inter-concept relations from identifiers. Two of the analyzers have been adapted to directly work on identifiers through training while the other two are standard English analyzers. The training of the analyzers was conducted automatically using a training set constructed from the documentation of the corresponding system. To extract concepts and inter-concept relationships, we have used natural language dependency relationships between the terms used to construct identifiers (NLP-based approach) and the program's structural information which is based on object oriented programming (structural-based approach).

In our study, we have used the analyzers to identify natural language dependencies between terms in the identifier and to extract concepts and inter-concept relationships. To analyze the effect of the analyzers in extracting concepts and inter-concept relationships, we have conducted a case study. The study was conducted in the context of the support the extracted concepts give to concept location while using LSI and grep-based approaches. The results of the study, show that using concepts taken from the ontologies extracted from the respective systems improves the effectiveness of concept location queries which can be formulated by experts, while using both LSI and grep-based approaches. This is achieved irrespective of the type of natural language analyzer used in the study. The statistical test conducted on the results also confirms this observation in the majority of the cases (i.e., the results of at least 8 out of 12 cases for both LSI-based

and grep-based approaches are found statistically significant at $\alpha = 0.05$). For average queries, improvement in effectiveness of queries is observed only when using the grep-based approach. The improvements observed in this case are statistically significant at $\alpha = 0.05$ in half of the cases.

The comparison of the ontologies generated using different analyzers shows that they are different, with some concepts and relations in common. However, this did not impact the support they give to concept location. The comparison on the support they give to concept location show that in the majority of the cases, they perform equally well and the observed small differences are not statistically significant.

The ontology extracted using the NLP based approach was also compared with the ontology extracted using the structural based approach in terms of the concepts they contain and the support they give to concept location. Our results indicate that the two ontologies have several unique concepts, which might stem from the different aspects the corresponding approaches exploit to extract the ontologies. The impact of the unique concepts is also reflected on concept location, where the union of the two ontologies gives better support for concept location than the individual ontologies.

Chapter 3

Domain concept filtering

A program is a formal representation of a solution to a problem, as perceived by its developers. At various times, different programmers need to understand the domain concepts captured in the code to perform maintenance tasks. However, understanding the domain knowledge captured in a program written by others or written some time ago is usually a difficult and time consuming activity. In fact, a solution of a problem in a domain can be formalized in a number of ways. Often there is no way of knowing how the domain knowledge is captured in the program other than by reading the code. Reading the code of a large program on the other hand is a difficult and expensive activity, which can be highly supported by approaches such as those described in Chapter 2 and by Ratiu *et al.* [102].

The approach described in Chapter 2 extracts concepts captured in program element names, while Ratiu *et al.* [102] extract common concepts captured in the APIs implementing a similar functionality. Both approaches use ontologies to present the concepts extracted and the inter-concept relationships. These approaches do not make any distinction between concept types, i.e., domain vs. implementation concepts. A concept in an ontology may represent a domain concept (e.g., *bank account*) or an implementation concept, such as a data structure (e.g., *array*, *list*), a GUI element (e.g.,

button, *canvas*), etc. In this chapter we present an information retrieval (IR) based approaches to separate (*filter*) domain concepts from implementation concepts [5]. The IR based techniques used in the filtering are keyword and topic based approaches. They are used in various natural language processing applications to identify the representative words, to perform clustering and to identify the topics present in a corpus of documents [103, 80, 51].

In the following sections we present two keyword based filtering techniques, a non-interactive and an interactive techniques, and a topic based filtering technique. The evaluation of the filtering techniques is described in Section 3.4.

3.1 Non-interactive keyword based filtering

In a document, keywords which refer to domain concepts are used with other auxiliary words to convey the intended (domain) message to its readers. A term is considered as a domain keyword depending on the context in which it is used. For example, the term *credential* may not be considered as a domain keyword when it is used in a document related to a *bank system*, while it is definitely a domain keyword if it is used in documents related to *security*. In information retrieval (IR) and text mining, different techniques are used to identify keywords of documents, such as term frequency in a document (TF), term weight ($TW = TF/AT$, AT= total number of terms in a document), and term frequency-inverse document frequency ($TFIDF = TW * log(D/d)$, D= number of all documents, d= number of documents containing the term). These techniques give a weight to the words in the documents and rank them in descending order of likelihood of being a keyword. Term frequency considers terms which have high frequency as keywords while term weight normalizes the frequency

by dividing it to the total number of terms in the respective documents. Term frequency-inverse document frequency, on the other hand, considers the distribution of a term across documents in addition to the term weight, to give more weight to document specific terms. A cut point or a percentage is used to take the terms at the top, which are considered more representative keywords of the domain.

To identify keywords related to the software system under analysis, we have opted to use the corresponding documentation (e.g., the user manual, the web site, etc.) as our main source. The documentation, usually, contains a description of the solution implemented in the source code. Hence, it is quite likely to contain domain keywords which are also used in program element names, in the source code. The keywords from the source code documentation are collected using term frequency (TF). In fact, we are not interested in terms that characterize a document specifically. Therefore, we merge all the documentation to one file, which makes the other techniques not applicable. Prior to applying the TF technique, we pre-process the merged document to remove stop words which are common English words (e.g., *the*, *is*, etc) and programming keywords. To remove variants of a word and have only their common root, we use stemming. Like in our previous study [5], we have used three different cut points, top 15, 50 and 100 terms, to obtain the set of keywords from the terms ranked by TF. We considered these cut points to investigate the impact of different thresholds and have a reasonable set to manually analyze.

The terms which are identified as keywords are used to automatically filter the recovered ontology. The filtering is conducted by matching identified keywords to the terms used to represent a concept in the ontology. A concept in the ontology is kept if all the composing terms of the concept have been identified as keywords. A relation is kept if both source and target concepts are kept. For example, for the ontologies of our running

Figure 3.1: The filtered ontology produced for the NLP ontology which is shown in Figure 2.14.

example, Figures 2.14 and 2.15, if we find as keywords all terms used in the ontology but *login*, *password*, *user* and *credential*, this approach will filter the part of the ontology purely related to the *bank system* domain (see Figures 3.1 and 3.2).

3.2 Interactive keyword based filtering

In the non-interactive keyword based filtering, all the terms which are classified as keywords using the three cut points are used for filtering. However, closer examination of the keywords in some preliminary experiments has shown that they contain terms such as *click* and *menu* which are used

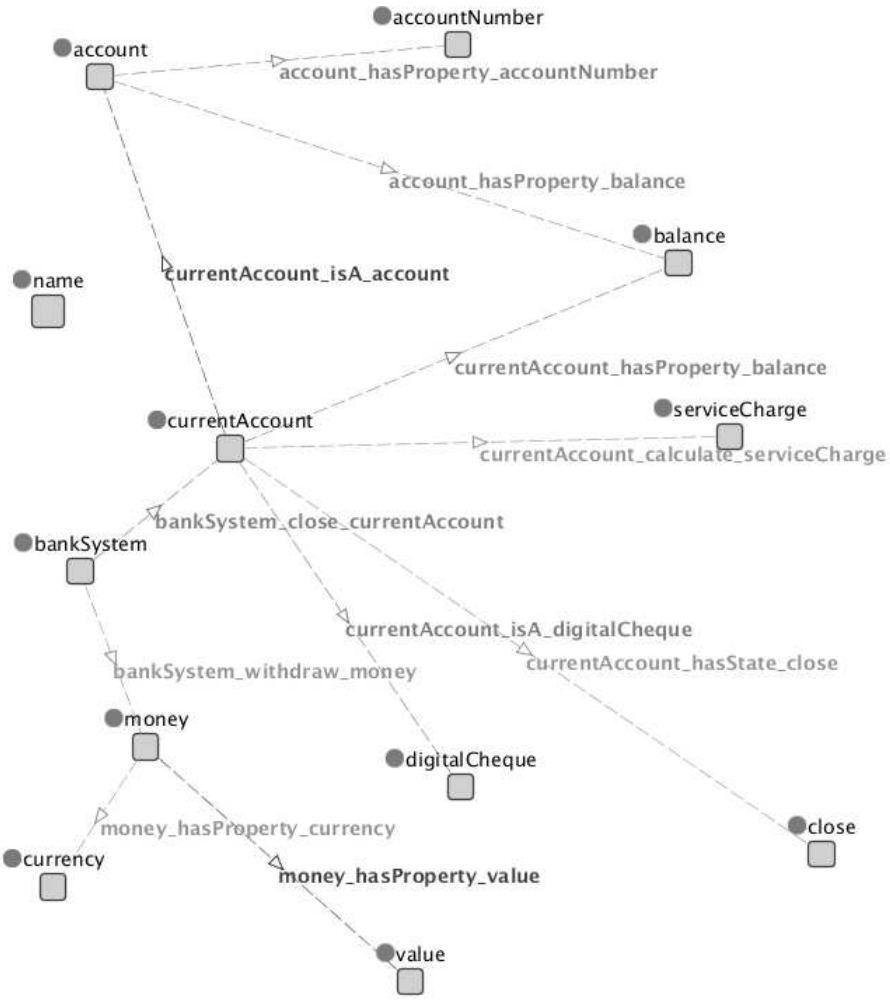

Figure 3.2: The filtered ontology produced for the structural ontology which is shown in Figure 2.15.

to describe *how to's* and *GUI elements* [5]. These terms are relevant for describing the solution implemented in the source code, hence they are typical of the documentation, but they are not domain terms. We introduce a limited manual intervention for keyword selection, which we call *domain keyword selection*.

The process of *domain keyword selection* involves a developer's decision on whether or not to consider a keyword as a true domain keyword. If a keyword is considered as a true domain keyword, it is kept in the list for the next step while the others are removed. This manual selection is

conducted for the top 15, 50 and 100 keywords automatically selected using one of the techniques described above and hence requires minimal effort.

The keywords we have after the *domain keyword selection* are then used in a similar way as in the non-interactive keyword based filtering to match the terms representing a concept in the ontology and filter the domain concepts.

3.3 Topic based filtering

Even though a software is an implementation of a solution to a given problem in a domain, it usually incorporates auxiliary implementations from other domains such as security, logging, GUI, etc. In the absence of proper documentation, a developer can learn about such auxiliary implementations by looking at the identifiers of the corresponding implementations. Considering the source code as a collection of documents composed of identifiers, one can similarly match the different but related domain implementations present in the source code to the topics in a document. A *topic* in a document represents a concept and is described using a collection of words [19]. In IR, various techniques such as pLSI (Probabilistic Latent Semantic Indexing) [59, 20] and LDA (Latent Dirichlet Allocation) [20, 19] are used to identify collections of words which correspond to the topics composing a document.

In this sub-section, we describe *pLSI* and *LDA* which we used to identify terms that correspond to the two major topics present in the source code: domain and implementation. The source code is considered as a collection of documents (files), each containing identifiers from classes, methods and attributes (split into stemmed terms).

pLSI (Probabilistic Latent Semantic Indexing): Also known as *aspect model*, is a generative topic modeling technique which is based on a prob-

abilistic model that uses the maximum likelihood principle [59, 20]. pLSI models a document as a mixture of words taken from different (unknown) latent topics from which the document is composed. The document model is achieved by first taking a document d, its probability $P(d)$ and the joint probability with the contained words w, $P(d, w)$. In order to select a latent topic z for d and to characterize the words w in a topic z, the probabilities $P(z|d)$ and $P(w|z)$ are estimated using the expectation maximization algorithm. This algorithm alternates two probability estimate steps: (1) the estimate step, to compute $P(z|d, w)$; and, (2) the maximization step, to compute $P(w|z), P(d|z), P(z)$. From the estimated probability $P(w|z)$ we can characterize topics as sets of most probable words, which we use as keywords for the topic.

LDA (Latent Dirichlet Allocation): The basic idea of LDA [20, 19] is also that documents are represented as a mixture of words taken from different latent topics, where each topic is characterized by a distribution over words. LDA is an advancement over pLSI, which addresses pLSI's limitations. pLSI generates the topic mixture, $P(z|d)$, only for documents which are known in advance, preventing its use with new documents. LDA overcomes this limitation by assuming that the probability distribution of a document over topics $P(z|d)$ is generated from a Dirichlet distribution with K parameters. In addition, this assumption addresses pLSI's problem of linear growth of parameters to be estimated as the number of documents grows, which may result in over-fitting [20, 28].

Both pLSI and LDA rank terms (words) representing a topic using a probability of membership, $P(w|z)$. The membership probability indicates the level of representativeness of the term in the respective topics in which it is found. To restrict the list of terms which represent a topic and have only those which are better representative, we have used three different cut points (minimum thresholds), 0.01, 0.02, and 0.03. The resulting set

of terms characterizing each topic is then regarded as keywords that are used to filter the ontology.

Similar to the keyword based approaches, the terms representing a concept in the ontology are matched to the keywords of a given topic. If all are matched, then the concept is kept in the filtered ontology. As we have two sets of terms corresponding to the two topics, domain and implementation, we will have two filtered ontologies. One of the filtered ontologies corresponds to the domain while the other corresponds to the implementation. For example, for the running example shown in Figure 2.6, the topic model may give us two sets of terms, one related to the implementation, including *login*, *password*, *user*, *credential*, and the other containing terms related to the domain, such as *account* and *balance*. Using these sets to filter the ontology results in two filtered ontologies, one corresponding to the domain and the other to the implementation. The latter can be easily identified and discarded (see Figures 3.1 and 3.2).

3.4 Evaluation

To assess the need for filtering and evaluate the filtering techniques, we have formulated the following three research questions. The evaluation is conducted using the three domain concept filtering techniques described in Sections 3.1, 3.2, and 3.3. For what concerns the ontology, the two, NLP and structural based, concept extraction approaches described in Sections 2.2 and 2.3 are used.

- ***RQ1. Adequacy of filtering:*** *Is there a filtering relationship between the concepts extracted following the NLP and structural based approaches and the domain concepts?*
- ***RQ2. Effectiveness of filtering techniques:*** *How effective are*

filtering techniques based on information retrieval in separating domain concepts from implementation concepts?

- ***RQ3. Support for concept location:*** *Do the filtered ontology concepts increase the effectiveness of programmer's queries formulated for concept location?*

As in the source code, where domain concepts are found together with implementation concepts, the ontologies we extract are also expected to be composed of two sets of concepts: domain and implementation. RQ1 focuses on analyzing this conjecture. In order to address RQ1 we will consider *gold concepts* which are concepts that express the problem domain implemented in the program for which the ontology is extracted. The gold domain concepts are manually collected from the user manuals of the corresponding systems. In fact, for RQ1 it is not important that the separation can be achieved in an automated way. What is important for RQ1 is that such separation, even if obtained manually, shows that most domain concepts are actually present in the automatically recovered ontology, such that a properly defined filter may in principle distill them from the ontology.

Assuming that a filtering relationship holds between extracted concepts and domain concepts, the next question is if and how such a filtering process can be automated. To this aim we investigate RQ2, where alternative IR based filtering techniques to separate domain from implementation concepts are compared.

In Section 2.5.2, we have shown that ontologies improve the effectiveness of queries used in concept location. In RQ3, we investigate if the filtered domain concepts can also be used to improve the effectiveness of queries formulated to locate concepts as compared to the unfiltered ontology. The investigation is conducted by comparing the effectiveness of

queries formulated using keywords taken from bug descriptions and the unfiltered ontology with those that use concepts from the filtered domain ontologies in addition to the keywords from the bug descriptions. We call such queries as enhanced queries. We have formulated the following null/alternative hypothesis to investigate if there is a statistically significant difference between the support provided by the filtered and unfiltered ontologies to concept location.

> **H_0** : There is no statistically significant difference between the effectiveness of enhanced queries formulated using concepts taken from filtered ontologies and effectiveness of enhanced queries formulated using concepts taken from unfiltered ontologies.
>
> **H_1** : There is statistically significant difference between the effectiveness of enhanced queries formulated using concepts taken from filtered ontologies and effectiveness of enhanced queries formulated using concepts taken from unfiltered ontologies.

To carry out the statistical test, we have conducted a two-sided, paired Wilcoxon signed rank test. To control the false discovery rate and correct for multiple comparison, we have adjusted the p-values using the Benjamini and Hochberg (BH) [15] correction.

3.4.1 Procedure

To conduct the analysis, we have followed the steps shown in Figure 3.3. To generate the filtered ontology, we first of all followed the steps described in Sections 2.2 and 2.3 to extract two ontologies from the source code. We refer to the ontology extracted using the NLP based approach and UIA (see Section 2.2) as *NLP ontology* while the ontology extracted using the structural information is referred to as *Str ontology*. We have also created

a third ontology, *NLPStr ontology*, which is the union of the two types of ontologies. This step can be replaced by any approach which extracts an ontology directly from the source code.

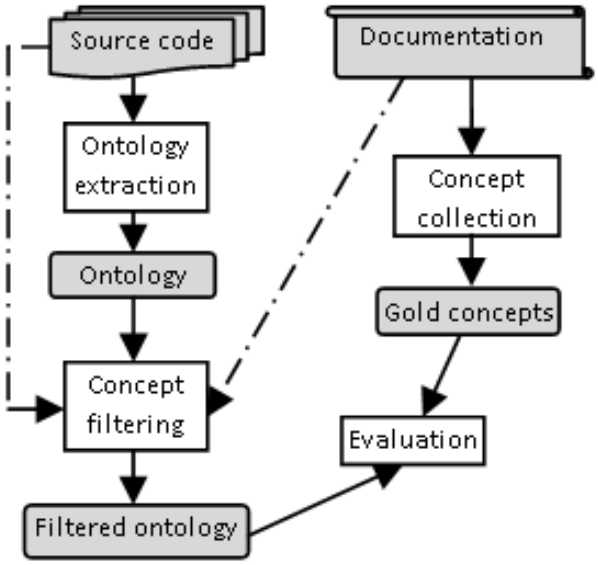

Figure 3.3: Overview of ontology filtering and evaluation process.

To filter the extracted ontologies, we use the IR techniques described in this chapter. The filtering techniques, as shown by the dotted arrows in Figure 3.3 use information from either the source code or the system's documentation, in particular user manuals as an input. To automate filtering, we have used the tool developed for our previous study [5]. The tool implements the IR filtering techniques described above. It uses the Dragon Toolkit [115] as a plug-in to generate topic terms from the source code corpus. The source code corpus is constructed from class, attribute, and method identifier terms identified after splitting the names using camel casing and underscore. The toolkit has a configuration file where the number of topics to be generated by both pLSI and LDA is set and the values for the parameters required by LDA, α and β, are specified. Like in our previous study [5], we have set the value of α to 25 (50/Number of Topics),

β to 0.1, and number of topics to 2. The same values of α and β have been used in another work [51]. Like in our previous study [5], to automatically filter the ontologies using the non-interactive keyword based filtering approach, we have used the top 15, 50, and 100 most frequent terms which are extracted from the manuals. When using the topic based approach, we used the terms in each topic to filter the ontology and the evaluation was conducted separately for each topic (we expect one topic, associated with the domain, to outperform the other topic).

To carry out interactive keyword based filtering a developer has to carry out *domain keyword selection* on the top 15, 50 or 100 keywords listed by the non-interactive keyword based filtering technique. *Domain keyword selection* is a process where the developer decides whether or not to consider a keyword as a true domain keyword. To simulate this step and avoid bias, we have carried out this process fully automatically and we have matched the top listed keywords with the gold concepts and considered those keywords which are matched as true domain keywords selected by the developer. The gold concepts can be extracted from different sources of information related to the program. For this study, one of the researchers involved in the study has manually collected the gold concepts from the user manuals which come with the systems and are found on the website of the case study programs. To avoid bias, the collection was conducted prior to extracting and filtering the ontologies. To show the gold concepts collected, FileZilla and WinMerge have been chosen as an example out of the six systems analyzed in the study (see Table 3.1).

To evaluate if there is a filtering relationship between the concepts in the ontology and domain concepts (RQ1), we have defined a metric called *Gold Concepts in the Source code (GCS). Gold Concepts in the Source code (GCS)* is the ratio of gold concepts found in the ontology concepts to the total number of gold concepts. GCS shows how many of the manually

Table 3.1: Manually collected gold concepts for WinMerge and FileZilla.

Systems	Gold Concepts
FileZilla	active mode, active transfer, client, connection, current directory, current server, data, directory, download, file, file transfer, ftp, ftp client, ftp server, host, host name, ip, local directory, local file, mode, passive mode, passive transfer, password, port, protocol, remote directory, remote file, server, session, sftp, sftp client, transfer, transfer mode, upload, user
WinMerge	archive folder, binary file, block, change, character difference, compare, content, control, data, date, difference, difference block, difference text, document, file, file compare, file content, file filter, file mask, file size, file version, filter, folder, folder compare, folder difference, line, line difference, line filter, mask, merged document, multiple lines, patch, patch file, recursive compare, size, sub-folder, syncronize, text, text block, time difference, unpacker, version, version control, whitespace compare, word difference

identified gold concepts are actually present in the ontology extracted from the source code.

Table 3.2: Domain-implementation filtering confusion matrix (TP=True positive, TN=True negative, FP=False positive, FN=False negative)

		Correct Filtering	
		Domain	**Implementation**
Filtering by technique A	**Domain**	TP	FP
	Implementation	FN	TN

Filtering the domain from the implementation concepts is a classification activity. Hence, we can use the confusion matrix which is commonly used for evaluating classifiers, to measure the effectiveness of the considered filtering techniques (see Table 3.2). The effectiveness evaluation is made by comparing the resulting classification with a reference, correct domain filtering ($TP + FN$), which is produced manually. To measure the effectiveness of a filtering technique and answer RQ2, we have computed precision, recall and F-measure from the confusion matrix. The definition

of these metrics are similar to those defined in Section 2.5.1 but, here, we consider domain concepts instead of relevant source code files. The precision indicates how many of the filtered concepts are domain concepts as compared to those actually present in the ontology, while the recall shows the percentage of domain concepts in the ontology that are filtered by the technique. The harmonic mean of the precision and recall, F-measure, is used to aggregate the inversely related values of precision and recall to a single value, which simplifies comparison of the effectiveness of the filtering techniques.

To automatically carry out concept location (described in Section 2.4) and answer RQ3, we have collected bug reports which are closed and have patch files in the associated bug tracking system. In Table 3.3, we list the number of bugs which have been collected for each system. The patch files are used to identify the files which are actually changed to fix the reported bug. The names of these files are used to finally evaluate the results of our experiment. To investigate the support of the filtered ontologies to concept location, we have formulated queries, which use concepts taken from the filtered ontologies besides keywords which are deemed relevant for the concept location task and have been manually collected from the titles of each bug description. We call such queries *enhanced queries* [4] (a detailed description of enhanced queries can be found in Section 2.5.1). In this study, the enhanced queries are formulated using concepts taken from the unfiltered ontology and filtered ontology and we compared the impact of the unfiltered and filtered ontologies.

The queries are applied to the source code files. To carry out concept location, we have resorted to a very simple (yet widely used) method, namely `grep` to query the code base. In Section 2.5.1, we have also used the state-of-the-art approach, LSI, to query a code base, and we have shown that for an expert user who formulates effective queries both approaches

give similar results while for average user who formulates average queries grep is better. Hence, in this study we consider only `grep`.

The effectiveness of the queries is evaluated following the measures defined in Section 2.5.1. The most effective query is the query which gives the highest F-measure among all possible queries formulated with a maximum combination of four or less keywords and concepts. Having more than one keyword or concept in a query, in our case, represents a filtering relationship. For example if a query contains two concepts, the second concept is used to filter the query results obtained using the first concept.

3.4.2 Subjects

To conduct our study we have used six open source programs: ADempiere, FileZilla client, JEdit, OpenOffice, ThunderBird and WinMerge. The summary of the systems is shown in Table 3.3.

Table 3.3: Summary of systems.

System	Version	Files	Classes	Lines of text	No. of Bugs
ADempiere	3.1.0	1833	1917	482094	10
FileZilla	3.0.0	264	208	89080	28
JEdit	4.2	224	639	79198	12
OpenOffice	1.0.0	12761	12112	4666417	18
ThunderBird	2.0.0.0	11019	5949	3548012	12
WinMerge	2.12.2	257	146	67643	7

ADempiere[1] is an enterprise resource planning software, while FileZilla client[2] is a cross-platform, graphical FTP, FTPS, and SFTP client. JEdit[3] is a programmer's editor which provides syntax highlighting for over 2000 file formats. OpenOffice[4] is an office software suite for word processing,

[1]http://www.adempiere.com/
[2]http://filezilla-project.org/
[3]http://www.jedit.org/
[4]http://www.openoffice.org/

spreadsheets, presentations, graphics and databases. ThunderBird[5] is an email and news client developed by Mozilla foundation, while WinMerge[6] is a differencing and merging utility for Windows. In all these systems, most components are developed using the object oriented paradigm. Two of the systems, JEdit and ADempiere, are developed using Java, while the other four are developed using C++.

3.4.3 Results

RQ1: Adequacy of filtering

We have compared the concepts found in automatically extracted ontologies with gold concepts which have been collected manually and represent the domain concepts of each subject system. Table 3.4 shows the number and ratio of gold concepts which are found in the respective ontologies.

More than 50% of the gold concepts are found in NLP based and the union of NLP and structural based ontologies for all systems except ADempiere. The structural ontology contains more than 40% of the gold concepts in five of the six systems. ADempiere has the lowest percentage (around 32%) of gold concepts in the extracted ontologies for all types of ontologies. The results show that in almost all the systems more than half of the gold concepts are contained in the ontologies extracted from the systems. Hence, a filtering relationship between the extracted ontologies and the domain concepts holds. Moreover, more than half of the domain concepts can potentially be obtained from NLP or union ontologies by means of filtering.

We have also computed the ratio of the number of filtered domain ontology concepts to the total number of concepts in the ontology (see Table 3.4 last column). Ratios show that a large portion of the ontologies is related

[5]http://www.mozilla.org/en-US/thunderbird/
[6]http://winmerge.org/

Table 3.4: Gold Concepts in the Source code (GCS) and ratio of the domain ontology concepts filtered to the total number of concepts in the corresponding ontology.

System	Ontology	GCS	Domain ontology concepts ratio
ADempiere	**NLP**	0.326 (60/184)	0.011 (124/10922)
	Str.	0.326 (60/184)	0.011 (103/9294)
	NLPStr	0.364 (67/184)	0.011 (134/12346)
FileZilla	**NLP**	0.629 (22/35)	0.019 (25/1343)
	Str.	0.486 (17/35)	0.021 (19/904)
	NLPStr	0.629 (22/35)	0.017 (25/1441)
JEdit	**NLP**	0.567 (97/171)	0.063 (141/2231)
	Str.	0.509 (87/171)	0.061 (119/1943)
	NLPStr	0.579 (99/171)	0.058 (146/2535)
OpenOffice	**NLP**	0.592 (180/304)	0.016 (1165/71379)
	Str.	0.523 (159/304)	0.014 (739/51529)
	NLPStr	0.599 (182/304)	0.016 (1185/75763)
ThunderBird	**NLP**	0.523 (46/88)	0.0030 (83/31598)
	Str.	0.443 (39/88)	0.0030 (56/21727)
	NLPStr	0.523 (46/88)	0.0020 (86/34819)
WinMerge	**NLP**	0.578 (26/45)	0.018 (31/1754)
	Str.	0.4 (18/45)	0.021 (21/1008)
	NLPStr	0.578 (26/45)	0.017 (32/1891)

to implementation concepts, not to domain concepts. Hence, filtering the domain concepts from the automatically extracted concepts would result in a small-size ontology, focused on the domain information represented in the source code.

RQ2: Effectiveness of filtering techniques

Tables 3.5, 3.6 and 3.7 show the results for the non-interactive and interactive keyword based filtering techniques, when the top 15, 50 or 100 keywords are used, respectively. For each system and ontology type, we have computed the average precision, recall and F-measure achieved in filtering domain concepts. As the number of top keywords used increases, the F-measure increases in all systems for the interactive keyword based approach. The increase is observed for all types of ontologies. For the non-interactive keyword based filtering technique, however, the increase is observed only in two of the systems (JEdit and OpenOffice), for all types of ontologies.

For the interactive keyword based filtering, the best results are observed when taking the top 100 keywords, while for non-interactive filtering there is no clear pattern among the systems and ontology types. To compare the effectiveness of the two keyword based filtering techniques, we have computed the delta percentage of interactive over non-interactive keyword based filtering technique (see Tables 3.5, 3.6, and 3.7). The results show that the interactive approach has a better performance in all the systems when considering the top 100 keywords and in four of the six systems when considering the top 50 keywords. The average delta percentage improvement for the top 100 keywords across systems is about 100% for all ontology types and is significant at $\alpha = 0.05$ (see Table 3.8). The box plots shown in Figure 3.4 also confirm the superior performance of interactive over non-interactive keyword based filtering techniques for top 100

keywords.

Table 3.5: Effectiveness of keyword and interactive keyword based filtering techniques for top 15 keywords.

System	Ontology	Keyword			Int-keyword			
		P	R	F	P	R	F	Δ% (F)
ADempiere	**NLP**	0.333	0.065	0.108	0.556	0.04	0.075	-30.6
	Str.	0.333	0.049	0.085	0.571	0.039	0.073	-14.1
	NLPStr	0.333	0.06	0.101	0.556	0.037	0.07	-30.7
FileZilla	**NLP**	0.615	0.32	0.421	1	0.24	0.387	-8.1
	Str.	0.5	0.211	0.296	1	0.211	0.348	17.6
	NLPStr	0.615	0.32	0.421	1	0.24	0.387	-8.1
JEdit	**NLP**	0.6	0.085	0.149	0.875	0.05	0.094	-36.9
	Str.	0.786	0.092	0.165	0.875	0.059	0.11	-33.3
	NLPStr	0.6	0.082	0.145	0.875	0.048	0.091	-37.2
OpenOffice	**NLP**	0.769	0.034	0.066	1	0.022	0.044	-33.3
	Str.	0.838	0.042	0.08	1	0.028	0.055	-31.3
	NLPStr	0.755	0.034	0.065	1	0.022	0.043	-33.8
ThunderBird	**NLP**	0.409	0.217	0.283	0.625	0.181	0.28	-1.1
	Str.	0.483	0.25	0.329	0.667	0.214	0.324	-1.5
	NLPStr	0.4	0.209	0.275	0.625	0.174	0.273	-0.7
WinMerge	**NLP**	0.647	0.355	0.458	1	0.258	0.41	-10.5
	Str.	0.692	0.429	0.529	1	0.333	0.5	-5.5
	NLPStr	0.632	0.375	0.471	1	0.25	0.4	-15.1

Table 3.9 shows the best F-measures for topic based filtering techniques, with the corresponding parameter configuration for each system. In all systems, except ThunderBird, the best result is achieved when using the structural based ontology. For ThunderBird, the topic based filtering did not retain any concept. Of LDA and pLSI, LDA has given the best result in three of the systems while pLSI gives the best result only in one of them. Both LDA and pLSI have performed equally in the remaining two systems. The effectiveness of topic based filtering techniques is substantially lower than all non-interactive keyword and interactive keyword based filtering techniques with the exception of ADempiere, when only the top 15 keywords are used, and some of the cases of JEdit, when only the

Table 3.6: Effectiveness of keyword and interactive keyword based filtering techniques for top 50 keywords.

System	Ontology	Keyword			Int-keyword			
		P	**R**	**F**	**P**	**R**	**F**	**Δ% (F)**
ADempiere	**NLP**	0.328	0.339	0.333	0.683	0.226	0.339	1.8
	Str.	0.406	0.379	0.392	0.743	0.252	0.377	-3.8
	NLPStr	0.336	0.328	0.332	0.69	0.216	0.33	-0.6
FileZilla	**NLP**	0.288	0.68	0.405	0.75	0.48	0.585	44.4
	Str.	0.256	0.579	0.355	0.714	0.526	0.606	70.7
	NLPStr	0.283	0.68	0.4	0.75	0.48	0.585	46.3
JEdit	**NLP**	0.372	0.362	0.367	0.717	0.27	0.392	6.8
	Str.	0.452	0.353	0.396	0.762	0.269	0.398	0.5
	NLPStr	0.367	0.349	0.358	0.717	0.26	0.382	6.7
OpenOffice	**NLP**	0.432	0.158	0.231	0.594	0.103	0.176	-23.8
	Str.	0.441	0.172	0.247	0.619	0.116	0.196	-20.6
	NLPStr	0.432	0.159	0.232	0.592	0.103	0.175	-24.6
ThunderBird	**NLP**	0.193	0.446	0.269	0.571	0.337	0.424	57.6
	Str.	0.226	0.464	0.304	0.595	0.393	0.473	55.6
	NLPStr	0.189	0.43	0.262	0.571	0.326	0.415	58.4
WinMerge	**NLP**	0.168	0.516	0.254	1	0.355	0.524	106.3
	Str.	0.188	0.571	0.282	1	0.476	0.645	128.7
	NLPStr	0.17	0.531	0.258	1	0.344	0.512	98.4

Table 3.7: Effectiveness of keyword and interactive keyword based filtering techniques for top 100 keywords.

System	Ontology	Keyword			Int-keyword			
		P	R	F	P	R	F	Δ% (F)
ADempiere	**NLP**	0.209	0.589	0.308	0.627	0.419	0.502	63
	Str.	0.219	0.553	0.314	0.672	0.379	0.484	54.1
	NLPStr	0.206	0.56	0.301	0.631	0.396	0.486	61.5
FileZilla	**NLP**	0.185	0.88	0.306	0.8	0.64	0.711	132.4
	Str.	0.193	0.842	0.314	0.778	0.737	0.757	141.1
	NLPStr	0.183	0.88	0.303	0.8	0.64	0.711	134.7
JEdit	**NLP**	0.321	0.56	0.408	0.722	0.369	0.488	19.6
	Str.	0.357	0.546	0.432	0.782	0.361	0.494	14.4
	NLPStr	0.319	0.555	0.405	0.73	0.37	0.491	21.2
OpenOffice	**NLP**	0.236	0.315	0.27	0.577	0.192	0.288	6.7
	Str.	0.246	0.327	0.281	0.604	0.208	0.31	10.3
	NLPStr	0.233	0.313	0.267	0.577	0.191	0.287	7.5
ThunderBird	**NLP**	0.116	0.602	0.194	0.55	0.398	0.462	138.1
	Str.	0.152	0.696	0.249	0.614	0.482	0.54	116.9
	NLPStr	0.116	0.605	0.194	0.55	0.384	0.452	133
WinMerge	**NLP**	0.104	0.71	0.182	1	0.484	0.652	258.2
	Str.	0.101	0.667	0.176	1	0.476	0.645	266.5
	NLPStr	0.105	0.719	0.183	1	0.469	0.638	248.6

Table 3.8: Average delta percentage of interactive over non-interactive keyword based filtering technique for top 100 keywords across systems.

Ontology	Avg Δ %	P-value
NLP	103	0.03125
Str.	101	0.03125
NLPStr	101	0.03125

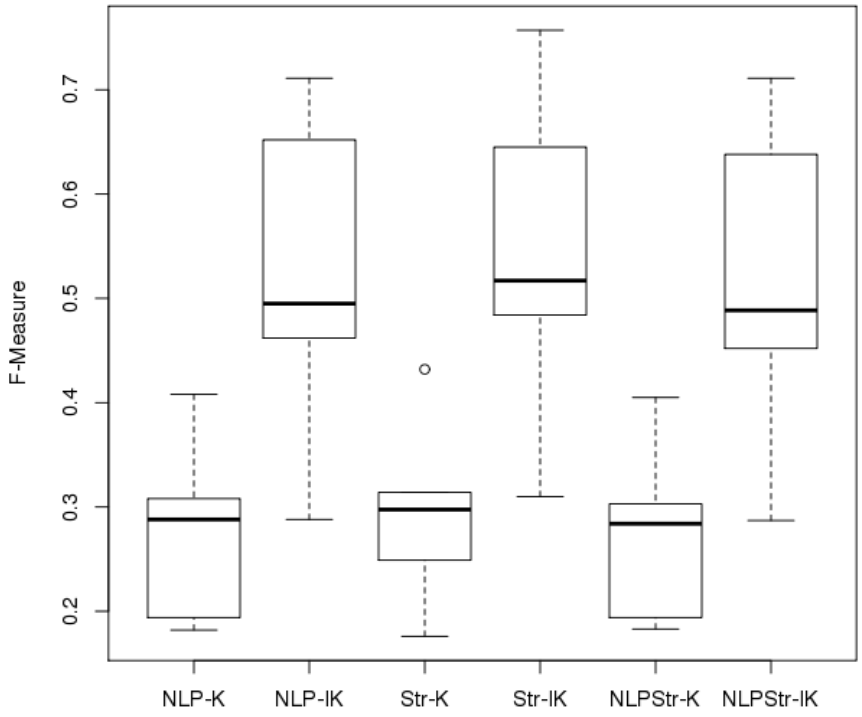

Figure 3.4: Effectiveness of non-interactive Keyword (K) and Interactive Keyword (IK) based filtering techniques across systems for top 100 keywords.

top 15 keywords are used.

Table 3.9: Topic based filtering: The highest F-measures with the corresponding ontology, filtering approach and setting combination (filtering approach, threshold, topic number).

System	Ontology	Setting combination	F
ADempiere	Str.	(LDA, 0.01, 2)	0.153
FileZilla	Str.	(LDA & pLSI, 0.02 & 0.03, 2)	0.24
JEdit	Str.	(pLSI, 0.01, 2)	0.157
OpenOffice	Str.	(LDA, 0.01, 2)	0.039
ThunderBird			-
WinMerge	Str.	(LDA, 0.02, 2)	0.25

RQ3: Support for concept location

In this research question, we investigate if the filtered ontologies, as compared to unfiltered ontologies, can be used to increase the effectiveness of programmers' queries formulated for concept location. To carry out the comparison, we have used the union of NLP and structural ontologies which in Section 2.5.2 gives better practical support for concept location. Table 3.10 shows the precision, recall and F-measure of the concept location task for keyword based and interactive keyword based filtering of the ontologies, using the top 100 keywords, as well as the corresponding delta percentages, obtained by comparing the effectiveness of filtered vs. unfiltered ontologies.

The F-measure delta percentage is negative for both types of filtering techniques. We have computed the two-sided, paired Wilcoxon signed rank test to investigate if the pattern observed is statistically significant. The p-values are significant for four of the systems. Hence we can conclude that filtering the domain concepts decreases the efficiency of the concept location task, as compared to using the unfiltered ontology.

Table 3.10: Concept location using the NLPStr ontology filtered with the top 100 keywords retrieved automatically (top) and semi-automatically (bottom) as compared to using the unfiltered NLPStr ontology. The p-values are adjusted for multiple comparison.

System	Avg. P (avg. Δ%)	Avg. R (avg. Δ%)	Avg. F (avg. Δ%)	P-value
ADempiere	0.227(-69.6)	0.867(-3.7)	0.271(-64.8)	***0.0092***
FileZilla	0.305(-44.6)	0.853(1.26)	0.358(-40.4)	***0.0005***
JEdit	0.625(-4.07)	0.903(-5.8)	0.636(-8.04)	0.3710
OpenOffice	0.0413(-90.8)	0.91(11.3)	0.0734(-83.9)	***0.0011***
ThunderBird	0.207(-36.1)	0.833(0)	0.21(-42.1)	***0.0223***
WinMerge	0.39(-43.9)	0.929(-7.14)	0.435(-42.8)	0.0591
ADempiere	0.152(-79.6)	0.9(0)	0.232(-69.9)	***0.0092***
FileZilla	0.259(-53)	0.843(0.0721)	0.307(-49)	***0.0004***
JEdit	0.433(-33.5)	0.902(-5.83)	0.482(-30.2)	0.0842
OpenOffice	0.0311(-93.1)	0.929(13.6)	0.0572(-87.5)	***0.0006***
ThunderBird	0.125(-61.4)	0.833(0)	0.13(-64.1)	***0.0223***
WinMerge	0.264(-62)	0.976(-2.43)	0.336(-55.9)	0.0591

3.4.4 Discussion

The extracted ontologies, as the program they model, are composed of domain and implementation concepts (see Table 3.4). The results show that implementation concepts constitute a large portion of the ontology. Hence, filtering out this large portion of concepts reduces the size of the ontology by a substantial amount and allows developers to focus on the domain concepts.

The reduced ontology is expected to help developers to more easily understand the domain concepts which are captured in the source code. However, if the objective of the programmer is to locate a concept, our results indicate that better effectiveness is achieved with the union of unfiltered NLP and structural based ontologies than with the filtered ontologies (see Table 3.10). The reason for this could be that in concept location not only domain concepts but also implementation concepts are required to identify

part of the source code relevant to a change request.

For filtering the domain concepts, we have investigated three IR based techniques: interactive and non-interactive keywords, and topic models. The keyword based filtering techniques have been found to be more effective in filtering domain concepts than topic based techniques. Of the two keyword based approaches, the interactive filtering technique is the most effective in all the systems when considering the top 100 keywords (see Tables 3.7 and 3.8). The keyword based techniques generate the keywords from user manuals, which use the elements of the GUI to describe how to use the functionality implemented in the programs. As a result, keywords are a mix of domain words and other words which are not specific to the domain, but are important to describe the functionality of the programs as provided through the GUI. Removing these non-domain words manually has improved the results considerably, while requiring a reasonable effort (just a few minutes). The topic based filtering techniques have given poor results. The reason for this is that most of the terms identified to describe either of the two considered topics are not necessarily domain terms, which results in reporting just few domain concepts after filtering. Hence, the recommended filtering technique to isolate domain concepts in the extracted ontologies is interactive keywords using the top 100 keywords.

3.4.5 Threats to validity

To evaluate if there is a filtering relationship between the concepts in the ontology and domain concepts and to simulate the interactive keyword filtering, we have used gold concepts which are manually collected by one of the researchers involved in the study. The gold concepts, if collected by another person and if a different resource is used, could result in a different set and the results could also be different. To address this threat, we have selected the user manuals which are a reasonable source of domain

concepts implemented in the source code. They allow to restrict the scope of domain concepts to the domain information captured in the program. To avoid personal bias during collection, the concepts have been collected prior to computing any results. Another threat can derive from the fact that the concepts used to identify neighboring concepts in formulating the enhanced queries are collected by one of the researchers involved in the study from bug report titles. To minimize this problem we have decided a-priori a standard strategy for query creation. We considered all the manually selected important terms in the bug titles and the corresponding neighboring concepts as initial query and then filtered the results.

To measure the effectiveness of queries formulated to locate concepts, we have considered precision, recall, and F-measure. However, to be more precise, the effort required from the user to process the output crucially depends on the choice of the query and on the following filtering performed by the developer. Performing this activity manually would have caused all threats to validity connected to human involvement [114], including the possibility that the subject learns how the system behaves or tends to unconsciously alter the system performance. We therefore preferred an automatic strategy, which also makes replicability easier.

The evaluation of the effectiveness of queries in locating concepts was assessed using the available bug fixing patches, as usually done in the concept location literature. However, bug fixing could require intervention only in a proper subset of all the places where the concept occurs and the number of gold positives could be larger than the one considered.

We adopted the Wilcoxon signed-rank test to test the statistical significance of our results. The tests involve multiple pair-wise comparisons. To take such multiple tests into account, we have applied the Benjamini and Hochberg (BH) [15] adjustment of the p-values.

The study was conducted using six open source systems which can limit

its generalizability. To address this threat, we have selected systems whose sizes range from medium to large and are developed using Java and C++. The systems are also from different domains and hence can be representative of other, similar systems that can be found in the open source. We feel that the approach could be effective also on commercial systems, but this should be verified by further assessment.

3.5 Conclusion

In this chapter, we have presented IR based filtering techniques to filter the domain concepts. Our results indicate that while fully automated filtering based on keywords or topics has poor performance, it is possible to highly improve it by involving the user in the selection of the relevant domain keywords. Such a user involvement requires minimal effort, since it consists just of browsing a list of 100 keywords and selecting those that are regarded as associated with the specific domain of the program under analysis. This task can be easily carried out in a few minutes. The interactive keyword based filtering technique is the most effective in all the systems, when considering the top 100 keywords. The results of filtering have confirmed our initial conjecture: ontologies are composed of both domain and implementation concepts as in the source code. Besides, results show that the vast majority of concepts in the ontologies are related to the implementation. This allows developers to easily navigate and focus on the domain concepts, once domain concept filtering has been performed.

To study the impact of filtering an ontology, we have conducted a study in the context of concept location and compared the unfiltered ontology with the filtered ontology. Results show that filtering reduces the effectiveness of queries used in concept location. Hence, while we recommend the use of filtered ontologies for understanding the domain knowledge captured

in the source code, the union of unfiltered NLP and structural ontologies is recommended when carrying out concept location tasks.

Chapter 4

Lexicon bad smells

While writing code, developers usually follow naming conventions adopted by their team or a commonly accepted standard. A study conducted on developers by Roehm *et al.* [104] indicates that consistently using such standards facilitates and simplifies program understanding. Despite their benefit, enforcing these standards and checking if they are strictly followed is difficult, especially when they refer to the semantics of identifiers.

Locating naming inconsistencies as early as possible and correcting them helps developers to maintain the quality of their code and increase its understandability. Besides, it prevents new developers or maintainers from misunderstanding the code and introducing other problems. To address this problem, we have introduced the notion of "lexicon bad smell" [2].

A "lexicon bad smell" is a concept similar to that of a "code smell" and it refers to potential lexicon construction problems, which could be solved by means of refactoring (typically renaming) actions. It is a relative notion, whose definition and use depends on the idiosyncrasies of the project, programming environment, skills of developers, etc. Hence, a bad smell in the source code of a system might not be considered a bad smell in another (*e.g.*, adding type information to identifiers may be both undesirable or desirable, depending on the IDE used and its support for type

identification).

We have defined a catalog of twelve lexicon bad smells (LBS) and created a Wiki[1] that maintains it. The catalog contains general lexicon bad smells that look at identifiers in the source code from different perspectives, such as the composing terms, their meaning, the syntactic structure and naming rules followed. We have also implemented a suite of tools for detecting the smells listed. The suite of tools is publicly available through the Wiki. Below, we present the catalog with the evaluation of the heuristics implemented by the suite of tools. A preliminary evaluation of the suite is presented in a previous paper [2]. In addition to evaluating the suite of tools, we have conducted an assessment of the effect of lexicon bad smells on a program understanding task, concept location [3]. We have also studied the role of lexicon bad smells on predicting fault prone classes [7].

4.1 Catalog

In this subsection, we present twelve lexicon bad smells. For each smell, we describe the smell's definition with its symptoms, and exceptions, if available. Examples, suggestions on how to remove the smell, and how the corresponding smell detector works are also discussed for each smell.

4.1.1 Extreme contraction

Definition. Extremely short terms are used in identifiers due to an excessive word contraction, abbreviation, or acronym.

Symptoms. Terms shorter than a threshold (*e.g.*, 2 characters) are used with a type of identifier which is intended to be self-explanatory (*e.g.*, class names, interface names, method names).

[1] http://selab.fbk.eu/LexiconBadSmellWiki/

Example. The attribute name *sz* in Figure 4.1.

```
public class Detector {
   private int sz; // sz = size
}
```

Figure 4.1: Example: Extreme contraction

Exception. This rule does not apply to prefixes introduced due to the naming conventions adopted in the system (*e.g.*, *m_* is a prefix used in the Hungarian notation to mark attributes of a class), common programming and domain terms (*e.g.*, *msg*, *SQL*, etc.), and short dictionary words (*e.g.*, *on*, *it*, etc.).

Refactoring. Rename identifiers using longer, more expressive terms.

Detector. The detector splits every class, method, and attribute identifier into its composing terms. The length of terms that are not known as exceptions is then compared with a user defined threshold and, if smaller, the terms are reported as bad smells.

4.1.2 Identifier construction rules

Definition. The naming of an identifier does not follow a standard naming convention (prefixes, suffixes, and term separators) adopted in the system.

Symptoms. Some existing naming convention (or the prevalent naming convention, if none is explicitly documented) for identifiers is not respected.

Example. The attribute name *address* of the class shown in Figure 4.2 does not follow the Hungarian naming convention (*i.e.*, it does not start with *m_*).

Refactoring.Restructure the identifier by following the adopted naming convention.

Detector. The detector collects class, attribute, and method identifiers

```
public class StudentInformation {
   private String m_name;
   private String address;

}
```

Figure 4.2: Example: Useless type

and verifies if they are constructed according to the predefined naming rules for each specific entity type.

4.1.3 Inconsistent identifier

Definition. A concept is not represented by two or more identifiers in a consistent and concise way.

Symptoms. In the absence of concept to identifier mapping, all terms of an identifier are contained in the same order in another identifier of the same type (*e.g.*, another class/method/attribute name), which is found in the same container entity (*e.g.*, package, class).

Example. In Figure 4.3 the attribute *path* is not named consistently and concisely.

```
public class Documents {
   private String absolute_path;
   private String relative_path;
   private String path;  //path is inconsistent
}
```

Figure 4.3: Example: Inconsistent identifier

Exception. This rule does not apply to class identifiers which are related by super-class sub-class relationship.

Refactoring. Renaming identifiers to make them concise and consistent.

Detector. The detector checks if an entire identifier is contained in another identifier of the same entity type (attributes or methods), inside the same container entity (a class).

4.1.4 Meaningless terms

Definition. Meaningless identifier terms, aka metasyntactic words, are used in an identifier.

Symptoms. A term from a list of known meaningless terms (*i.e.*, metasyntactic variables, common placeholder names) is used in an identifier.

Example. The method name *foo* in Figure 4.4 is meaningless.

```
public class Detector {
   public void foo() {} //foo is a metasyntactic variable
 }
```

Figure 4.4: Example: Inconsistent identifier

Refactoring. Rename identifiers using meaningful terms.

Detector. The detector checks if the terms of an identifier are in the dictionary of meaningless terms.

4.1.5 Misspelling

Definition. The words (abbreviations, contractions, and acronyms excluded) used to construct an identifier are misspelled words.

Symptoms. English (or other natural language) words are spelled incorrectly (*e.g.*, containing duplicate letters, reversed letters, etc.).

Example. The class name *Examlpe* in Figure 4.5 is not correctly spelled.

```
public class Examlpe{
   //l and p are reversed
}
```

Figure 4.5: Example: Misspelled identifier

Exception. Words which are computer science or domain specific (*e.g.*, refactoring) abbreviations and contractions.

Refactoring. Correct the misspelled words.

Detector. The detector checks all terms of an identifier which are not in the exception list and have a length greater than or equal to a user defined value using Jazzy[2], a Java open source spell checker.

4.1.6 No hyponymy/hypernymy in class hierarchies

Definition. The identifier of a child class in an inheritance hierarchy is not a hyponym of the identifier of its parent class.

Symptoms. The identifier of a class and that of its superclass are each made of a single dictionary word, but they are not related by an *is-a* relationship.

Example. The class name *Violin* in Figure 4.6 is not a hyponym of the class it extends, *Mammal*.

```
public class Mammal {
    //...
};

// Violin is not a hyponym of mammal
public class Violin extends Mammal {
    //...
};
```

Figure 4.6: Example: No hyponymy/hypernymy in class hierarchy

Exception. When class identifiers are compound words or they contain abbreviations, contractions, or acronyms, hyponymy and hypernymy can be hard to assess.

Refactoring. Refactoring may just require identifier renaming, or may involve deeper restructuring of the inheritance hierarchy.

Detector. The detector checks for an *is-a* relationship between the identifier of the class and that of its superclass, when these consist of single

[2]http://jazzy.sourceforge.net/

dictionary words. It uses *WordNet* [89, 44] to identify the relationships between the words in the class identifiers.

4.1.7 Odd grammatical structure

Definition. The grammatical structure of an identifier is not appropriate for the specific type of entity it represents.
Symptoms. A syntactical rule concerning the construction of an identifier is not respected, such as:

- class identifiers should not contain verbs;
- method identifiers should start with a verb;
- attribute identifiers should not contain verbs;
- etc. (users can define their own rules.)

Example. The class and method names in Figure 4.7 are grammatically incorrect names (the class name is a verb while the method name is a noun).

```
public class Compute { //verb
    public void initialization(); //noun
}
```

Figure 4.7: Example: Odd grammatical structure

Refactoring. The identifier should be renamed following the proper syntactic rules for the specific entity it represents.
Detector. For every class, attribute, and method identifier in a system, the detector checks if the related structuring rules are followed. It uses the *Minipar*[3] English parser to determine the parts of speech for every identifier.

[3]http://www.cs.ualberta.ca/ lindek/minipar.htm

4.1.8 Overloaded identifiers

Definition. An identifier is overloaded with multiple semantics, which might indicate the entity it represents is also overloaded with multiple responsibilities.

Symptoms. The grammatical structure of the identifier suggests overloading:

- two verbs in a method/function name;
- two nouns in a class or attribute name, none of which is used as a specifier.

Example. The method name *compute_create_document* in Figure 4.8 is composed of two verbs (*compute* and *create*) which could refer to two tasks: computing a document, and creating a document.

```
public class DocumentManager {
    //Two responsibilities: computing and creating a document
    public void compute_create_document();
}
```

Figure 4.8: Example: Overloaded identifier

Refactoring. Split the entity into two (or more) entities, each having a single responsibility and name each entity using an appropriate (non-overloaded) identifier, or name the entity with a proper name which reflects a single responsibility.

Detector. The detector for this smell checks only overloaded method identifiers, *i.e.*, it checks the number of verbs found in the phrase constructed from the terms of the identifier. *Minipar* is used to identify the parts of speech of the phrase.

4.1.9 Synonym and similar terms

Definition. Synonyms or similar terms are used to construct the identifiers representing different entities declared in the same container, such that differentiating between their responsibilities becomes difficult.

Symptoms. Two or more entities have identifiers which contain terms that are either synonyms or are very similar in form, regardless of the order in which they appear in the identifier.

Example. In Figure 4.9, the term *replicate* in the method name *isIdReplicate* is synonym to the term *copy* in method name *idCopy*. *copy* is also very similar in form to the term *cpy* of the method name *keyCpy*.

```
public class IdentifierKey {
   private String id;
   private String key;
   // replicate is synonym to copy
   public boolean isIdReplicate(String id);
   // idCopy contains Copy
   public String idCopy(String text);
   // keyCpy contains Cpy, which is very similar in form to Copy
   public String keyCpy (String text);
}
```

Figure 4.9: Example: Similar and synonym terms

Refactoring. Rename the different entities so as to differentiate their role/functionality. If necessary, introduce a common superclass or interface for the shared properties.

Detector. The detector checks synonymy and similarity of the terms used in identifiers of different entities inside the same class. The synonymy between two terms is computed using *WordNet*[4]. The similarity is computed based on the Levenshtein edit distance, and a threshold is used to filter out the terms which are not similar. For determining the synonyms, the stems

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

of the words are considered instead of the full form, in order to account for inflections.

4.1.10 Terms in wrong context

Definition. Terms that pertain to the domain of another container (*e.g.*, package) are used. This indicates that the entity named by such terms may be misplaced.

Symptoms. The terms used to name an entity in a given container are more frequently used to name entities in another container.

Example. In Figure 4.10, the class *TypeDetector* is wrongly placed in package *collections* or incorrectly named as all the other classes that refer to *detector* are in package *detectors*.

```
package collections;
class IntArray;
class TypeDetector ;

package detectors;
class MuonDetector ;
class PhosDetector ;
class HLTDetector ;
```

Figure 4.10: Example: Terms in wrong context

Refactoring. Move the misplaced entity to the container it logically belongs to or rename it to better reflect the role it has in its currently assigned container.

Detector. The detector computes the frequency and spread of terms to identify those which are prominent to a package and those crosscutting the system. The terms which are not crosscutting and are found in a package where they are not prominent are reported to be in the wrong context.

4.1.11 Useless type indication

Definition. The type of a variable is explicitly indicated in its identifier. Since modern programming environments provide easy access to type information for all variables, such an indication is often useless and provides no extra information about the role of the variable in the program.

Symptoms. An identifier contains more than one term, one of which is the identifier's type name.

Example. The term *short* in attribute name *key_short* gives redundant information about its type (see Figure 4.11).

```
public class Rental {
   private short key_short;  // type in attribute name
}
```

Figure 4.11: Example: Useless type

Exception. A static attribute used to realize the singleton design pattern has usually the same identifier as the class. Also, some naming conventions impose the use of individual characters or groups of characters which denote the type of the variable (*e.g.*, in the Hungarian notation, i is used in the identifiers of integer values).

Refactoring. Rename the identifier by removing the type name and, if necessary, rename it such that it conveys information about its role in the program.

Detector. The detector checks if attribute identifiers contain their type name.

4.1.12 Whole-part

Definition. The same term is used to represent a concept and its properties or operations.

Symptoms. A term is used to name a class and it appears also in some method or attribute identifier in the same class. This might indicate either ambiguous use of the term or redundancy.

Example. Figure 4.12 shows the ambiguous and redundant use of the concept *account*.

```
public class Account {
   private int account; //Ambiguous use
   public void computeAccount(); //Account is redundant information
}
```

Figure 4.12: Example: Whole-part

Exception. A static attribute, used to realize the singleton design pattern has usually the same identifier as the class. Constructor methods have the same name as the class.

Refactoring. Rename different entities so as to differentiate their role and/or avoid redundant information.

Detector. The detector identifies the last noun (when possible) or takes the last term of the class identifier and checks if it is used in attribute and/or method identifiers. The stems of the words are considered instead of the full form, in order to account for inflections.

4.2 Detectors

Manual inspection of the source code to identify lexicon bad smells is a tedious and difficult task. Hence, we have developed a suite of tools, called LBSDetectors[5] that automatically locate and report lexicon bad smells based on the heuristics mentioned above. To implement these heuristics, the tools use the following plug-ins and software components:

[5]http://selab.fbk.eu/LexiconBadSmellWiki/

- ***Jazzy***: is a Java open source spell checker which is based on algorithms of Aspell[6]. It uses a dictionary which is based on contents of the Ispell[7] (ver 3.1.20) word list.

- ***JAWS***[8]: is a Java API for searching the WordNet dictionary. It is used in the tool to retrieve synonyms and hyponyms of terms.

- ***PaWs***[9]: (Parser Wrappers) is a Minipar wrapper which accepts Minipars output and converts the output to XML. The detectors use the XML output of the wrapper for further analysis.

- ***Minipar***[10]: is a parser for English, which is used to identify the parts of speech of the terms composing an identifier (regarded as a phrase) [77].

- ***src2srcml***[11]: transforms the source code files into XML [35].

- ***Porter stemmer***[12]: is used to obtain the stems of terms.

- ***LCS***[13]: implements Levenshtein edit distance.

To check the source code for a specific bad smell, the tools use some thresholds (see Section 4.1) and some configuration files, including a list of known abbreviations, a list of meaningless terms, and a grammar for checking the structure of identifiers. These can be easily customized to meet specific needs and can be adapted to a particular software system using the suite's configuration file.

[6]http://aspell.net/
[7]http://fmg-www.cs.ucla.edu/geoff/ispell.html
[8]http://lyle.smu.edu/ tspell/jaws/index.html
[9]http://ontoware.org/projects/paws/ (visited on October 2009)
[10]http://webdocs.cs.ualberta.ca/ lindek/minipar.htm
[11]http://www.sdml.info/projects/srcml/
[12]http://tartarus.org/ martin/PorterStemmer/
[13]Internal tool, FBK, Trento, Italy

4.3 Evaluation

In this subsection, we present the evaluation conducted on the accuracy of the detectors developed for the bad smells defined in Section 4.1. We also present the assessment we carried out to study the impact of LBS on a program understanding task, concept location, and the approach we propose to locate fault prone classes based on the occurrence of LBS.

4.3.1 Accuracy of detectors

The detectors for bad smells use heuristics (see Section 4.1) to turn the conceptual definition of a smell into a set of operational rules, which sometimes approximate the conceptual definition. We have carried out a preliminary evaluation [2] on the accuracy of the suite of detectors implemented for some bad smells. Based on the findings of this analysis, we have improved the detectors and carried out a new assessment. In this assessment, we have considered all the detectors but two (*No hyponymy/hypernymy in class hierarchies*, and *Terms in the wrong context*), as they require domain knowledge and are quite difficult to assess.

The different detectors in the suite use different thresholds and take in to consideration the exceptions stated for the smells. Prior to running the detectors, we have defined the thresholds (3 for *extreme contractions* and 4 for *Misspelling*), and identified the exceptions and naming conventions applied in the systems we used for our study. In cases where a convention is not explicitly stated, we have considered the most common naming practices in the code as conventions [29].

To evaluate the accuracy of the suite of detectors, we have computed *precision*. *Precision* (P) is defined as the ratio between reported smells that are correct and total number of reported smells. Similar retrieval tasks also use recall to measure the effectiveness of retrieval tools. In our case, we do

not have any data about the total number of bad smells in the software, hence recall can not be computed. In order to evaluate the precision of the detectors, we have applied the following general guideline: *A reported bad smell is a false positive if developers are not expected to be willing to take any action (e.g., renaming) to improve the "smelly" identifier.* Using this guideline, two researchers involved in this study have independently evaluated 10 randomly selected lexicon bad smells for each program entity type (class, method, and attribute) on each subject and have categorized the reported smells as *true positives* or *false positives.* In cases where the number of reported smells was small, all smells have been evaluated. The evaluation result produced by each evaluator is then compared, and in cases where there is a mismatch a discussion was held to reach a consensus.

Subjects

Our evaluation was conducted on four open source systems: ADempiere, FileZilla client, OpenOffice, and WinMerge. A summary of the features of these systems is shown in Table 4.1.

Table 4.1: Features of the subject systems. The identifier count does not include constructor and destructor identifiers; overloaded method names are counted only once.

System	Version	Files	Classes	Lines of text	Identifiers count
ADempiere	3.1.0	1833	1917	482094	38241
FileZilla	3.0.0	264	208	89080	2663
OpenOffice	1.0.0	12761	12112	4666417	182258
WinMerge	2.12.2	257	146	67643	2859

ADempiere[14] is an enterprise resource planning software, while FileZilla

[14]http://www.adempiere.com/

client[15] is a cross-platform, graphical FTP, FTPS, and SFTP client. OpenOffice[16] is an office software suite for word processing, spreadsheets, presentations, graphics, and databases while WinMerge[17] is a differencing and merging utility for Windows. In all these systems, most components are developed using the object oriented paradigm. One of the systems, ADempiere, is developed using Java, while the other three are mainly developed using C++.

Results and discussion

The summary of the results on the accuracy of the detectors is shown in Table 4.3. Below we describe the results for each smell.

Extreme contraction: The extreme contractions bad smell detector was run on the systems with a threshold of 3, *i.e.*, all the terms which have 3 characters or less are considered by the detector as potential short terms. The tool has detected 2,276 (214 class, 772 attribute, and 1,290 method) identifier terms in ADempiere, 95 (31 class, 35 attribute, and 29 method) identifier terms in FileZilla, 272 (21 class, 140 attribute, and 111 method) identifier terms in WinMerge, and 4,859 (819 class, 3,467 attribute, and 573 method) identifier terms in OpenOffice as extreme contractions.

Table 4.2: Sample results of extreme contraction detector and the corresponding evaluation.

System	Entity	Identifier	Contracted term	Evaluation
ADempiere	attribute	okMailUser	ok	False Positive
ADempiere	method	setEftValutaDate	Eft	True positive
FileZilla	class	TiXmlHandle	Ti	True positive
WinMerge	method	SetMessageIDs	Ds	False Positive

[15]http://filezilla-project.org/
[16]http://www.openoffice.org/
[17]http://winmerge.org/

The precision of the extreme contractions detector for 30 randomly selected identifier terms reported as bad smells is 100% for FileZilla and OpenOffice while it is 93% for the other two systems, ADempiere and WinMerge. For ADempiere we have manually checked 10 randomly selected identifier terms reported for each entity (class, attribute, and method) while for WinMerge all class identifier terms reported (21 identifier terms), and 10 randomly selected attribute and method identifier terms have been checked. A sample of the bad smells reported is shown in Table 4.2.

The false positives reported by the detector are due to the splitting mechanism used to identify hard words in identifiers, based on camel casing (*e.g.*, *Ds* is reported as a bad smell in *SetMessageIDs* because it was separated from *I*, which is also in upper case). The other false positives are caused by the dictionary used. An example of false positive due to this is *ok* (see Table 4.2). In this case, the tool is correct in identifying *ok* as a bad smell because it is less than 4 characters and it is not considered as a dictionary word (it should have been written with all words in capital, *OK*). However, in the context of the source code, we have considered it to be a false positive following our guidelines.

Identifier construction rules: We ran this detector using a set of identifier construction rules as input, which are defined separately for each system, based on the naming conventions adopted in each of them. For OpenOffice-class and method identifier, and ADempiere-method identifier we were not able to find identifier construction rules defined by the developers or a pattern adopted in the majority of the respective entity identifiers. Hence, the corresponding precision values are not computed.

The number of violations identified by the detector are 8,960 (814 class, and 8,146 attribute) identifiers in ADempiere, 214 (16 class, 177 attribute, and 21 method) identifiers in FileZilla, 2,207 attribute identifiers in OpenOffice, and 190 (80 class, 72 attribute, and 38 method) identifiers in Win-

Table 4.3: Accuracy of detectors (NA=not applicable, NC=not computed).

Lexicon bad smell	System	Precision (%)	Reported LBS count			
			Class	Attribute	Method	Total
Extreme contraction	ADempiere	(28/30)93.3	214	772	1290	2276
	FileZilla	(30/30)100	31	35	29	95
	OpenOffice	(30/30)100	819	3467	573	4859
	WinMerge	(38/41)92.7	21	140	111	272
Identifier construction rules	ADempiere	(20/20)100	814	8146	NC	8960
	FileZilla	(30/30)100	16	177	21	214
	OpenOffice	(10/10)100	NC	2207	NC	2207
	WinMerge	(30/30)100	80	72	38	190
Inconsistent identifier	ADempiere	(20/20)100	NA	223	1917	2140
	FileZilla	(17/20)85	NA	44	253	297
	OpenOffice	(19/20)95	NA	194	528	722
	WinMerge	(20/20)100	NA	98	264	362
Meaningless terms	ADempiere	(2/2)100	1	1	0	2
	FileZilla	-	0	0	0	0
	OpenOffice	(6/6)100	1	1	4	6
	WinMerge	-	0	0	0	0
Misspelling	ADempiere	(30/30)100	58	278	1224	1560
	FileZilla	(30/33)90.9	13	34	54	101
	OpenOffice	(30/30)100	796	626	813	2235
	WinMerge	(28/28)100	8	84	184	276
Odd grammatical structure	ADempiere	(12/30)40	642	3075	7399	11116
	FileZilla	(14/30)46.67	119	228	763	1110
	OpenOffice	(8/30)26.67	1123	2293	3367	6783
	WinMerge	(4/30)13.3	86	262	1124	1472
Overloaded identifiers	ADempiere	(7/10)70	NA	NA	552	552
	FileZilla	(10/21)47.62	NA	NA	21	21
	OpenOffice	(7/10)70	NA	NA	139	139
	WinMerge	(4/10)40	NA	NA	63	63
Synonym and similar terms	ADempiere	(8/20)40	NA	1382	17397	18779
	FileZilla	(12/20)60	NA	59	704	763
	OpenOffice	(5/20)25	NA	282	3174	3456
	WinMerge	(11/20)55	NA	35	2087	2122
Useless types	ADempiere	(10/10)100	0	596	0	596
	FileZilla	(20/21)95.23	NA	21	NA	21
	OpenOffice	(10/10)100	NA	283	NA	283
	WinMerge	(5/5)100	NA	5	NA	5
Whole-part	ADempiere	(19/20)95	NA	480	2681	3161
	FileZilla	(17/20)85	NA	55	77	132
	OpenOffice	(18/20)90	NA	224	384	608
	WinMerge	(18/20)90	NA	59	230	289

Merge.

We manually evaluated 10 randomly selected bad smells from each entity type of each system, and found that all selected bad smells are correct. The precision of the detector for all systems is 100%. An example of the violations reported are shown in Table 4.4.

Table 4.4: Examples of identifier construction LBS.

System	Entity	Identifier	Violation	Evaluation
FileZilla	attribute	output_text_special	Does not start with m_p, m_b, m_n, m_cb, b, n, p, c, or m_	True positive
FileZilla	method	clear	Does not start with a capital letter	True positive
WinMerge	class	ListEntry	Does not start with C or I	True positive
WinMerge	method	remove_prefix	Does not start with a capital letter	True positive

Inconsistent identifier: The detector for this bad smell has identified 2,140 bad smells (223 in attribute identifiers and 1,917 in method identifiers) in ADempiere, 297 bad smells (44 in attribute identifiers and 253 in method identifiers) in FileZilla, 722 bad smells (194 in attribute identifiers and 528 in method identifiers) in OpenOffice, and 362 bad smells (98 in attribute identifiers and 264 in method identifiers) in WinMerge.

To compute the precision of this detector, we have randomly selected 20 entries (10 attribute identifiers and 10 method identifiers) for each system. The evaluation of the samples indicates a precision of 100% for ADempiere and WinMerge, 85% for FileZilla, and 95% for OpenOffice.

Table 4.5: Examples of inconsistent identifier LBS.

System	Entity	Identifier1	Identifier2	Class	Evaluation
ADempiere	method	test	***test***Port	ConfigurationData	True positive
FileZilla	attribute	time	has***Time***	Direntry	False positive
OpenOffice	attribute	Value	Has***Value***	XMLTableCellContext_Impl	False positive
OpenOffice	method	get	***get***Type	IdlFieldAdapter_Impl	True positive

Samples of the inconsistent identifier LBS identified by the corresponding detector are shown in Table 4.5. All of the false positives reported in this case are due to the detectors inability to identify the cases when a boolean typed identifier starting with a verb contains entirely another one. For example, the attribute *Value* in the OpenOffice class *XMLTableCellContext_Impl* is contained in the attribute *HasValue* which is boolean (See Table 4.5).

Meaningless terms: The detector of this bad smell checks for the occurrence of 65 metasyntactic words in the identifiers. It has identified 2 occurrences (1 in class and 1 in attribute identifiers) of the metasyntactic words in ADempiere while 6 occurrences (1 in class, 1 in attribute, and 4 in method identifiers) in OpenOffice. In the remaining two systems no meaningless term is found. The meaningless term identified in ADempiere and OpenOffice is *var*. The accuracy of the detector on the two systems in which it identified the LBS is 100%.

Misspelled words: This detector is meant to identify all terms which are not in the dictionary and have a length greater than or equal to a threshold, which in our study is set to 4. The detector has identified 1,560 misspelled words (58 in class identifiers, 278 in attribute identifiers, and 1,224 in method identifiers) in ADempiere, 101 misspelled words (13 in class identifier, 34 in attribute identifiers, and 54 in method identifiers) in FileZilla, 2,235 misspelled words (796 in class identifiers, 626 in attribute identifiers, and 813 in method identifiers) in OpenOffice, and 276 misspelled words (8 in class identifiers, 84 in attribute identifiers, and 184 in method identifiers) in WinMerge.

To manually analyze the results, we have randomly selected 30 (10 class identifiers, 10 attribute identifiers, and 10 method identifiers) records reported for ADempiere and OpenOffice. For FileZilla and WinMerge, we have considered all bad smells reported for class identifier which are 13 and

8, respectively, and 10 randomly selected bad smells from attribute identifiers and method identifiers. The precision of the detector for ADempiere, OpenOffice, and WinMerge is 100% while for FileZilla it is 90%. Examples of the violations reported are shown in Table 4.6. The false positives in FileZilla are due to the tools inability to differentiate between terms commonly used in programming (*e.g.*, *initialize*) and misspellings.

Table 4.6: Example results of misspelling LBS detector.

System	**Entity**	**Misspelled word**	**Identifier**	**Evaluation**
FileZilla	class	Combo	ComboBoxEx	False positive
FileZilla	method	Initialize	Initialize	False positive
OpenOffice	attribute	Droenk	lDroenk	True positive
WinMerge	class	Outputter	CompilerOutputter	True positive

Odd grammatical structure: The detector for this bad smell has identified 11,116 (642 in class identifiers, 3,075 in attribute identifiers, and 7,399 in method identifiers) bad smells in ADempiere, 1,110 (119 in class identifiers, 228 in attribute identifiers, and 763 in method identifiers) in FileZilla, 6,783 (1,123 in class identifiers, 2293 in attribute identifiers, and 3,367 in method identifiers) in OpenOffice, and 1,472 (86 in class, 262 in attribute, and 1,124 in method identifiers) in WinMerge. The precision obtained after manual investigation of the results for 30 randomly selected entries (10 class identifiers, 10 attribute identifiers, and 10 method identifiers) in ADempiere and FileZilla is 40% and 46.67%, respectively, while it is 26.67% and 13.3% in OpenOffice and WinMerge, respectively. Samples of the results with the related evaluation are shown in Table 4.7.

The false positives are mainly due to two reasons. First, the detector relies on the output of Minipar to identify the parts of speech of the terms in the identifier. For example the parser has identified word *Floating* in the class name *MenuFloatingWindow* as a verb while it is used as a spec-

Table 4.7: Example results of odd grammatical structure detector.

System	Entity	Identifier	Evaluation
ADempiere	attribute	ConfirmType	True positive
FileZilla	method	GetAllImages	False positive
OpenOffice	attribute	VerifyMode	True positive
OpenOffice	class	MenuFloatingWindow	False positive

ifier (see Table 4.7). The second main reason for the false positives is the grammar used. The grammar used by the detector assumes that class and attribute names are constructed from adjectives and nouns, while method names are constructed by verbs only or verbs followed by adjectives and nouns. In some identifiers however different types of words are used, which result in false positives (*e.g.*, *All* in the FileZilla method name *GetAllImages*, see Table 4.7).

Overloaded identifiers: The overloaded identifiers lexicon bad smell detector has identified 552 bad smells in method identifiers in ADempiere, 21 bad smells in FileZilla, 139 bad smells in OpenOffice, and 63 bad smells in WinMerge. To evaluate if the method names actually imply two or more functionalities, we have randomly selected 10 reported method identifiers from all systems except for FileZilla and manually investigated the corresponding method implementations. For FileZilla the number of overloaded identifiers reported is small (21), and hence we have checked all of them. The precisions for ADempiere, FileZilla, OpenOffice, and WinMerge are 70%, 47.62%, 70%, and 40%, respectively. Sample results for the overloaded identifier detector are shown in Table 4.8.

The false positives reported are mainly due to specifiers of nouns which are considered as verbs by Minipar. For example *Encoding* in the method name *GetEncodingType* is considered as a verb while it is used as a specifier of *Type*. The other category of false positives is observed in get and set

Table 4.8: Example results of overloaded identifiers detector.

System	Identifier	Verb count	Evaluation
ADempiere	getM_AttributeSetInstance_ID	2	True positive
FileZilla	GetEncodingType	2	False positive
OpenOffice	METSetAndPushLineInfo	2	True positive
WinMerge	GetTimeoutDisabled	2	False positive

methods of boolean type attributes (*e.g.*, see *GetTimeoutDisabled* in Table 4.8). Boolean attributes usually have verbs in their names, and hence the corresponding get and set methods will have two verbs.

Synonym and similar terms: The detector for synonyms and similar terms compares the attribute and method identifiers to attribute, method and class identifiers inside the same class. In the case of similar terms, it takes as input a parameter, which specifies the minimum Levenshtein distance between two terms in order to decide if they are similar. We used 90% as the minimum Levenshtein distance for reporting two terms as similar. In the case of synonyms, it uses WordNet synsets to consider two terms as synonym. A term is considered synonym if it is found in the synsets of the other term. The detector has identified 18,779 bad smells (1,382 in attribute identifiers, and 17,397 in method identifiers) in ADempiere, 763 bad smells (59 in attribute identifiers, and 704 in method identifiers) in FileZilla, 3,456 bad smells (282 in attribute identifier, and 3,174 in method identifiers) in OpenOffice, and 2,122 bad smells (35 in attribute identifiers, and 2,087 in method identifiers) in WinMerge.

The evaluation of the precision was conducted by randomly selecting 20 entries from the bad smell list (10 in attribute identifiers and 10 in method identifiers) for each system. The precision of this detector is 40% in ADempiere, 60% in FileZilla, 25% in OpenOffice, and 55% in WinMerge. The detector used WordNet synsets for its judgment of synonymy. However, the

semantics and relationship between two words in the source code turned out to be, in some cases, different than those found in WordNet. Often, words have a specific meaning in the source code, which make them semantically distinct and not similar to the corresponding WordNet synonyms (*e.g.*, see *Draw* and *Get* in Table 4.9).

Table 4.9: Example results of synonym and similar terms LBS detector.

System	**Term ∼in∼ entity *identifier***	**Synonym or similar to**	**Term ∼in∼ entity *identifier***	**Evaluation**
ADempiere	PAYMENTRULEPO ∼in∼ attribute ***PAYMENTRULEPO_****DirectDebit*	similar to, edit dist. =91	PAYMENTRULE ∼in∼ attribute ***PAYMENTRULE_****DirectDeposit*	True positive
FileZilla	Response∼in∼ method *Connect*Parse***Response***	Synonym to	Reply ∼in∼ method *Process***Reply**	True positive
OpenOffice	get ∼in∼ method ***get****ExactName*	Synonym to	has ∼in∼ method ***has****ByName*	False positive
WinMerge	Draw ∼in∼ method *Can***Draw***3DImageList*	Synonym to	Get ∼in∼ method ***Get****MenuDrawMode*	False positive

Useless type indication: The useless type indication detector has identified 596, 21, 283, and 5 lexicon bad smells in the attribute identifiers of ADempiere, FileZilla, OpenOffice, and WinMerge. We manually checked 10 randomly selected reported bad smells from ADempiere and OpenOffice while all reported bad smells of FileZilla and WinMerge have been checked. The detector has identified all the bad smells manually checked for ADempiere, OpenOffice, and WinMerge correctly and hence has a precision of 100%. While manually checking the bad smells reported for FileZilla, the evaluators have identified all bad smells correct but one, *pIOThread*, which uses a commonly used coding style, naming the identifier with the same name as the type and a prefix probably indicating the type of the identifier (p). Hence, the precision of the detector for FileZilla is 95% (see Table 4.10).

Table 4.10: Example results of useless types LBS detector.

System	Attribute identifier	Attribute data type	Evaluation
ADempiere	headerBG_Color	Color	True positive
FileZilla	pIOThread	IOThread	False positive
OpenOffice	UserImageList	ImageList	True positive
OpenOffice	maDropTimer	Timer	True positive

Whole-part: The detector for this LBS has identified 3,161 lexicon bad smells (480 in attribute identifiers, and 2,681 in method identifiers) in ADempiere, 132 lexicon bad smells (55 in attribute identifiers, and 77 in method identifiers) in FileZilla, 608 lexicon bad smells (224 in attribute identifiers, and 384 in method identifiers) in OpenOffice, and 289 lexicon bad smells (59 in attribute identifiers, and 230 in method identifiers) in WinMerge. Examples of whole-part LBS are shown in Table 4.11.

Table 4.11: Example results of whole-part LBS detector.

System	Entity	Member identifier	Class identifier	Evaluation
ADempiere	method	PaymentRule	Payment	True positive
OpenOffice	method	_setPropertyValues	DocumentSettings	False positive
OpenOffice	attribute	OriginalRequest	Request	True positive
WinMerge	attribute	xml	XmlUniformiser	False positive

From the reported lexicon bad smells of each system we have randomly selected 20 bad smells (10 from attribute identifiers, and 10 from method identifiers) and manually evaluated them. The precision of the detector for ADempiere, FileZilla, OpenOffice, and WinMerge is 95%, 85%, 90%, and 90%, respectively. The false positives are due to specifiers which are incorrectly identified as the main concepts represented by the class (*e.g.*, *xml* in class identifier *XmlUniformiser*, see Table 4.11), and due to stemming (*e.g.*, *Settings* in class identifier *DocumentSettings* is stemmed to *set*).

Threats to validity

What is considered a bad smell in one system might not be a bad smell in another, due to the specifics of every system. Also, what one person identifies as bad smell might not be interpreted as such by another person. We handle these threats by using configurable files for the particular settings used by our tools and by having the results verified by two researchers.

The case study can easily be replicated using the detectors and source code of the systems, all available online. However, as the number of some of the reported bad smells was large, we manually evaluated a sample of the results. Even though such a sample was chosen randomly, a different choice might have produced different values of precision.

4.3.2 Effect of lexicon bad smells on concept location

Identifiers play an important role during program understanding and maintenance activities as they are usually used to communicate the intention of a program entity and relate domain concepts to their representation in the code. Hence, we conjecture that having low quality identifiers impacts these activities. We consider a low quality identifier as one having numerous LBS, and high quality identifier one that has few to none such smells.

To validate our conjecture, we have conducted a case study on the effect of LBS on one of the program comprehension tasks, concept location, using reenactment in before-and-after studies [3]. The case study was carried out using two open source systems. We performed concept location using two different Information Retrieval (IR) techniques, before and after the bad smells are identified and removed from the code, and we have compared the outcomes.

Having real developers performing concept location on software systems

before and after removing the lexicon bad smells has several problems, which make it very difficult to realize in practice. First, it requires a lot of time and effort and finding developers willing to invest the time needed to perform the changes can be very difficult. Second, it also introduces an additional variable, *i.e.*, the developer, which is very difficult to control, and can, thus, introduce bias in the study. Hence, as done in similar studies such as Gay *et al.* [47], we resorted to an approach which uses historical data and IR techniques to locate concepts. The historical data is used to determine the outcome of a task that was performed in the past. Using bug tracking systems and versioning systems, we can find out what parts of the code were changed in response to a bug fix request. In the context of concept location, we call these *target changes* (*target classes*, *target methods*, etc.), as they are the targets of the concept location task.

For our study, we have collected target classes from the patches that often accompany bug reports in the bug tracking system, or from commits to the versioning system hosting the source code. In this second case, the target classes are located by identifying the bug ID in the commit messages found in the source code versioning system and then analyzing the changes that occurred in those commits. The bug reports are filtered such that the commits include only one bug ID in the commit message. Besides the target classes, we have collected the bug description for which the target classes are changed.

To automate the reenactment process we have used two concept location tools that implement IR techniques (see Section 2.4), Latent Semantic Indexing (LSI) [83] and an improved version of the Vector Space Model implemented in Lucene[18]. We used these two different IR techniques in order to observe if the effect of the lexicon bad smells on concept location depends on the IR engine used or not. LSI takes the number of dimensions

[18]http://lucene.apache.org/

to which the vector space should be reduced during the Singular Value Decomposition (SVD) and the weight to be used when scoring documents. In this study, we used 100 dimensions for SVD [38] and TF-IDF (term frequency inverse document frequency) as the weight [47]. The focus of this study was to observe the difference in performance when lexicon bad smells are present versus when they are absent, given that all the other variables are fixed, including those used for LSI. Thus, it is of less importance which dimension or weight is used, as long as they are the same before and after refactoring the bad smells.

Both IR techniques use a corpus generated from the original source code of the systems or refactored code. A document in the corpus corresponds to a class in the source code. It is constructed from the terms composing the identifiers and the original identifiers. To identify the terms composing an identifier, we have used common naming conventions such as underscore and camel casing. For example, "setValue", "set_value", "SETValue", etc. are all split to "set" and "value". The original identifiers are kept in the corpus, in order to account for any identifiers that might be included in a query. Filtering is also used to eliminate programming language specific keywords (Java and C++), and also common English stop words[19].

The queries for which we computed the ranks are formulated from the change request. The formulation of the query can be done by the developer who analyzes the change request and select terms which she considers to be relevant. In this case, we have followed the simplest approach and the whole change request as the query. The query is processed in the same way as the corpus, *i.e.*, by splitting any identifier present in the change request and by filtering out programming keywords and common English stop words.

In this study, we focused on eight of the twelve types of bad smells

[19]http://www.webconfs.com/stop-words.php

described in Section 4.1, for which we could get enough information from the artifacts of the systems to detect the smells and suggest new names. For the other four smell types, we did not have enough information about the systems in order to provide a reliable renaming of the identifiers. The lexicon bad smells we focused on in this study are *extreme contraction, inconsistent identifier, meaningless terms, misspelling, odd grammatical structure, overloaded identifiers, useless type indication,* and *whole-part*. To detect these lexicon bad smell, we have used the tool *LBSDetectors* which is described in Section 4.2.

To refactor the lexicon bad smells detected, two researchers involved in the study have independently proposed names and then compared their suggestions. In the cases where the names proposed by the two researchers were different, there was a discussion about the two names and an agreement was reached. All renamings were performed across the entire system. At the same time, if the bad smells occurred in the bug titles and descriptions, they were renamed also there.

In studies that use IR-based concept location, it is common to use the ranks of the target classes as effort measures [47]. These represent the number of classes a developer would have to look at before finding the target classes if she would analyze the list of results in the order provided. Clearly this is an approximate measure as it considers that the effort to investigate a class is identical for all classes, which may not be the case in real-life scenarios. However, the measure is consistently applied in each treatment. The impact of the lexicon bad smells on IR-based concept location can be, therefore, measured using the difference between the ranks of the target classes before and after the refactoring. If these ranks improve (*i.e.*, less effort is required to locate the classes) then we can conclude that removing the lexicon bad smells improves IR-based concept location.

Subjects

The two software systems used in the study are FileZilla Client 3.0.0, an open source, cross-platform, graphical FTP, FTPS, and SFTP client and OpenOffice 1.0.0, a well-known open source office software suite for word processing, spreadsheets, presentations, databases, etc. FileZilla is medium-sized, having 208 classes, while OpenOffice is a large system, with 12,761 classes. Both are implemented mostly in C++.

Both systems have online bug tracking systems, from which we extracted a set of 29 bugs for FileZilla and 19 for OpenOffice. The bugs are selected such that the target classes could be identified, either from the patches that sometimes accompany the bug reports in the bug tracking system, or from commits to the versioning system hosting the source code.

The corpus of the two systems is extracted both before and after the lexicon bad smells are fixed, and identifiers were split, keeping the originals. Also, filtering is applied in order to remove common English terms and C++ keywords.

We have used the concatenation of the title and the description of the bugs, as retrieved from the bug reports, as the queries for IR-based concept location. We applied the same processing as for the corpus (*i.e.*, splitting and filtering) on the queries.

Refactoring

We have used the *LBSDetectors* to identify lexicon bad smells found in the target classes identified for the set of bugs selected in the two systems. We have found 192 identifiers containing at least one bad smell in the 28 unique target classes in FileZilla and 775 identifiers for the 26 unique target classes in OpenOffice (see Table 4.12). Among the eight lexicon bad smells, the number of identifiers found to contain odd grammatical

structure, extreme contraction, misspelling and inconsistent identifiers was high, while the number of meaningless terms was low in both systems (see Table 4.12).

Table 4.12: Number of identifiers containing bad smells in the target classes and number of refactored identifier occurrences in FileZilla and OpenOffice

Lexicon bad smell	FileZilla	OpenOffice
Extreme contraction	86	480
Inconsistent identifier use	95	74
Meaningless terms	0	1
Misspelling	64	436
Odd grammatical structure	147	434
Overloaded identifiers	4	12
Useless type indication	2	7
Whole-part	13	25
Number of identifiers containing bad smells in target classes	192	775
Number of identifier occurrences refactored in the system	2,216	90,749
Number of unique target classes	28	26

As mentioned before, the bad smell correction was done manually for each identifier. For example, the method name *command* is identified as an odd grammatical structure bad smell because it does not contain a verb. It is renamed to *executeCommand* after consulting the comment of the method in the source code of FileZilla. Not all identified bad smells are corrected, though. For example, the extreme contractions which are due to the use of the Hungarian notation are not changed, but other extreme contractions, such as, *Lev* (refactored to *Levenshtein*) and *Exc* (refactored to *Excel*), are refactored. Inconsistent identifiers, detected in large numbers in FileZilla, refer generally to method identifiers which are included within other method names in the same class, thus making their meaning not specific enough. An example of such a method name is *Connect*, which

appeared also in the name of another method, *ConnectToClient* inside the same class in FileZilla. It is thus unclear how the two methods are different and to what exactly *Connect* refers to. In consequence, based on the body of the method and the comments, we renamed the identifier to *ConnectToServer*, which is more specific and reflects the functionality of the method better. For the misspellings bad smell, a high number of bad smelling identifiers are reported in OpenOffice. One example of such an identifier is *isApplyable*, which is renamed to *isApplicable*. There are almost no identifiers containing meaningless terms in either system, as the terms in our predefined list of metasyntactic variable names did not occur in the target classes. The only exception is *var*, which occurred in one target class.

While suggesting the new names the actions listed in Table 4.13 are performed. The most frequent action is term expansion, where extremely contracted terms are expanded to the terms they are referring to (*e.g.*, *nTrot* is expanded to *nTextRotation*). Other frequent actions were addition and deletion. Addition included adding missing verbs to method names or replacing a term with a meaningful one. In OpenOffice, a few identifiers that contained German terms are also encountered. These terms are replaced with their English translation (example: *importGraf* is renamed to *importGraphic*).

The total number of occurrences of bad smelling identifiers changed in the whole system is 2,216 for FileZilla and 90,749 for OpenOffice (see Table 4.12).

Results and discussion

For each bug in the two systems we reenacted concept location using the title and description of the bug as the query. We simulated the user by using this initial query and no subsequent query reformulations. We as-

Table 4.13: Types of actions performed to fix the lexicon bad smells and the corresponding number of identifiers on which they are applied.

Type of action while correcting a smell	OpenOffice	FileZilla
Term expansion	484	38
Spelling correction	2	0
Term reordering	35	31
Added term	283	71
Deleted term	139	42
Replaced term	138	37
Language translation	33	0

sumed the users would inspect the classes in the order suggested by the tool. For each bug we performed four reenactments: two before the bad smells removal, using LSI and Lucene, respectively, and two after. In each run we recorded the effort measures, represented by the rank of the target classes in the list of search results. The measures for FileZilla and OpenOffice are reported in Table 4.14 and Table 4.15, respectively. We present the results for each of the two systems separately and then discuss the differences between them.

For FileZilla, when using Lucene, out of the 45 non-unique target classes for the 29 bug reports selected, 21 had the same rank in the list of search results before and after the refactoring. At the same time, the results for 13 target classes were worse after refactoring. For the remaining 11 classes the results were better after refactoring the lexicon bad smells. Although there were more target classes for which the ranking did not improve, the overall ranking of the target classes slightly improved after the bad smells were removed (see Table 4.16).

The absolute difference between the ranks of the target classes before and after refactoring, which is the sum of all the individual differences for each target class, is 14, and the average difference is 0.31. This indicates

Table 4.14: The rank of changed classes in the list of search results when using LSI and Lucene on the original and refactored FileZilla source code.

No.	**Bug ID**	**LSI**		**Lucene**	
		Before	***After***	***Before***	***After***
1	1299	68, 54	65, 56	24, 17	26, 19
2	3023	36	37	20	19
3	3198	51, 2, 84	53, 3, 86	4, 1, 2	5, 1, 2
4	3220	68, 39, 17, 45	67, 40, 17, 45	65, 16, 11, 1	63, 16, 11, 1
5	3230	33	32	1	1
6	3232	2	1	4	2
7	3235	178, 52, 13	91, 50, 10,	130, 85, 64	101, 86, 65
8	3239	28	50	1	1
9	3252	6	9	8	6
10	3270	52	52	2	2
11	3272	84	81	16	17
12	3278	72	80	2	3
13	3284	91	86	8	8
14	3287	51	52	2	2
15	3307	68	67	2	2
16	3308	21	21	4	2
17	3319	124	122	25	24
18	3323	3	3	2	2
19	3334	64	62	7	6
20	3341	66	61	7	7
21	3343	21	19	4	4
22	3344	57	54	5	5
23	3345	3	5	4	6
24	3348	26	52	3	5
25	3356	53, 34, 47, 25, 68, 67, 116	51, 45, 41, 25, 68, 66, 117	11, 27, 22, 24, 1, 10, 12	12, 33, 23, 20, 1, 9, 10
26	3372	24	50	1	1
27	3373	51, 101	55, 99	1, 55	1, 67
28	3397	17, 25	17, 25	2, 3	2, 3
29	3403	42	67	1	1

Table 4.15: The rank of changed classes in the list of search results when using LSI and Lucene on the original and refactored OpenOffice source code.

No.	**Bug ID**	**LSI**		**Lucene**	
		Before	***After***	***Before***	***After***
1	4378	691	453	1174	29
2	5923	49	51	48	47
3	6906	1894	1671	48	44
4	7114	7366	7932	304	5
5	7868	6540	6919	95	79
6	8148	3531	3669	177	44
7	8426	5222, 2814, 2613	4039, 2169, 1349	2, 21, 57	1, 11, 4
8	8640	126	79	29	12
9	8755	3222	3489	434	434
10	8779	120	142	1146	1220
11	9391	431, 7102	639, 10749	2, 3	2, 5
12	9959	2185, 3132	1347, 3300	32, 8	22, 12
13	10424	1380	1141	780	798
14	10532	7199, 1621	6323, 1165	1757, 535	1766, 278
15	10828	915	747	1	1
16	10995	4560, 1152, 40	4029, 1193, 50	444, 277, 2298	57, 11, 126
17	11776	7279, 4357	6346, 3911	82, 18	10, 13
18	17620	2023, 9093, 9292, 4112,10058, 2099, 5250	1181, 3922, 8533, 4792, 10137, 1999, 5541	1, 609, 8260, 2, 987, 14, 790	1, 195, 6599, 2, 970, 11, 534
19	101603	583	729	274	85

Table 4.16: Summary statistics for the rank delta in FileZilla and OpenOffice. A positive delta indicates improved rank after bad smell fixing.

Statistics	**FileZilla**		**OpenOffice**	
	LSI	**Lucene**	**LSI**	**Lucene**
Absolute rank delta	-6	14	8315	7281
Average delta (std dev)	-0.13 (15.4)	+0.31 (4.9)	251.97 (1212.9)	220.64 (495.7)
Average positive delta	6.95	4.27	831.06	321.22
Maximum positive delta	87	29	5171	2172
Average negative delta	-8.12	-2.53	-443.93	-21.4
Minimum negative delta	-26	-12	-3647	-74
Median delta	0	0	100	10
Delta p-value	1	0.9884	0.0879	0.0004

an overall improvement of 14 positions in the list of ranked results over all target classes after the lexicon bad smells were removed, with an average improvement of 0.31 positions for each target class. The distribution of the deltas can be seen in the histogram presented in Figure 4.13a. The overall improvement is due to the fact that a few classes had a significant improvement in the rank, which overcame the decrease in other target classes (see Figure 4.13a). In fact, the average (4.27) and maximum (29) positive deltas were higher than the average (-2.53) and minimum (-12) negative deltas.

In order to see if the difference between results before and after the refactoring is statistically significant, we performed a two-tailed, paired Wilcoxon signed rank test between the two series of data. The p-value of 0.988 indicates that there is no statistical proof that the refactoring had an effect on the ranks of the target classes.

When using LSI for FileZilla, there were 9 cases in which the ranking of the target classes was the same before and after refactoring, 17 cases in which the ranking was worse and 19 where the results were better after refactoring. However, the absolute delta between ranks before and after

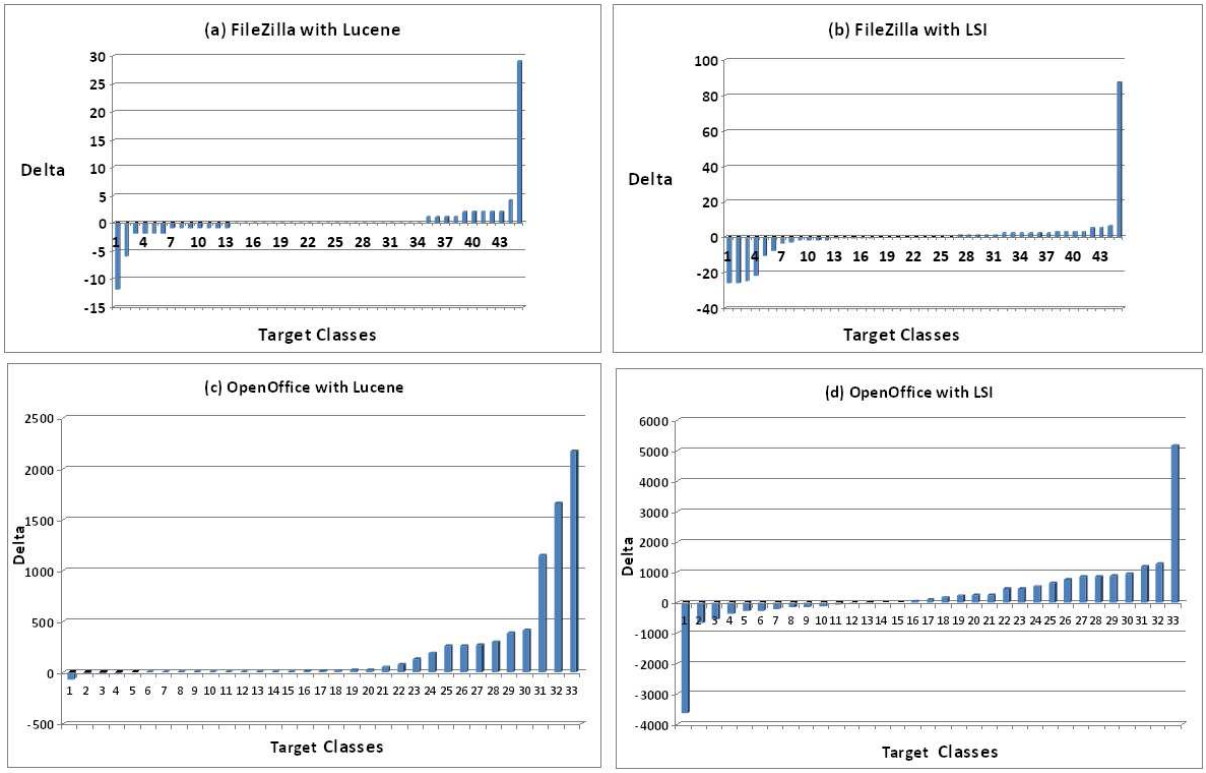

Figure 4.13: Histogram of deltas for FileZilla and OpenOffice.

refactoring was -6, indicating a slight decrease in the results (-0.13 positions per target class). This time, even though the maximum delta value was an improvement of 87 positions (see Figure 4.13b), the average rank decrease (-8.12) was higher than the average improvement (6.95).

The results for FileZilla indicate that removing the lexicon bad smells has little impact overall, when considering the effect on all target classes. However, the effect on the ranking was significant in the case of some target classes, which registered an improvement in rank of almost 50% (first target class in bug 3235, see Table 4.14). At the same time, the study suggests that the IR technique used might have a small impact on the overall difference in ranks before and after refactoring for FileZilla. The difference between absolute deltas for Lucene and LSI was 20, with an average of 0.44 delta per target class. Although the delta in ranks before

and after refactoring was not greatly affected by the IR technique used, Lucene generally placed the target classes higher than LSI in the list of search results, by an average of 34 ranks both before and after refactoring.

For OpenOffice, the results were very different than those obtained for FileZilla. The results after refactoring were significantly better than the results before the refactoring was performed, for both IR techniques.

For Lucene, there were 23 cases out of the 33 non-unique target classes for which the results after refactoring were better than before, 5 classes for which the results were better before refactoring, and 5 classes which had no change in rank (see Table 4.15). The distribution of the deltas for all 33 non-unique target classes for the 19 bug reports can be found in Figure 4.13c. The absolute delta between the ranks of the target classes before and after refactoring was 7,281 (see Table 4.16), with an average delta of 221. This means that after refactoring, the target classes were ranked 221 positions higher in the list of search results, on average. As for FileZilla, we computed the two-sided, paired Wilcoxon signed rank test between the series of target class ranks before and after the lexicon bad smell refactoring was performed, in order to see if the difference between the two data series was statistically significant. The p-value obtained was 0.0004, indicating that the observed positive effect on the result due to refactoring was statistically significant in OpenOffice, when using Lucene as the IR technique.

When using LSI on OpenOffice, there were 18 classes for which refactoring brought an improvement in their rank and 15 for which the results were worse after the refactoring. The absolute delta was in this case 8,315, with an average of difference in ranks of 252 per target class. Thus, even when using LSI, refactoring the lexicon bad smells significantly improves the rank of the target classes in the list of search results, with a maximum improvement of 5,171 positions, for the second target class of bug 17620

(see Table 4.15). In this case, the p-value obtained for the two-sided, paired Wilcoxon signed rank test between the target class ranks before and after the refactoring was 0.0879. This is not statistically significant according to the 5% rule; however, positive average of the delta ranks (252) indicates that there was a positive effect on the ranks of the target classes when refactoring the lexicon bad smells in OpenOffice and using LSI for concept location.

One example of a significant improvement in the ranking of the target classes after the refactoring of lexicon bad smells is in the case of the first bug for OpenOffice, *i.e.*, bug 4378. The only target class for this bug, *i.e.*, *ExcXf8*, was initially located on position 1,174 in the list of results obtained when searching the source code of the system using Lucene and on position 691 when using LSI. The class originally contained 10 bad smelling identifiers, having a total of 22 lexicon bad smells. These were spread among three categories: extreme contraction (13 bad smells), misspelled terms (5 bad smells), and odd grammatical structure (4 bad smells). After refactoring, the bad smells were removed, which resulted in a significant improvement in the rank of the target class, *i.e.*, position 29 with Lucene (improvement of 1,145 positions) and 453 with LSI (improvement of 242 positions). The improvement in rank can be attributed to the meaningful terms introduced in the identifiers and thus in the corpus after expanding the abbreviations and acronyms (*e.g.*, *ExcXf8* was expanded to *ExcelFile8*, *nTrot* expanded to *nTextRotation*). This increased the number of common terms between the bug description and corpus (*e.g.*, the term *excel* appeared in the corpus only after refactoring) and also increased the frequency of other common terms (*e.g.*, the frequency of the term *rotation* has changed from 1 in the original corpus to 6 in the refactored corpus). Table 4.17 shows the description of Bug 4378, the original identifiers with lexicon bad smells, the same identifiers after refactoring, and the terms

contained only in the original and then refactored class.

Table 4.17: Example of refactoring that led to a significant improvement in rank for the target class.

Bug: 4378	**Bug description**		orientation of cell content gets lost if exporting as excel 97 or html. in my spreadsheet I rotated the writing in one row for 90 degrees to the left. If I export the sheet as excel 97 or html the writing is not rotated anymore. Exporting as excel 95 works fine
	Original	**Identifiers with LBS**	bFMergeCell, bFShrinkToFit, nCIndent, nDgDiag, nGrbitDiag, nIReadingOrder, nIcvDiagSer, nTrot, ExcXf8, GetLen, GetNum
		Terms only in original corpus	xf8, excxf8, trot, ntrot, ncindent, nireadingorder, diag, ngrbitdiag, nicvdiagser, ndgdiag, bfshrinktofit, bfmergecell, num, getnum, len, getlen
	Refactored	**Refactored identifiers**	bFormatMergeCell, bFormatShrinkToFit, nCharacterIndent, nDiagonalBorderStyle, nGrbitDiagonalBorder, nIndexReadingOrder, IndexColorValueDiagonalBorderSerial, nTextRotation, ExcelFile8, GetLength, GetNumber
		Terms only in refactored corpus	excel, file8, excelfile8, ntextrotation, character, ncharacterindent, nindexreadingorder, diagonal, border Ngrbitdiagonalborder, serial, nindexcolorvaluediagonalborderserial, ndiagonalborderstyle, format, bformatshrinktofit, bformatmergecell, number, getnumber, length, getlength

The difference in results between the two systems can be explained by the fact that OpenOffice had a significantly worse lexicon than FileZilla, containing many more lexicon bad smells, in spite of the lower number of target classes (see Table 4.12). In particular, OpenOffice contained many unusual abbreviations and acronyms that were expanded during refactoring, resulting in new, more expressive terms. This probably contributed in making the source code come closer to the language used in the bug descriptions and, thus, in the queries, making it easier to locate the target classes.

Another factor that could have contributed to the difference in results between the two systems is the fact that some of the types of bad smells, which are dominant in FileZilla, might have a low impact on IR-based concept location. These bad smells, *i.e.*, odd grammatical structure and inconsistent identifier use, take into consideration the grammar and lexical form of the words found in identifiers. IR techniques, on the other hand, disregard such aspects of the identifiers, as they are purely statistical approaches. Thus, IR could be marginally impacted by the refactoring of such bad smells, which often involves changing the order of the terms in an identifier, transforming a noun in its corresponding verb, etc. While this might not have a big influence on automated tools like IR, we argue that these types of bad smells can have a significant impact on comprehension when developers are involved.

In OpenOffice, on the other hand, the most common types of bad smells were the misspellings and extreme contractions. The performance of IR can be significantly affected by these bad smells, as they can lead to the appearance of new, statistically significant terms in the corpus of classes. Thus, the good results obtained for OpenOffice after refactoring could be explained partially by the removal of these two types of bad smells.

Threats to validity

Like any case study, this study presents some threats to its validity, which we discuss in this section. First of all, generalization of the results has to be done with care. We analyzed the impact of lexicon bad smells on IR-based concept location in only two software systems, both written in C++. Having more systems, written in other programming languages, might have led to different results.

The names proposed for the refactoring of the identifiers might have also been different if other developers would have chosen them. However, we tried to minimize this variation by having two researchers suggest the names individually.

Last, we only identified the lexicon bad smells found in target classes and did not consider the bad smells in the rest of the source code.

4.3.3 Effect of lexicon bad smells on class bug proneness

The cost of identifying and fixing faults in a system already in production may be extremely high. To avoid such costs, developers spend a large portion of the system development time on testing, to identify faulty classes prior to release. To assist developers in this respect, various studies have been conducted in the research community measuring the quality of the source code using structural metrics [116, 117, 87], process metrics [90, 56] or previous faults [66, 112]. Structural metrics are a lightweight alternative and they have been shown to have good performance for fault prediction [37].

Besides the structural metrics, several factors contribute to the faultiness of a class. One factor which we believe contributes to the faultiness of a class is LBS. LBS address the quality of the source code from the lexicon point of view. Hence, we conjecture that adding such information

to the structural metrics used in fault proneness prediction will improve the prediction. This subsection presents the investigation we conducted in Abebe *et al.* [7] to assert if this conjecture holds or not.

To prove this conjecture, we have formulated the following three research questions:

RQ1. Additional information: *Do LBS bring new information with respect to structural metrics?*

RQ2. Prediction improvement: *Do LBS improve fault prediction?*

RQ3. LBS contribution: *Which LBS help more to explain faults?*

In the first research question, RQ1, we investigate if LBS measure the same aspects of the code as structural metrics or not. To carry out this investigation, following Marcus *et al.* [81], we have used principal component analysis (PCA). PCA is a technique that uses solutions from linear algebra to project a set of possibly correlated variables into a space of orthogonal principal components (PC), or eigen vectors, where each PC is a linear combination of the original variables which in our case are the metrics. PCA is used to reveal hidden patterns that cannot be seen in the original space and to reduce the number of dimensions. We use the information captured in the PCs to analyze and answer RQ1.

For each principal component, PCA reports the coefficients of the attributes on the corresponding eigen vector. Those coefficients are interpreted as the importance of the attribute on the PC. When using PCA it is a common practice to select a subset of the principal components and discard those that explain only a small percentage of the variance. Like in Marcus *et al.* [81] we have used a threshold of 95% to select a subset of the PC. That is, we retained the components that explain up to 95% of the variance. For each principal component, we apply a 10% relative threshold

to decide which attributes contribute to the component and we rank the attributes of each PC based of their importance (weight). If LBS bring new information with respect to structural metrics then LBS will be kept in the retained principal components and will give major contributions to them. To answer RQ1 we analyze two aspects: i) the number of times an LBS contributes to at least one retained PC, and ii) the number of times an LBS is the major contributor of at least one retained PC.

In RQ2 we, then, investigate if our conjecture holds by assessing LBS' contribution, in addition to the structural metrics, in improving the capability of a prediction model. The prediction models used in this study are *logistic regression*, *random forest*, and *support vector machine*. To assess LBS' contribution, we have carried out predictions using as independent variables, on the one hand, only structural metrics, and on the other hand, structural metrics plus LBS. The capability of prediction is then evaluated using the evaluation metrics described later in this section. We then compare the results using the achieved net improvements and the average delta percentage. Prior to the comparison of the two sets of independent variables, we compare and select the best model in predicting fault prone classes using only the structural metrics.

The last research question, RQ3, is focused on identifying those LBS that contribute the most to the prediction of fault prone classes. To answer this research question, we rank each LBS based on the their importance in the best model selected in RQ2. We then calculate the median rank across the versions of the system and select the top three LBS separately for each subject system.

Variables

For building the prediction models we considered the following dependent and independent variables:

Dependent variable: As dependent variable we use a dichotomous variable, *has bug*, indicating whether a class is faulty or not. The associated experimental data have been previously published by Khomh *et al.* [65].

Independent variables: The overall set of independent variables consists of the structural metrics considered by Kpodjedo *et al.* [69] (see Table 4.18), and nine LBS defined in Section 4.1. The structural metrics list consists of the set of well-known CK metrics [33], two metrics measuring the lack of cohesion in methods (LCOM2 and LCOM5) defined by Briand *et al.* [24], and two metrics counting the number of declared attributes and methods [79]. Here after we collectively call the structural metrics used in this study as CK metrics. The LBS used in the study are *extreme contraction, inconsistent identifier, identifier construction rules, meaningless terms, misspellings, odd grammatical structure, overloaded identifiers, synonym and similar terms,* and *useless types.*

Table 4.18: List of considered structural metrics.

Acronym	**Description**
CBO [33]	Coupling between objects
DIT [33]	Depth of Inheritance Tree
LCOM1 [33]	Lack of COhesion in Methods 1
LCOM2 [24]	Lack of COhesion in Methods 2
LCOM5 [24]	Lack of COhesion in Methods 5
LOC [33]	Line Of Code
NAD [79]	Number of Attributes Declared
NMD [79]	Number of Methods Declared
NOC [33]	Number Of Children
RFC [33]	Response For a Class
WMC [33]	Weighted Methods per Class

The set of CK metrics has been calculated using the POM framework [52]. To identify LBS, we have used *LBSDetectors* presented in Section 4.2. The detectors implement general heuristics that can be configured to accommodate some variability. Hence, for each system used in our

study, we have manually explored their documentations, when available, and configured the detectors accordingly.

Evaluation metrics

In the literature, various evaluation metrics are used to evaluate the prediction capability of independent variables and to compare prediction models [116, 117, 87, 56, 112]. We have categorized these metrics into three groups: *rank*, *classification*, and *error* metrics. Below we present the details of each category.

Rank

Rank metrics sort the classes based on the value of the dependent variable assigned to each class. Then a cumulative measure is computed using the actual values of the dependent variable over the ranked classes to assess the model and/or the independent variables. In our study, we have considered two types of rank metrics: P_{opt} and FPA (Fault Percentile Average).

P_{opt}: is an extension of the Cost Effective (CE) measure defined in [10]. P_{opt} takes into account the costs associated with testing or reviewing a module and the actual distribution of faults, by benchmarking against a theoretically possible optimal model [87]. It is calculated as $1 - \Delta_{opt}$, where Δ_{opt} is the area between the optimal and the predicted cumulative lift charts. The cumulative lift chart of the *optimal* curve is built using the actual defect density of classes sorted in decreasing order of the defect density (and increasing lines of code, in case of ties). The cumulative lift chart of the *predicted* curve is built like the optimal curve, but with classes sorted in decreasing order of fault prediction score.

FPA: is obtained from the percentage of faults contained in the top $m\%$ of classes predicted to be faulty. It is defined as the average, over all values of m, of such percentage [112, 14]. On classes listed in increasing order of

predicted numbers of faults, FPA is computed as:

$$\frac{1}{NK}\sum_{k=1}^{K}(k * n_k)$$

where N is total number of actual faults in a system containing K classes, n_k is the actual number of faults in the class ranked k [112].

In our study, however, we predict the probability of fault proneness of a class instead of the number of faults. Hence, we have adapted the metrics by using the predicted probability of fault proneness to sort the classes, and 0 and 1 are used as a replacement of the number of defects. 1 is used when a class is actually faulty; 0 otherwise.

Classification

Predicting fault proneness of a class is a classification problem. Hence, in various studies the confusion matrix (shown in Table 4.19) is used to evaluate models and analyze the prediction capability of the independent variables. From the confusion matrix the following measures are computed to conduct the evaluation.

Accuracy (A): measures how accurately both the actual faulty and non-faulty classes are classified as faulty and non-faulty by the predictor. It is computed as the ratio of the number of classes that are correctly predicted as faulty and non-faulty to the total number of classes $A = (TP + TN)/(TP + TN + FP + FN)$. A score of 1 indicates that the model used for the prediction has classified all classes as faulty and non-faulty correctly.

Correctness (P): is the precision of a predictor in identifying the faulty classes as faulty. It is computed as the ratio of classes which are correctly predicted as faulty to the total number of classes which are predicted to be faulty $P = TP/(TP + FP)$. A prediction model is considered very precise if all the classes predicted as faulty are actually faulty, i.e. if $P = 1$.

Completeness (R): is the recall of a predictor. It tells how many of the actually faulty classes are predicted as faulty. Completeness is computed as the ratio of the number of classes which are correctly predicted as faulty to the total number of classes which are actually faulty in the system $R = TP/(TP + FN)$.

F-measure (F): is a measure used to combine the above two inversely related classification metrics, correctness, and completeness. F-measure is computed as the harmonic mean of correctness and completeness ($F = (2 * P * R)/(P + R)$).

Matthew's Correlation Coefficient (MCC): is a measure commonly used in the bioinformatics community to evaluate the quality of a classifier [86]. It is a measure which is quite robust in the presence of unbalanced data. MCC is computed as:

$$\frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

The value of MCC ranges from -1 to 1. -1 indicates a complete disagreement while 1 indicates the opposite.

Table 4.19: Prediction confusion matrix (TP=True positive, TN=True negative, FP=False positive, FN=False negative)

		Actual	
		Faulty	**Not faulty**
Predicted	**Faulty**	TP	FP
	Not faulty	FN	TN

Error

In the last category of the evaluation metric types, we have *absolute error (E)*. Absolute error is a measure based on the number of faults incorrectly

predicted or missed:

$$E = \sum_{k=1}^{K} |\hat{y_k} - y_k|^2$$

where $\hat{y_k}$ is the predicted number of faults in class k and y_k the actual number of faults [56]. As we are interested in the fault proneness of a class and not in the number of faults it contains, we use 0 and 1, as a replacement of the number of faults. 1 is used when a class is actually faulty/predicted to be faulty and 0 otherwise. Unlike the other evaluation metrics, for absolute error a value closer to 0 indicates better prediction capability.

Prediction models settings

Here we describe the particular settings of each prediction model. All computations are performed using R[20].

Logistic Regression Model: We used the *Generalized Linear Model* (package *stats*) *glm (family=binomial("logit"))*. We perform backward variable elimination and predict using the retained variables.

Random Forest: We use the function *randomForest* (package *randomForest*) with the number of trees being 500 as did Weyuker *et al.* [112].

Support Vector Machine: We used the Support Vector Machine model (package *e1071*) *svm (kernel="radial")*. Elish *et al.* [42] used the same kernel, which showed good performance.

Common settings: The following settings are common for all models: As Gyimóthy *et al.* [53] we standardize all metrics before performing the calculations (*i.e.*, zero mean and unit variance). Like in Kamei *et al.* [64], for each type of model, we predict faulty classes in two configurations: within the same version and for the next version. Prediction within the same version represents scenarios in which there is no prior record of buggy

[20]http://www.r-project.org/

classes and new systems while the latter represents scenarios in which a system's evolution is available and documented. When predicting within the same version, we use 10-fold cross validation. For each configuration we build two models: one where the independent variables are the CK metrics alone and the second where the independent variables are CK and LBS.

Subjects

For our case study, we have considered three open source systems written in Java, ArgoUML[21], Eclipse[22], and Rhino[23]. ArgoUML is a UML modeling tool which includes support for all standard UML 1.4 diagrams while Eclipse is an IDE which supports different languages. In this study we have used the IDE for Java. Rhino is a Java implementation of JavaScript. The summary of the versions of the systems used in our study are shown in Table 4.20.

Results and discussions

RQ1: Additional information

Table 4.21 shows the percentage of the analyzed versions that retained the specific LBS in at least one PC. In Table 4.22 we show the percentage of the analyzed versions where each LBS was ranked first. Table 4.23 shows the weight and ranking (in parentheses) of the attributes for ArgoUML v0.16 after the relative threshold is applied.

ArgoUML: For all versions of ArgoUML we retained between 11 and 13 principal components that explain at least 95% of the variance. Two LBS attributes were kept in at least one PC in all versions and those are:

[21]http://argouml.tigris.org/
[22]http://www.eclipse.org/
[23]http://www.mozilla.org/rhino/

Table 4.20: Summary of the systems.

System	Version	LOC	Classes	
			Total	Defective
ArgoUML	0.10.1	154442	863	49
	0.12	171746	946	47
	0.14	182627	1227	93
	0.16	185335	1185	152
	0.18.1	196505	1249	52
	0.20	186055	1333	127
Eclipse	1.0	1049434	4596	96
	2.0	1471858	5985	163
	2.1.1	1735010	6748	98
	2.1.2	1737345	6750	78
	2.1.3	1740487	6754	149
Rhino	1.4R3	43791	94	66
	1.5R1	68086	124	22
	1.5R3	86937	166	98
	1.5R4	92398	180	35
	1.5R5	92687	181	39
	1.6R1	102511	178	37
	1.6R4	102974	180	138
	1.6R5	79144	124	37

inconsistent identifier and *useless types*. Between them, *useless types* was the major contributor of at least one PC in all versions.

Rhino: The number of components that explain at least 95% of the variance for Rhino is the same as for ArgoUML. Five LBS attributes were kept in at least one PC in all versions and those are: *inconsistent identifier*, *synonym and similar terms*, *odd grammatical structure*, *overloaded identifiers*, and *meaningless terms*. As in ArgoUML, one LBS attribute was present as a major contributor in all versions and this is *overloaded identifiers*.

Eclipse: The number of retained PC is between 13 and 14. The six LBS that are present in all versions are: *inconsistent identifier*, *odd grammatical structure*, *extreme contraction*, *overloaded identifiers*, *useless types*, and *meaningless terms*. The majority of them (four) are ranked first: *inconsistent identifier*, *extreme contraction*, *overloaded identifiers*, and *meaningless terms*.

Overall: All LBS were present in more than 50% of the analyzed systems. *inconsistent identifier* was present in at least one dimension in all analyzed versions meaning that it is the major LBS attribute that helps to explain a new variability dimension. Another different variability dimension in most cases seems to be captured by *overloaded identifiers* and *useless types*.

The results show that the majority of LBS (all considered in the study but three) are major contributors in at least one dimension for more than 50% of the analyzed versions. The strongest percentages are obtained by *inconsistent identifier*, *overloaded identifiers*, and *useless types*. The weakest percentages across versions appear to be *odd grammatical structure*, *misspelling*, and *synonym and similar terms*.

RQ2: Prediction improvement

For each evaluation metric, Table 4.24 shows the average values scored by the corresponding model on all types of prediction (same and next version).

Table 4.21: LBS retained in the principal components (MS=Misspelling, II=Inconsistent identifier, SST=Synonym and similar terms, OGS=Odd grammatical structure, EC=Extreme contraction, OI=Overloaded identifiers, IC=Identifier construction, UT=Useless types, MT= Meaningless terms).

System	MS	II	SST	OGS	EC	OI	IC	UT	MT
Eclipse	0.0%	100.0%	40.0%	100.0%	100.0%	100.0%	80.0%	100.0%	100.0%
ArgoUML	66.7%	100.0%	50.0%	66.7%	66.7%	83.3%	83.3%	100.0%	16.7%
Rhino	87.5%	100.0%	100.0%	100.0%	75.0%	100.0%	62.5%	87.5%	100.0%
All	57.9%	100.0%	68.4%	89.5%	78.9%	94.7%	73.7%	94.7%	73.7%

Table 4.22: LBS ranked first in the retained principal components (MS=Misspelling, II=Inconsistent identifier, SST=Synonym and similar terms, OGS=Odd grammatical structure, EC=Extreme contraction, OI=Overloaded identifiers, IC=Identifier construction, UT=Useless types, MT= Meaningless terms).

System	MS	II	SST	OGS	EC	OI	IC	UT	MT
Eclipse	0.0%	100.0%	20.0%	20.0%	100.0%	100.0%	80.0%	80.0%	100.0%
ArgoUML	50.0%	83.3%	0.0%	16.7%	33.3%	83.3%	66.7%	100.0%	16.7%
Rhino	0.0%	87.5%	50.0%	0.0%	50.0%	100.0%	62.5%	62.5%	87.5%
Overall	15.8%	89.5%	26.3%	10.5%	57.9%	94.7%	68.4%	78.9%	68.4%

Table 4.23: Detailed results of PCA for ArgoUML v0.16.

PC	**PC1**	**PC2**	**PC3**	**PC4**	**PC5**	**PC6**	**PC7**	**PC8**	**PC9**	**PC10**	**PC11**
Cumulative proportion	40.9%	51.8%	59.88%	65.54%	71.06%	76.2%	81.02%	85.29%	89.33%	92.91%	95.53%
CBO	**0.275(9)**	0.203	0.35	0.0741	0.0176	0.0994	0.0954	0.11	0.0853	0.0607	0.0953
DIT	0.0311	0.0551	0.123	0.13	**0.772(1)**	0.46	0.0998	0.338	0.0457	0.0665	0.107
LCOM1	**0.281(7)**	0.36	0.0835	0.0277	0.00268	0.0387	0.0503	0.0812	0.184	0.28	0.0641
LCOM2	**0.278(8)**	0.366	0.0879	0.0272	0.00323	0.0382	0.0507	0.0807	0.188	0.282	0.0685
LCOM5	0.111	0.307	0.00385	0.0976	0.206	0.269	0.0478	**0.753(1)**	0.35	0.16	0.0615
LOC	**0.29(5)**	0.15	0.367	0.0276	0.0123	0.0217	0.0729	0.165	0.0668	0.0341	0.134
NAD	0.21	0.101	**0.442(1)**	0.0404	0.0119	0.0138	0.0763	0.0568	0.368	0.0434	**0.604(1)**
NMD	**0.338(1)**	0.0988	0.0846	0.0403	0.0618	0.00984	0.0373	0.0147	0.0764	0.0998	0.108
NOC	0.0205	0.107	0.0854	0.386	0.428	**0.774(1)**	0.00912	0.113	0.132	0.0118	0.0585
RFC	**0.296(4)**	0.176	0.274	0.0458	0.0453	0.00323	0.0397	0.0971	0.0342	0.0189	0.0197
WMC	**0.318(2)**	0.12	0.286	0.0338	0.0116	0.0181	0.0958	0.0996	0.0384	0.0614	0.131
misspelling	0.24	0.201	0.187	0.207	0.255	0.0979	0.0973	0.0529	0.0174	0.211	**0.571(2)**
inconsistent identifier	0.205	0.246	0.147	0.0116	0.0484	0.0543	0.383	0.29	0.189	**0.6(1)**	0.178
synonym and similar terms	**0.288(6)**	0.314	0.102	0.00884	0.00461	0.000561	0.155	0.0958	0.0317	0.154	0.241
odd grammatical structure	**0.305(3)**	0.0148	0.28	0.013	0.048	0.0274	0.173	0.0301	0.0203	0.0892	0.152
extreme contraction	0.0772	0.253	0.266	**0.592(1)**	0.276	0.212	0.0624	0.166	0.336	0.288	0.15
overloaded identifiers	0.144	0.0224	0.00102	0.236	0.161	0.00659	**0.802(1)**	0.0575	0.00447	0.467	0.0241
identifier construction	0.14	**0.416(1)**	0.271	0.266	0.0227	0.143	0.0104	0.0399	0.41	0.139	0.299
useless types	0.0413	0.248	0.236	**0.539(2)**	0.0042	0.153	0.304	0.318	**0.561(1)**	0.186	0.00662

CK metrics are used to build the prediction models. The values in bold are the best values of the three models considered for the given metrics. For all the systems, SVM scores first for the majority of the evaluation metrics. Hence, we have based our investigation of LBS' contribution to the improvement of fault prediction on SVM.

Table 4.25 shows the number of versions in which CK plus LBS metrics improve, decrease or keep the prediction unchanged, when compared to CK metrics alone. The last two columns show the net improvement within/across versions and the average delta percentage of LBS plus CK metrics over CK alone for the various evaluation metrics. Positive values of net improvements, for all types of evaluation metrics, indicate that in the majority of the versions CK plus LBS are better predictors than CK alone, while negative values indicate the opposite. A zero net improvement means that both sets of independent variables were found better than the other in an equal number of versions or that they are equal in all versions. For all evaluation metrics except *absolute error*, the same is true for the average delta percentage, which is computed on the average values over all versions of the corresponding system. For *absolute error*, a negative value means that there is a reduction in the amount of error and hence indicates an improvement while the opposite holds for positive values of *absolute error*.

The predictions using CK plus LBS metrics have outperformed those of CK alone in most of the versions of the three systems, when considering both within and across version prediction. For ArgoUML, the prediction on the same versions using CK and LBS together has improved in at least 4 of the 6 versions considered, according to the different evaluation metrics. For Eclipse the improvement observed in all versions is consistently reported by all evaluation metrics. Figure 4.14 shows the average values of all versions of Eclipse for the evaluation metrics. We observe an

important improvement for all metrics except for *accuracy* where the improvement is minor. The evaluation metrics result for Rhino shows that there is improvement in at least half of the versions considered (4 out of 8). The distributions of the evaluation metrics for all systems are shown in Figure 4.16.

When predicting on the next version, CK plus LBS have been found to be good predictors in the majority of Eclipse's and Rhino's versions by some evaluation metrics; according to other evaluation metrics they are the same as CK alone. Figure 4.15 contrasts the predictions of the two models for Eclipse. For ArgoUML, negative net improvement values are observed in three of the evaluation metrics while the other three show that there is a net improvement in at least 3 out of the 5 versions predicted. Overall, in both types of predictions, within and across versions, CK plus LBS are better than CK alone in the majority of the versions. This result is confirmed by almost all average delta percentage values shown next to each net improvement. The average delta percentage decreased only in 7 out of the 36 metrics computed for the three systems. Hence, we can answer RQ2 affirmatively.

Of the two types of predictions, the predictions conducted on the same versions using LBS plus CK metrics have shown improvement in more versions than observed in predictions on the next version. For example, in Eclipse LBS plus CK metrics improved the prediction in all versions (5 of 5), while across versions the improvement is observed in at most half of the versions (2 of 4). The difference can be observed by comparing Figures 4.14 and 4.15.

RQ3: LBS contribution

Table 4.26 shows the ranked LBS according to their contribution to SVM. The median rank across versions is indicated within brackets.

Table 4.24: Average values of each model while using the CK metrics as independent variable (LRM=Logistic regression model, RF=Random forest, SVM=Support vector machine).

System	Category	Metric	LRM	RF	SVM
ArgoUML	**Rank**	P_{opt}	0.468	0.505	**0.603**
		FPA	38.9	4.91	**45.8**
	Error	E	91.6	88.7	**86.8**
	Classifi.	A	0.922	0.925	**0.927**
		F	0.0797	**0.199**	0.0812
		MCC	0.0991	**0.199**	0.12
Eclipse	**Rank**	P_{opt}	0.458	0.521	**0.637**
		FPA	**67**	0.444	60.8
	Error	E	124	127	**118**
	Classifi.	A	0.98	0.98	**0.981**
		F	0.0101	**0.0985**	0.0439
		MCC	0.0167	**0.139**	0.104
Rhino	**Rank**	P_{opt}	0.528	0.535	**0.568**
		FPA	21.3	16.3	**21.4**
	Error	E	42.8	42.8	**41.2**
	Classifi.	A	0.71	0.71	**0.717**
		F	0.552	0.538	**0.579**
		MCC	0.346	0.336	**0.375**

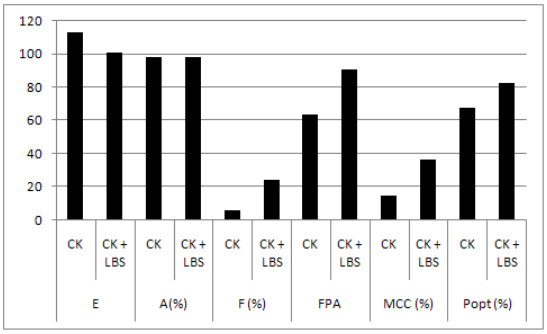

Figure 4.14: Eclipse: Average of the evaluation metrics for same version prediction.

Table 4.25: CK and CK + LBS prediction capability comparison using SVM.

Systems	Predi. version	Category	Metric	Imp.	Dec.	Equal	Net imp.	Avg. delta %
ArgoUML	**Same**	**Error**	E	5	0	1	5	-8.94
		Rank	P_{opt}	5	1	0	4	-5.878
			FPA	5	1	0	4	1.917
		Classifi.	A	5	0	1	5	0.6738
			F	5	0	1	5	81.51
			MCC	5	0	1	5	56.72
	Next	**Error**	E	1	3	1	-2	3.165
		Rank	P_{opt}	2	3	0	-1	4.405
			FPA	4	1	0	3	9.948
		Classifi.	A	1	3	1	-2	-0.2805
			F	4	0	1	4	100
			MCC	4	0	1	4	-1400
Eclipse	**Same**	**Error**	E	5	0	0	5	-11.11
		Rank	P_{opt}	5	0	0	5	22.91
			FPA	5	0	0	5	43.53
		Classifi.	A	5	0	0	5	0.2176
			F	5	0	0	5	314.6
			MCC	5	0	0	5	140.8
	Next	**Error**	E	2	2	0	0	1.212
		Rank	P_{opt}	2	2	0	0	-3.067
			FPA	2	1	1	1	0.8696
		Classifi.	A	2	2	0	0	-0.02364
			F	3	1	0	2	234.3
			MCC	3	1	0	2	200
Rhino	**Same**	**Error**	E	6	1	1	5	-11.27
		Rank	P_{opt}	6	2	0	4	3.233
			FPA	6	0	2	6	3.518
		Classifi.	A	6	1	1	5	2.343
			F	7	0	1	7	8.521
			MCC	6	1	1	5	15.24
	Next	**Error**	E	2	1	2	1	-0.6042
		Rank	P_{opt}	3	2	0	1	3.085
			FPA	4	0	1	4	8.861
		Classifi.	A	2	1	2	1	0.4126
			F	3	0	2	3	3.925
			MCC	3	0	2	3	10.53

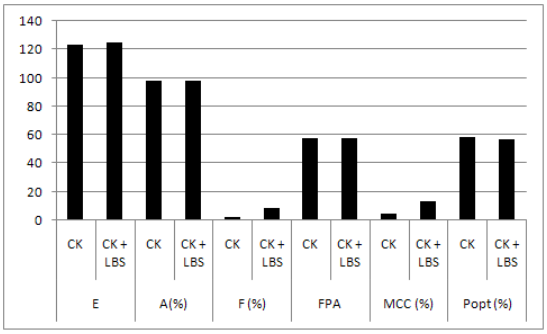

Figure 4.15: Eclipse: Average of the evaluation metrics for next version prediction.

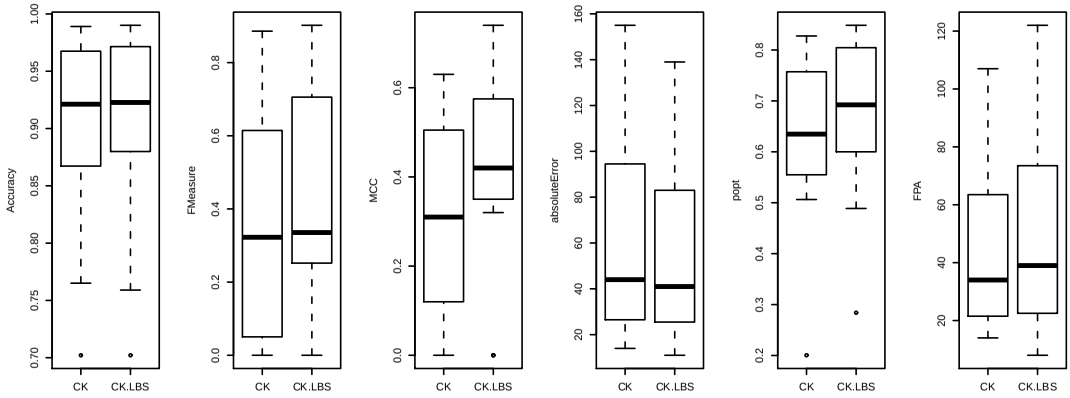

Figure 4.16: All systems: Evaluation metrics for same version prediction.

Table 4.26: Ranked LBS according to SVM.

ArgoUML	**Rhino**	**Eclipse**
Synonym and similar term (4)	Odd grammatical structure (6.5)	Extreme contraction (3)
Inconsistent identifier (6.5)	Misspelling (7.5)	Overloaded identifiers (4)
Overloaded identifiers (8.5)	Inconsistent identifier (10)	Identifier construction (4)
Identifier construction (9.5)	Synonym and similar term (11)	Useless types (7)
Odd grammatical Structure (10)	Meaningless terms (12)	Synonym and similar term (8)
Misspelling (10.5)	Identifier construction (12.5)	Odd grammatical structure (8)
Useless types (13)	Extreme contraction (13)	Meaningless terms (10)
Extreme contraction (15.5)	Overloaded identifiers (14)	Inconsistent identifier (11)
Meaningless terms (20)	Useless types (17.5)	Misspelling (14)
Whole-part (20)	Whole-part (20)	Whole-part (20)

The following observations can be made across the different systems: *synonym and similar terms* is in the top five most important LBS for all systems. *Inconsistent identifier* and *overloaded identifiers* are in the top three for two of the systems. *Inconsistent identifier* and *synonym similar* have a median rank at most 11. Finally, *whole-part* does not seem to be important for fault prediction.

Our findings are consistent with previous research on program identifiers which suggest that identifiers using synonyms lack conciseness and consistency [74].

We also observe that some LBS tend to have a specific contribution for particular systems. For instance, *extreme contraction* is ranked first among all LBS for Eclipse, while *misspelling* is ranked second for Rhino.

Threats to validity

Our study uses the structural metrics considered by Kpodjedo *et al.* [69] as a baseline to investigate the contribution of LBS in predicting fault proneness of a class. In the literature, however, there are other metrics which are proposed to achieve the same goal. In our future work, we plan to investigate if LBS are complementary also to these metrics.

Different evaluation metrics assess different aspects of prediction models and hence might give different results. To see if our results are consistent across different evaluation metrics, we based our evaluation on selected evaluation metrics which assess different aspects and have been commonly used in recent studies.

The prediction results depend on the used models and their configurations. We used default configurations or configurations used in other studies. Further tuning of the parameters however could change the rankings of the models. The best model from RQ2, SVM, was used with default parameters. Di Martino *et al.* [40] suggest the use of genetic algorithms to select the parameters for further improvement of the results.

In this study, we have considered only three Java systems which limits its generalizability. However, these systems have been selected from different domains and with different size to limit this threat. Besides, they are real world open source programs which are actively evolving.

4.4 Conclusion

In this chapter, we have introduced the notion of lexicon bad smells and defined a catalog, which lists and describes a set of twelve such smells. We have developed a suite of detectors based on the heuristics described for each smell in the catalog. The accuracy of ten detectors in identifying LBS has been assessed on four real world open source systems, and we have

discussed the limitations of the detectors.

We have also assessed the impact of LBS in concept location which is one of the program understanding tasks, and the contribution LBS make to the fault prediction approaches that use source code structural metrics. The impact of LBS in concept location was studied using IR-based concept location techniques on two real world open source systems. The results indicate that lexicon bad smells can be an important factor to consider when performing IR-based concept location and that refactoring these smells can have a significant positive impact on the task. In particular, if the lexicon of the system being maintained is known to be of relatively low quality (as was the case in one of the two systems analyzed), the benefits of lexicon smell removal are expected to be quite significant.

To assess the contribution of LBS to the structural metrics based fault prediction approaches, we have conducted a study using three real world open source systems. The results show that in the majority of the cases using LBS with the structural metrics improves fault prediction. To assess the improvement, we have used different evaluation metrics that address different aspects of the prediction. The assessment shows that the improvement is consistent in almost all types of evaluation metrics.

In the future, we plan to address the limitations of LBS detectors, and extend the existing list of bad smells with the collaboration and feedback of the research community. In addition, we plan to further analyze the impact of each LBS in concept location, investigate whether LBS provide additional contributions to other types of metrics that are used to predict the fault proneness of a class, and validate our results on more systems, possibly written in different programming languages.

Chapter 5

Automated identifier completion and replacement

One of the identifier quality attributes commonly agreed by many authors is conciseness and consistency [8, 39, 74, 101, 2]. In Section 4.1, we have presented inconsistent identifiers as one type of lexicon bad smell. This smell is usually caused by lack of knowledge about how concepts are named in the source code. Acquiring and maintaining such knowledge as the software evolves, on the other hand, is not an easy task, especially if programmers are working on a large software system, or if they are new to the software system. Learning the knowledge from other programmers is not always possible, as programmers might be located in other parts of the world, or may not work on the software any longer.

To address this problem, some works have proposed approaches to help programmers in detecting and avoiding such "violations" [39, 74, 101, 2, 60]. Deissenboeck and Pizka [39] have formally defined conciseness and consistency of identifiers. Their definition involves a bijective mapping between concepts and names. Such mappings are stored in a manually maintained identifier dictionary, used during identifier naming. Lawrie *et al.* [74] have proposed a syntactic approach which addresses the cost that might be incurred due to the construction and maintenance of the identi-

fier dictionary in Deissenboeck and Pizka's approach [39]. Lawrie *et al.*'s approach [74] defines syntactic rules based on containment of identifiers to identify violations of conciseness and consistency.

In this chapter, we present an automated approach, based on the ontological concepts and relations automatically extracted from the source code (see Chapter 2), to suggest new names for identifiers [6]. The suggestions can be used to complete part of a new identifier or to replace it with a better name. During software maintenance, this approach helps programmers to identify and reuse concept names already used in the code. The ontology is extracted from the source code following the NLP-based approach described in Section 2.2, which exploits the natural language information captured in the existing identifiers.

Completion and replacement suggestions are obtained from the names of the concepts and relations in the ontology. We take into consideration the context in which the identifier is being defined to rank the suggestions. The details of the approach are discussed in Sections 5.1 and 5.2. To demonstrate our approach we use the example ontology shown in Figure 5.1. Our proposal is in-line with Deissenboeck and Pizka [39] and Lawrie *et al.* [74], but it differs substantially in approach (ours is based on an automatically extracted ontology) and application scenario (identifier completion and replacement).

We have evaluated the approach by simulating the programmers' activity during identifier naming [6]. The approach followed in the evaluation and the obtained results are discussed in Section 5.3.

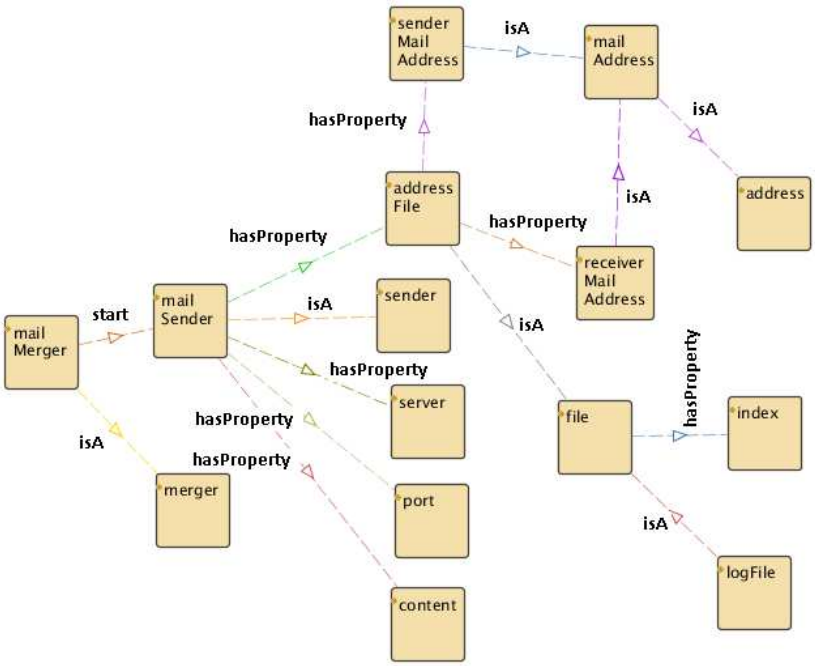

Figure 5.1: An example ontology

5.1 Identifying candidate concepts

Our proposed approach identifies candidate concepts and relations which can be used as suggestions for a partially written new identifier. We distinguish three types of program entities: *class*, *attribute*, and *method*. The suggestions can be used to complete or replace part or all of the identifier terms. To identify the candidate suggestions, we use the following methods.

Term prefixes: Identifiers are composed of one or more terms. In this method we consider the initial letters of the term being currently written as a prefix to be matched. The prefix is used to search and identify candidate concepts in the ontology which start with the given prefix. The search is conducted as the developer writes the first few letters of a term in an identifier. The result is further filtered as she adds more letters to the prefix. If the developer is writing the first term of a method name, the

relations of the ontology are searched for possible candidate relations that start with the prefix of the first term. In fact, in the ontology extracted following our approach (see Section 2.2), verbs in method names are mapped to relations between a doer concept and the object concept which takes the action. The suggestions which are identified by this method are used to complete part of the identifier or to replace the whole identifier.

For example, if a developer starts to type an attribute name by writing letter "*s*", in a system from which the ontology shown in Figure 5.1 is extracted, our approach searches the ontology for concepts that start with "*s*" and lists the candidate suggestions *sender*, *senderMailAddress*, and *server*. Had the identifier been a method name, the relations in the ontology would also be searched and the relation name *start* would be added to the list of candidate suggestions. If the developer types "*er*" after "*s*", the list of candidate terms will contain only the term *server* (see Table 5.1).

Table 5.1: Suggestions using prefix information

Being typed	**Suggestions**
s...	sender senderMailAddress server
se...	sender senderMailAddress server
ser...	server

Neighboring concepts: When a fully written term in a new identifier being currently typed is matched to a concept (or relation for methods) in the ontology, all neighboring concepts are considered as candidate suggestions. The rationale behind this method is that the neighboring concepts are concepts which are closely related to the matched concept and possibly

used together with it in other parts of the system. If for example, in a system from which the ontology shown in Figure 5.1 is extracted, a developer writes *file* as part of an identifier, the neighboring concepts *addressFile*, *index*, and *logFile* will be presented as candidates (see Table 5.2).

Table 5.2: Suggestions using prefix and neighboring concepts information

Being typed	**Suggestions**
f...	file
⋮	⋮
file	addressFile index logFile

If among the suggested candidate concepts in Table 5.2, the most appropriate name for the whole identifier is the specialized concept *addressFile*, the developer may use the suggested concept to replace what she already wrote. If the intention is to name the identifier as *fileIndex*, then she may use the suggested concept *index* to complete the naming.

Synonyms: The terms used in identifiers can be abbreviations, acronyms or dictionary words. As the developer adds more letters to a prefix and this becomes a dictionary word, we use WordNet [89, 44] to identify its synonyms. Such synonyms are then matched to the concepts and relations in the ontology. If they are present in the ontology, they are added to the candidate concepts list. For example, if the developer types the term *message*, the synonyms of *message*, *content*, *subject matter*, and *substance*, are collected from WordNet and matched to the concepts and relations in the ontology shown in Figure 5.1. The only match found is *content* and, hence, only this one is presented as a candidate term. The suggestions provided in this manner are used mainly to replace the word for which the synonym is identified.

5.2 Prioritizing candidate concepts

To prioritize the candidate suggestions, we rank them based on their "relevance" to the *context* in which the identifier term is being written. The *context, X,* is defined as the set of class, attribute and method terms found in the enclosing scope (*e.g.*, the enclosing class or package), in which the identifier term of interest is found. In case of ties in the relevance score, suggestions are listed alphabetically.

Relevance of a candidate concept, c_c, to a context, X, is defined as the sum of ratios of terms in the neighboring concepts shared with the context:

$$\sum_{\forall c_i \epsilon C | c_c \Upsilon c_i} \frac{|split(c_i) \cap X|}{|split(c_i)|} \tag{5.1}$$

where C is the set of concepts in the ontology, Υ represents any ontological relation in the ontology and does not take the order of the concepts into consideration (*i.e.*, it is regarded as a undirected relation), and *split* is a function which gives the terms in the concept name.

For example, if the developer is writing the identifiers shown in Tables 5.1 and 5.2 in a context $X = \{mail, sender\}$, the suggestions will be ranked using Equation 5.1 as shown in Table 5.3 top and bottom, respectively.

Relevance of a candidate relation, Υ_c, to a context, X, is defined as the sum of ratios of terms in the two related concepts shared with the context:

$$\sum_{\forall c_i, \forall c_j \epsilon C | c_i \Upsilon_c c_j} \frac{|split(c_i) \cap X| + |split(c_j) \cap X|}{|split(c_i)| + |split(c_j)|} \tag{5.2}$$

where C is the set of concepts in the ontology and *split* is a function which gives the terms in the concept name.

For example, if the ontological relation *start* in Figure 5.1 is one of the candidate suggestions for a method name in a context $X = \{mail, sender\}$, its relevance score will be 0.75 (see Equation 5.2).

Table 5.3: Ranked suggestions

Being typed	Rank	Suggestions	Score
s...	1	sender	1
	2	server	1
	3	senderMailAddress	0.5
se...	1	sender	1
	2	server	1
	3	senderMailAddress	0.5
ser...	1	server	1
f...	1	file	0
⋮	⋮	⋮	⋮
file	1	addressFile	2
	2	index	0
	3	logFile	0

5.3 Evaluation

To evaluate our approach, we have simulated the activities of a developer and automatically collected the suggestions provided by the methods described above. In our experiments, between identifier completion and identifier replacement we have chosen the identifier completion scenario, since in this scenario it is easier to automate the simulation of the developer's activities, with no need for any external, subjective input. The automatically produced suggestions have been analyzed to answer the following research questions:

- **RQ1** *How many completion suggestions are correct?*

- **RQ2** *Are the correct completion suggestions listed on the top of the ranked suggestion list?*

In the first research question, we investigate if the suggestions provided by our approach to complete part of an identifier are *correct*. By *correct*, in this context, we mean that a suggestion from the suggestion list matches in part or entirely the identifier to be completed. A suggestion matches an identifier to be completed partially, if it can be used to complete part of the identifier. For example, a suggestion list containing *file* for the prefix character *f* is considered correct if the identifier to be completed is *fileIndex*. In fact, we are re-enacting the identifier writing process considering identifiers that already exist in the system, so that we know in advance how the "correct" completion looks like.

A suggestion list may contain more than one suggestion which are ranked according to their relevance to the context in which the identifier is being declared (see Section 5.2). When our approach gives correct suggestions, in the second research question we further investigate the ranks of the top correct suggestions in the suggestion lists. The analysis evaluates how good our approach is in ranking high the relevant suggestions.

To answer the research questions, we have defined two metrics: *success rate* and *average rank*. *Success rate* is defined as the probability of getting correct suggestions, and is used to answer RQ1. It is computed as the ratio of the number of correct suggestions to the total number of suggestions provided. A suggestion is provided for each prefix of terms, as written by the developers of the subject systems under analysis.

The second metrics *average rank* is computed only for correct suggestions (*i.e.*, suggestion lists containing at least one correct suggestion). It is computed as the average of the ranks given to the top correct suggestion in the suggestion lists.

5.3.1 Methodology

Naming suggestions can be provided to developers as they write each character of the identifier being defined. To evaluate the suggestions, we have simulated the developers' activity using a tool. The simulation is conducted by first removing the identifier from the system (*i.e.*, we pretend that this identifier has not been introduced yet) and then automatically collecting the suggestions generated for each prefix sequence of characters and terms of the identifier. The ontology used to produce the suggestion list is obtained from a version of the system in which the identifier being completed has been removed.

In practice, the correctness of a suggestion as a replacement or completion would be assessed by the developers. In our simulation, however, we do not involve developers and, hence, we rely on the actual name of the identifier that has been removed from the system and has been considered as to be completed. This experimental setting makes it difficult to evaluate the replacement scenario. In fact, we use as "correct" identifier only the original, fully completed name, while in a replacement scenario, other alternative names might be equally acceptable. However, assessing their acceptability would require subjective, user judgment. For this reason, we consider only the identifier completion scenario. Since the *synonyms* method is not useful in a completion scenario where the prefix is already correct, we did not evaluate this method in our experiments. We focused on evaluating the other two methods, which can be used to effectively provide suggestions for completion, *term prefixes* and *neighboring concepts* (see Section 5.1).

Our approach uses an ontology extracted from the system to provide suggestions. To extract the ontology, we have exploited the natural language information captured in identifiers of the source code as presented in

Section 2.2. The ontology is a reflection of the current state of the system at hand. Hence, during the simulation, we have eliminated the concepts and relations which can be extracted from the removed identifier and used the updated ontology to generate the suggestions.

To evaluate the suggestions provided using *term prefixes*, all prefixes of the terms in an identifier which are of length four or less are used to search and identify candidate suggestions. For suggestions provided using *neighboring concepts*, we have used initial sequence of four or less terms (without including the last term) in the identifier. To get the terms which constitute an identifier, the name is split using camel casing and underscore. The splitting can also be carried out using the approaches described in the related literature [72, 36].

When a concept is composed of more than one term, the terms appearing on the left are often specifiers. For example, in the identifier *userFileIndex*, *userFile* and *user* can be considered as specifiers of *index* and *fileIndex* concepts, respectively. Hence, as a variant of the *neighboring concepts* method, we have used the first four or less terms as a specifier, and we have searched and identified candidate concepts in the ontology which start with the given specifier. We refer to this variant as *concept prefixes*. The candidate suggestions provided following the above three methods are then evaluated using the metrics *success rate* and *average rank*.

Identifiers in different classes may happen to have the same name. In such cases, a suggestion for completing an identifier can easily be obtained by keeping track of already existing identifiers; with no need to resort to an ontology. In order to see how many of the correct completion suggestions can be obtained from the same identifier appearing elsewhere in the system, we have introduced the metrics *duplicate completions*. *Duplicate completions* is the ratio of the number of correct suggestions of identifiers which are duplicated in other parts of the system (*i.e.*, in other classes),

to the total number of suggestions.

In our evaluation, method identifiers which appear more than once in a class due to overloading are considered only once, and we have also excluded *main*, *constructor*, and *destructor* method identifiers.

5.3.2 Subjects

Our experiment was conducted on six open source systems: ADempiere, FileZilla client, JEdit, OpenOffice, ThunderBird, and WinMerge. A summary of the features of these systems is shown in Table 5.4.

Table 5.4: Features of the subject systems. The identifier count does not include constructor and destructor identifiers; overloaded method names are counted only once.

System	Version	Files	Classes	Lines of text	Identifiers count
ADempiere	3.1.0	1833	1917	482094	38241
FileZilla	3.0.0	264	208	89080	2663
JEdit	4.2	224	639	79198	5000
OpenOffice	1.0.0	12761	12112	4666417	182258
ThunderBird	2.0.0.0	11019	5949	3548012	72431
WinMerge	2.12.2	257	146	67643	2859

ADempiere[1] is an enterprise resource planning software, while FileZilla client[2] is a cross-platform, graphical FTP, FTPS, and SFTP client. JEdit[3] is a programmer's editor which provides syntax highlighting for over 2000 file formats. OpenOffice[4] is an office software suite for word processing, spreadsheets, presentations, graphics, and databases. ThunderBird[5] is an

[1]http://www.adempiere.com/
[2]http://filezilla-project.org/
[3]http://www.jedit.org/
[4]http://www.openoffice.org/
[5]http://www.mozilla.org/en-US/thunderbird/

email and news client developed by the Mozilla foundation, while WinMerge[6] is a differencing and merging utility for Windows. In all these systems, most components are developed using the object oriented paradigm. Two of the systems, JEdit and ADempiere, have been developed using Java, while the other four have been developed using C++.

Of the six systems, we have assessed our approach on all class, attribute, and method names of ADempiere, FileZilla client, JEdit, and WinMerge. For OpenOffice and ThunderBird, however, the assessment on all identifiers took quite long, due to the size of the respective ontologies extracted from these systems. Hence, we have randomly sampled 538 and 4205 identifiers from OpenOffice and ThunderBird, respectively, and conducted our analysis on them. Actually, our current implementation of the proposed identifier suggestion methods is not optimized for efficiency. We think that major performance improvements are indeed possible.

5.3.3 Results

The detailed evaluation results of our approach on the six systems are shown in Tables 5.5, 5.6, 5.7, and 5.8. Table 5.8 shows the results for the union of the suggestions provided by *concept prefix* and *neighboring concepts*. The column *Size* in Table 5.5 refers to the number of prefix characters of a term, while it refers to the number of preceding terms in Tables 5.6, 5.7, and 5.8. The three metrics values are computed for all terms which have a prefix with the specified size in case of *term prefix*, and for all identifiers with the number of preceding terms indicated as size in the remaining three cases.

To answer RQ1, we have summarized the results obtained for the average success rate metrics across systems as shown in Figure 5.2. Results indicate that *term prefix* has returned correct suggestions in the majority

[6]http://winmerge.org/

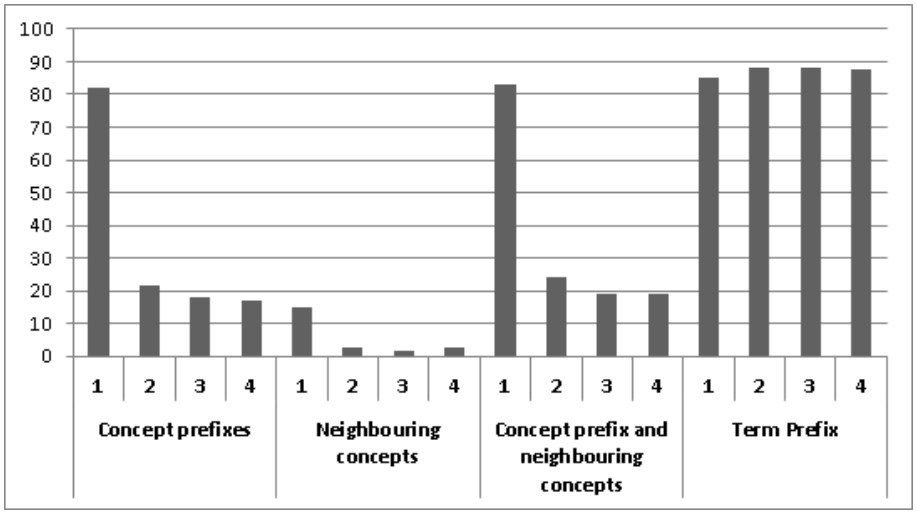

Figure 5.2: Average success rate for each size across systems

of the cases with an average success rate of more than 80% for all prefix sizes across systems. The next highest number of correct suggestions is returned by the union of *concept prefixes and neighboring concepts* with an average success rate higher than 80% for term sizes equal to one and around 20% for a number of terms equal to two, three, and four. On the other hand, *neighboring concepts* has returned very few correct suggestions, which resulted in a success rate below 20%.

Suggestions to identifiers can be provided by keeping track of names defined in other classes of the system. To assess how many of the correct suggestions can be easily retrieved from duplicate identifiers, we have computed the *duplicate completions* metrics (see Tables 5.5, 5.6, 5.7, and 5.8). This metrics measures the number of correct suggestions that can be retrieved from duplicate identifiers. Results indicate that our approach provides substantially more correct suggestions than those retrieved just from duplicate identifiers. In several cases, no suggestion can be provided from duplicate identifiers, while our approach gives correct suggestions. For example, in FileZilla the union of *concept prefixes and neighboring*

concepts has a success rate of 39% while no suggestion can be retrieved from duplicate identifiers, when the number of preceding terms is four.

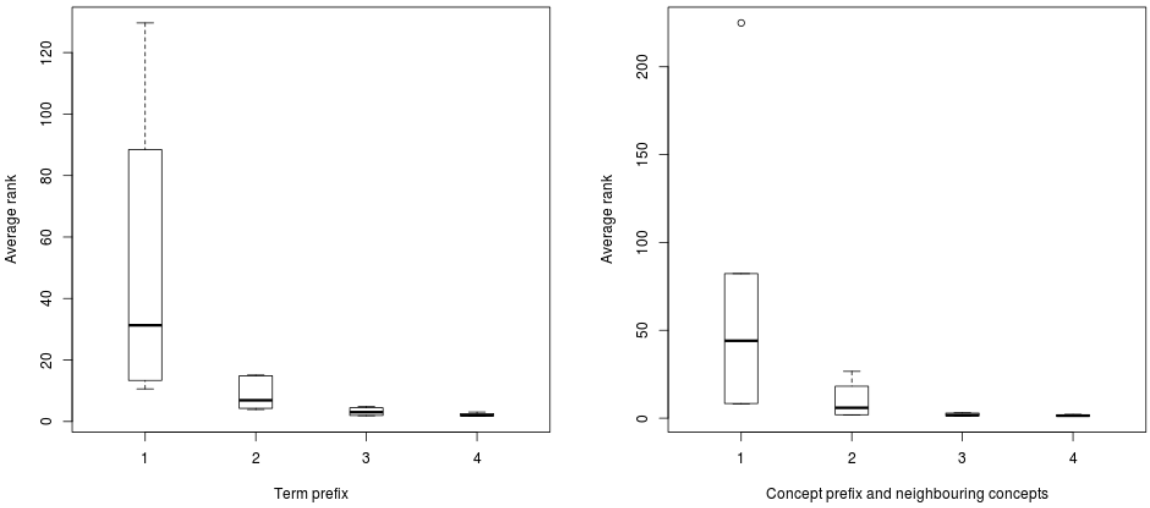

Figure 5.3: Average rank distribution for each size across systems.

RQ2 investigates the quality of our ranking method on correct suggestions. The average top rank for correct suggestions is shown in the last column of Tables 5.5, 5.6, 5.7, and 5.8. In Figure 5.3, we show two boxplots for the two methods which have the overall highest success rate, *term prefix*, and *concept prefixes and neighboring concepts*. In all cases, the median and the distribution of the top ranks across the systems is reduced (*i.e.*, it improves) as the size increases. For correct suggestions provided using three or four characters and *term prefixes*, the average top rank for all systems is below five. While for *term prefixes* with two characters, the average top rank is below five in three systems, FileZilla, JEdit, and WinMerge. The top rank of correct suggestions provided using only one character as a *term prefix* is between 10 and 20 for the above three systems, while in the other three systems such rank is above 40.

The average top rank of correct suggestions provided using *concept prefix*

and neighboring concepts is below five in all systems for three and four sequences of terms, while for term sequences of size two it is below five in three systems, again FileZilla, JEdit, and WinMerge. For the same three systems, the average top rank for a term sequence size of one is below 12. Overall, we can see that the rank of correct completion suggestions gets closer to one (best rank) as the number of prefix characters or preceding terms increases.

Table 5.5: Term prefix results.

Program	Size	Duplicate completions	Success rate	Average rank
ADempiere	1	68.08	92.30	47.91
	2	67.46	93.01	9.45
	3	63.53	90.53	4.46
	4	62.35	90.49	2.56
FileZilla	1	33.55	77.15	10.59
	2	36.29	82.62	3.88
	3	36.81	82.69	1.96
	4	35.74	81.39	1.90
JEdit	1	35.64	77.78	14.67
	2	36.69	79.61	4.27
	3	39.51	85.14	1.87
	4	39.46	85.45	1.65
OpenOffice	1	58.57	97.51	129.65
	2	60.08	98.00	14.88
	3	61.72	97.61	4.89
	4	65.64	97.75	3.07
ThunderBird	1	48.33	92.14	88.44
	2	49.51	93.18	15.10
	3	49.17	93.56	3.77
	4	48.80	93.15	2.11
WinMerge	1	23.38	74.89	13.35
	2	25.66	81.90	4.36
	3	25.66	80.73	2.27
	4	25.70	78.22	1.85

Table 5.6: Neighboring concepts results.

Program	Size	Duplicate completions	Success rate	Average rank
ADempiere	1	12.48	15.34	41.88
	2	2.90	4.54	15.03
	3	5.60	6.37	4.17
	4	12.02	13.93	2.26
FileZilla	1	7.18	9.56	6.68
	2	0	1.06	4.90
	3	0	0.36	1.00
	4	0	0	-
JEdit	1	7.64	10.31	8.14
	2	0.74	2.35	5.13
	3	0	0	-
	4	0	0	-
OpenOffice	1	23.84	31.25	73.5
	2	2.08	2.49	201.00
	3	0	1.11	38.00
	4	0	3.45	6.00
ThunderBird	1	12.48	16.01	55.21
	2	1.58	3.31	35.15
	3	0.13	1.01	1.75
	4	0	0	-
WinMerge	1	6.08	8.83	17.48
	2	1.01	2.48	10.95
	3	0	0.14	3.00
	4	0	0	-

Table 5.7: Concept prefix results.

Program	Size	Duplicate completions	Success rate	Average rank
ADempiere	1	66.32	91.91	6.07
	2	10.69	21.24	15.40
	3	6.70	13.04	1.14
	4	5.11	9.17	1.12
FileZilla	1	28.68	73.95	1.85
	2	5.84	26.46	1.32
	3	0.72	26.09	1.10
	4	0	39.39	1.39
JEdit	1	28.23	69.32	1.87
	2	4.39	25.40	1.17
	3	0.46	18.29	1.11
	4	0	7.55	1.00
OpenOffice	1	59.03	97.22	6.28
	2	1.66	20.75	1.48
	3	1.11	25.56	1.22
	4	3.45	24.14	1.14
ThunderBird	1	49.13	92.81	11.57
	2	7.17	24.12	1.74
	3	4.30	20.76	1.15
	4	2.60	19.05	1.14
WinMerge	1	22.25	67.42	2.08
	2	2.35	13.26	1.20
	3	1.13	5.50	1.08
	4	0	1.44	1.00

Table 5.8: Concept prefix and neighboring concepts results.

Program	Size	Duplicate completions	Success rate	Average rank
ADempiere	1	68.68	94.28	82.25
	2	12.24	23.64	18.19
	3	11.05	17.70	3.06
	4	16.72	22.49	2.16
FileZilla	1	29.42	74.69	8.36
	2	5.84	26.99	1.91
	3	0.72	26.09	2.01
	4	0	39.39	1.50
JEdit	1	29.27	70.37	8.16
	2	4.88	27.26	1.93
	3	0.46	18.29	1.14
	4	0	7.55	1.12
OpenOffice	1	59.03	97.22	224.96
	2	3.73	23.24	26.64
	3	1.11	26.67	2.88
	4	3.45	24.14	1.86
ThunderBird	1	49.13	92.81	76.43
	2	8.70	27.14	7.64
	3	4.43	21.65	1.48
	4	2.60	19.05	1.34
WinMerge	1	22.25	67.42	11.71
	2	3.36	15.55	4.20
	3	1.13	5.64	1.30
	4	0	1.44	1.00

5.3.4 Discussion

Among the four proposed methods, most of the correct suggestions for all sizes is provided using *term prefix*, while only a small number of correct suggestions are provided using *neighboring concepts* (see Figure 5.2). The average success rate of *term prefix* is more than 80% for all sizes. The small number of correct suggestions provided by *neighboring concepts* could be due to the nature of the task we are performing in this investigation, identifier completion, and the relation of the neighboring concepts to the matched concept. If most of the neighboring concepts are in a specialization relationship with the matched concept in the ontology, they are more suited for a replacement rather than a completion. For example, the suggestion *addressFile* provided for the term *file* in Table 5.2 can only be used to replace *file*, if the intention of the developer is to represent the concept *address file*. In our study, however, we do not have the developer in the loop and we focused only on the name completion task. The correctness of suggestions for replacement needs to be further studied by involving the developers in the experiment.

The average success rate has decreased for all types of suggestions except *term prefix*, as the size increases. This is not, in general, the case for the average top ranks of the correct suggestions (see Figure 5.3). In almost all cases, the average top rank improves considerably as the size increases (see Tables 5.5, 5.6, 5.7, and 5.8). One of the few (four) cases where the rank becomes worse as the size increases (from 1 to 2) is observed for ADempiere in the *concept prefix* results (see Table 5.7). Such variations are due to an unequal number of terms in identifiers and to the length of the terms. For the example taken from ADempiere, for instance, most top ranks could be observed for identifiers composed of two terms, associated with a pretty good average rank of 6.07 (for size=1). The top ranks of the suggestions

for identifiers with two terms, however, are not considered when the size used by *concept prefixes* is two or more.

The generally observed inverse relationship between average success rates and average ranks indicates that as more information is provided, our approach ranks successful suggestions closer to the top, but it also excludes many correct suggestions as the size grows, probably because a smaller portion of ontology is taken into account. This inverse relationship, however, is not observed in the suggestions provided by *term prefix*. In *term prefix*, the average top rank improves, as well as the average success rate, as the size grows. In particular, the average top rank of correct suggestions improves by more than three times as the size used by *term prefixes* increases from one to two.

From the experimental results, we can conclude that when two or more characters have been typed, the *term prefixes* method can provide correct suggestions which are ranked quite close to the top. If only one character is typed, the success rate remains high, but the rank of the correct suggestions decreases, hence forcing the developer to scroll a relatively long list of suggestions before the correct one can be found. When one or more preceding terms have already been typed, the *concept prefixes* method can be used quite effectively to obtain correct suggestions. Their rank is generally good if two or more preceding terms have been typed, while it can either be still good or become dramatically worse, depending on the subject system, when only one preceding term is available. The performance of *concept prefixes* can be slightly improved if it is combined with *neighboring concepts*. The amount of correct suggestions which are not just redundant completions indicates that the proposed approach is potentially very useful. Its performance is compatible with the typical cognitive needs and limitations of humans (high success rate and top positioning of correct suggestions) when two or more characters have been typed and when one

(depending on the system) or two preceding terms are available.

5.3.5 Threats to validity

In this study, we have investigated the usability of our approach in only six open source systems. Hence, the study has limited generalizability. To mitigate this limitation, however, we have selected real world open source systems which vary in size and programming language. The systems are also taken from different types of software domains.

Neighboring concepts provide suggestions which can be used for both completion and replacement. In our evaluation, we considered suggestions as correct if they can be used to complete the respective identifier. This could miss suggestions provided by this method which can actually be regarded as correct, if the decision is made by developers. In the future, we plan to further investigate the correctness of the suggestions by including developers in the study. Despite this limitation, as our evaluation is automated and does not have any subjectivity, it is repeatable.

5.4 Conclusion

In this chapter, we have presented an approach which exploits the automatic concept extraction approach discussed in Chapter 2 to help developers address one of the lexicon bad smells, *inconsistent identifier*, discussed in Chapter 4. The proposed approach automatically suggests identifier completions or replacements using the concepts and relations extracted from the existing code by exploiting natural language information captured in the identifiers. To provide the suggestions, we have defined three methods, *term prefix*, *neighboring concepts*, and *synonym*, plus an extension of the *neighboring concepts* method, called *concept prefixes*. In addition, we have presented an approach to rank the suggestions based on their

relevance to the context in which the new identifier is being defined.

We have evaluated two of the proposed methods, *term prefix*, and *neighboring concepts* with its variant *concept prefix*, on identifier completion. The evaluation results show that *term prefix* has high success rate (>80%) in providing correct suggestions; the union of *neighboring concepts* and *concept prefix* performs also very well. The average rank of the correct suggestions, in general, improves as more terms and characters are used to obtain the suggestions. Results indicate overall applicability and usefulness of the proposed approach, when two or more characters have been typed, or when one (depending on the system) or two preceding terms are available in the identifier being written.

Chapter 6

Related works

The programmer's lexicon used in constructing identifiers is exploited by various approaches and tools to support the programmer in understanding the software and carrying maintenance. The level of support tools give to the programmer and the programmer's effort to understand the program depend on the quality of the identifiers involved. In this chapter we describe different approaches developed to support program understanding by exploiting the information captured in the identifiers and approaches aimed at improving the quality of the lexicon used in the identifiers.

6.1 Concept extraction

Program understanding involves mapping existing knowledge of a program to its elements and enriching this knowledge [16, 100, 94]. Mapping (high level) knowledge to source code elements can be achieved using concept location. To support and improve existing concept location techniques, various approaches which exploit the natural language information captured in identifiers have been proposed, while to acquire and enrich knowledge about concepts implemented in a program, various ontology based approaches have been proposed.

A comprehensive survey of the approaches which improve/support con-

cept location using various information, such as dynamic and textual information, are discussed in [41]. Here we focus on those approaches which exploit textual information. Marcus *et al.* [83] and Gay *et al.* [47] used information retrieval (IR) based approaches to reduce the effort required to understand and locate the part of the source code that needs to be changed. In their approach, they used Latent Semantic Indexing (LSI) to convert source code documents (composed of identifier terms) and user query to their respective mathematical representations. Such formalizations are then used to compute the similarity between them and get a ranked list of source code documents (by decreasing similarity). The results of these approaches are dependent on the quality of user queries. To assess the quality of queries prior to using them and reduce the effort and time required to assess the results, Haiduc *et al.* [55, 54] have proposed query assessment metrics. The metrics are used to evaluate and classify the query as *high-performing query* and *low-performing query* prior to its execution. Cleary *et al.* [34] have proposed an approach to expand queries using information flow and term co-occurrence information in the system documentation to identify terms which can be used to expand the queries.

To reduce the developers' effort in locating concepts using IR techniques, Poshyvanyk and Marcus [95] have combined Formal Concept Analysis (FCA) with LSI. The approach produces a concept lattice using the most relevant attributes (terms) selected from the top n ranked documents (methods). The evaluation of their approach has shown that the concept lattice is effective in grouping relevant information. Poshyvanyk *et al.* [97] have integrated the Google Desktop Search Engine[1] into Eclipse, to take advantage of the engine's features and to facilitate searching of the source code. Grant *et al.* [50] have also proposed automated concept location using Independent Component Analysis (ICA). In this approach, the authors

[1] http://googledesktop.blogspot.com/

use ICA to identify statistically independent signals which correspond to concepts. The concepts are then mapped to methods which are related in functionality.

Fry *et al.* [45] have defined a set of rules and algorithms which uses the information captured in method names to automatically extract a natural language representation of the source code, called action oriented identifier graph (AOIG). The AOIG is used in *Find Concept* to support concern location and understanding [105]. *Find Concept* supports developers by automatically searching AOIG and re-formulating the initial query. Hill *et al.* [58] have also presented an approach that supports programmers in (re-)formulating queries and locating program components. The approach identifies the context of query words in the source code by extracting and generating hierarchies of natural language phrases from method and field signatures. They have compared the context search approach to *Find Concept* [105] to see if natural language phrases beyond verbs and direct objects improve the searching capabilities of *Find Concept*. Results indicate that context search significantly outperforms *Find Concept* in terms of effort and effectiveness.

Petrenko *et al.* [94] have used fragments of ontologies to partially comprehend a program and locate concepts in the source code. In this approach, programmers first build an initial ontology fragment based on their previous knowledge of the domain and the information contained in a change request. They, then, formulate a query based on the knowledge captured in the initial ontology fragment. By looking at the results of the query and available documentations, they extend the ontology fragment and repeat the process until they are satisfied with the result. This approach is reported to reduce the search space and the number of missed program elements that implement a concept. Our approach used to extract concept from the source code is in line with the work by Petrenko *et al.* [94],

the main difference being that we use NLP and structural information to automatically extract ontology concepts, instead of relying on ontology fragments constructed manually by programmers. Nonnen *et al.* [93] have defined heuristics to identify source code locations where terms are defined. The heuristics are evaluated on 8000 manually evaluated samples and achieved a precision of 75%.

Ratiu *et al.* [102] have proposed an approach to automatically extract a domain ontology from different APIs that are implemented to address similar problems in a given domain. Concepts and relations of the domain ontology are retrieved using a graph matching algorithm which is applied to a graph representation of the APIs. The matching algorithm identifies concepts using similarity between terms of identifiers found in the APIs and maps structural relations of APIs to ontological relations. This approach extracts all prevalent domain concepts found in the different APIs considered for the extraction. However, unlike our concept extraction approaches (see Sections 2.2 and 2.3), it depends on the existence of several, similar APIs, and on the chance of finding two related concepts having similar names and connected with similar paths in different APIs. Falleri *et al.* [43] have proposed an approach to automatically extract and organize concepts from identifiers in a WordNet-like structure which is referred to as *lexical view*. Their approach is similar to our NLP based concept extraction approach (see Section 2.2), but differs on the technique used to identify the relations and number of relations used. They have used the longest common prefix to identify common concept between two identifier term lists sorted by dominance order, and create a hypernymy/hyponymy relation, while we have used parse trees produced for the identifier term lists to identify concepts and inter-concept relations. In addition, our approach considers more relations than hypernymy/hyponymy.

Maskeri *et al.* [85] have proposed an LDA (Latent Dirichlet Allocation)

based human assisted approach to extract topics from a system. The experiments they conducted have shown that their approach is successful in extracting some of the domain topics. Support Vector Machines (SVM) and subgraph identification algorithms have been used by Carey and Gannod [32], and Hsi *et al.* [62], respectively, to identify the core concepts in a system. Carey and Gannod [32] have used object oriented metrics to train an SVM classifier and identify classes related to core concepts, while Hsi *et al.* [62] applied graph analysis techniques to ontologies extracted from the interface of a program. The ontology extraction is carried out by first manually building an interface map from the user interface. With a similar objective as these works, we have conducted concept filtering to identify domain concepts (see Chapter 3). Our approach, however, utilizes different techniques both to extract and filter ontologies, with the aim of substantially reducing the manual effort involved, by resorting to fully automated techniques for both steps with the exception of interactive keyword based filtering (see Section 3.2), which requires a very limited amount of human intervention.

6.2 Identifier quality improvement

Identifiers are one of the main sources of information used during software understanding and maintenance [9]. Hence, their quality has a direct impact on maintenance and understanding. Various works have proposed different approaches which can be used to assess/identify poor quality identifiers in the source code, to improve and to maintain their quality, and to predict possibly faulty parts of the source code.

Anquetil and Lethbrige [8] suggest to asses the quality of the identifiers prior to relying on them. In their work, the authors have defined what a reliable naming convention is and proposed a framework based on similar-

ity metrics to identify if the naming is reliable or not. Lawrie *et al.* [76] have developed the QALP (Quality Assessment using Language Processing) tool to assess the effort required to understand a program, identify parts of a program that need preventive maintenance and make related changes. QALP is based on the assumption that a high quality code will have comments that give a good description of the code. The tool uses information retrieval techniques to carry out the assessment.

Besides the assessment, quality of identifiers is used to investigate and identify part of the source code which is likely to be problematic. Binkley *et al.* [17] have used QALP to predict number of faults in a module. Butler *et al.* [25, 26] have studied the relationship between identifiers violating naming guidelines and code quality issues reported by FindBugs[2], and found that poor quality identifiers are associated with lower quality source code. In a similar line, Boogerd and Moonen [21, 22] have studied the relationship between the MISRA C 2004 standards which include identifier naming guidelines and issues found in the issue tracking system. The result of their study indicates that only a subset of the standards correlates with the issues and suggest to adhere to a customized, project specific set of rules to decrease the probability of fault occurrence. Arnaoudova *et al.* [12] have defined *term entropy* and *context coverage* measures to study the relationship between the terms composing identifiers and fault proneness. Term entropy is used to measure the "physical" dispersion of a term in a program, while context coverage is used to measure the "conceptual" dispersion of the entities in which the term appears. In their study, they have showed that high term entropy and high context coverage could help to locate fault prone methods.

When the assessment of the code indicates that there are low quality identifiers, locating the identifiers with problems is important. In this re-

[2]http://findbugs.sourceforge.net/

gard, Ratiu and Deissenboeck [101] have presented a mapping between a graph like representation of domain knowledge and the program. Ratiu and Deissenboeck's approach identifies semantic defects in program element names which are categorized into four groups: fatal polysemy, polysemy, logical redundancy, and synonymy. Høst and Østvold [61] have also defined naming bugs and used automatically extracted rules from method names to identify the bugs in most common method names written in Java. In addition, they have presented an approach to automatically suggest more suitable names which can be used to fix the naming bugs. In a similar line, Arnaoudova *et al.* [11] have recently defined a family of linguistic anti-patterns which are inconsistencies between names of methods and attributes, and the corresponding definitions and comments. They have classified the anti-patterns in to six categories based on what the names say they do and what the corresponding implementations actually do.

To improve the quality of identifiers, Deissenboeck and Pizka [39] proposed a formal model to consistently and concisely name identifiers. To achieve consistency, the model defines a rule which requires a bijective mapping between a set of names and a set of concepts. By applying this rule while giving names to program elements, programmers eliminate the inconsistency that might arise due to homonyms and synonyms of lexicon in an identifier. To ensure conciseness of names, the authors have also defined two more rules related to correctness and conciseness.

The bijective mapping between concepts and names requires human intervention. Hence, constructing such mapping is difficult especially for large existing systems. Considering this weakness of the model, Lawrie *et al.* [74] proposed a syntactic approach to concisely and consistently name identifiers. Lawrie *et al.* 's approach defines two rules based on containment of soft-words of an identifier in another. The first rule states that there is a syntactic synonym consistency violation, if an identifier's soft-words

are contained in another identifier in the same sequence. While the second rule, syntactic conciseness, states that there is violation of this rule if two or more identifiers contain soft-words of another identifier in the same order. This approach has limitations in discovering inconsistencies that arise from homonyms and abbreviations (e.g. abspos and pixel_absolute_position) [73]. In addition, it gives false positives when a containing identifier has a different meaning due to the other composing word(s) or is used in a hierarchy (for example, inheritance) of the system. These false positives are regarded in the paper as addressable, by considering parts of speech.

The expansion of abbreviations found in identifiers increases the recall of Lawrie *et al.* 's approach [74] which is used to identify inconsistencies [73]. In addition, expanding abbreviations and acronyms, to their appropriate full length words improves programmer's code understanding [75]. To expand abbreviations, similar approaches are suggested in different works [70, 73, 57, 71, 36]. The approaches in general follow three steps to expand abbreviations or acronyms. In the first step, they identify non-dictionary words which are potential abbreviations or acronyms by splitting the identifiers. Then, they try to identify possible expansions from a list of words in pre/user-defined dictionaries [70, 73, 36] or automatically extracted word list [57] and phrases [73], or by looking at patterns of text in the code [57, 71]. In the last step of Laitinen *et al.* 's approach [70], the user is asked to choose the appropriate substitution while Lawrie *et al.* 's tool [73] automatically returns a substitute only when there is a single match. In Hill *et al.* 's approach [57] the tool computes the most frequent expansion and suggests the one with the highest score as the correct expansion. Corazza *et al.* 's approach [36] uses Baeza-Yates and Perleberg's string matching algorithm [13] to match the abbreviations with dictionary words and considers the expansion with lowest cost as the suggested expansion, while Lawrie and Binkley [71] use a similarity measure based on Google

data set [23] to select the most likely expansion.

To make the identifiers in the programs more meaningful, Caprile and Tonella [30] have proposed restructuring the names. The restructuring of identifiers is carried out on two different aspects of identifiers, lexicon and syntax. The words used to construct identifiers are restructured by using standard lexicon that can be extracted from the source code or provided by the company. To syntactically restructure the identifiers, particularly function identifiers, they have proposed to use a standard grammar that can again be proposed by the company or derived from the source code. The standard grammar is also used to further study the syntactical behavior of method names and identify the basic blocks needed by programmers to build new identifiers [31]. In addition, it improves the efficiency of tools (example *Find-Concept* in [105]) that rely on the syntactical composition of identifiers. Binkley *et al.* [18] have defined a template similar to what is defined in Section 2.1.2 to improve the accuracy of parts-of-speech (POS) tagger on field names. From the POS tags, they have defined field name formation rules which can be used to support improved naming. Their approach is similar to ours in improving the accuracy of natural language taggers/analyzers, but we differ on the application of the results. They have used the POS tags to define rules which can be used to improve naming while we use the POS tags and dependency relations to identify concepts and relations among concepts to build an ontology.

The aforementioned approaches present different rules and techniques in which identifiers become more readable and hence comprehensible. However, programmers might not follow them all the time due to various reasons, like the additional effort required or time constraints.

In our approach (see Chapter 4), we introduce naming anti-patterns which are bad smells of the lexicon that reduce the quality of the names used and compromise the comprehensibility of the source code. We pro-

posed heuristics to identify such lexicon bad smells, detect them and suggest refactoring techniques to improve the quality of the lexicon.

Chapter 7

Conclusions and future works

7.1 Conclusions

During software evolution developers consult various artifacts related to the software (*e.g.*, requirement document, source code, etc.) to understand the software and make changes. For most software, however, all artifacts except the source code are often unavailable or not up-to-date. Hence, developers mostly rely on the information captured in the source code to understand and evolve the software. Information such as the concepts implemented in the source code and where they are implemented is mainly communicated through the identifiers.

Program understanding involves (re-)building the mental model of the program as it is represented in the source code and identifiers play an important role in this. The effort involved in the understanding process heavily depends on the quality of the programmers' lexicon used to construct the identifiers.

In this work, we have presented two approaches, which are based on (i) the natural language information captured in the identifiers; and (ii) the structural information of the code, to extract concepts and inter-concept relationships. The concepts extracted have been evaluated on the support they give to a program understanding task, concept location. The results

indicate that combining the concepts and inter-concept relations extracted using both approaches give better practical and statistically significant results.

To extract the natural language information we have used natural language analyzers. Natural language analyzers, however, are mainly developed to work with sentences which are different from the term sequences generated after splitting identifiers. In this regard, we have proposed two approaches where (i) heuristics are used to convert the term sequences constructed from identifiers to sentences and adapt them to the form expected by natural language analyzers; and (ii) a natural language analyzer training set which resembles term sequences constructed from identifiers is built from documentations close to the software under consideration and used to adapt the analyzer to work with identifier term sequences. In our study, the extracted ontologies have been evaluated based on the support they give to a program understanding task, concept location. The results show that they are equally good in improving the effectiveness of queries formulated in concept location. They also indicate that the extracted natural language information is essential to improve the performance of concept location, regardless of the used analyzer.

The concepts extracted following our approach are composed of both domain concepts and implementation concepts, as in the source code from which they are extracted. In some cases, the objective of the developer can be to acquire the domain information captured in the source code and understand the relationships among the domain concepts. In such cases, separating the domain concepts from the implementation concept facilitates understanding. In this regard, we have presented three information retrieval (IR) approaches to filter domain concepts: non-interactive keyword based filtering, interactive keyword based filtering, and topic based filtering. Among the three approaches interactive keyword based filtering,

which requires minimal human involvement was found to be effective in filtering domain concepts.

Besides supporting the extraction of concepts from the source code, we have studied identifier naming patterns and characteristics, and defined a publicly available catalog of lexicon bad smells, that may have a negative impact on the effort put to understand a program. Lexicon bad smells (LBS) are problems related to identifier construction. The catalog contains definition of the LBS, symptom, exceptions to the LBS, a heuristics on how to detect them, and a suggestion on how to refactor them. We have also developed a publicly available suit of tools which implement the heuristics proposed to detect the LBS. The evaluation conducted to assess the accuracy of the detectors shows that the implemented detectors are good approximations of the corresponding LBS definitions.

We have conducted an experiment to investigate if LBS have an impact on concept location. The result shows that LBS can be an important factor to consider when performing IR-based concept location and that refactoring LBS can have a significant positive impact on the task. In particular, if the lexicon of the system being maintained is known to be of relatively low quality the benefits of lexicon smell removal on concept location are expected to be quite significant. LBS have been also assessed in terms of the contribution they can give to fault prediction. Fault prediction is one of the mechanisms used to predict faulty parts of the software and take measures to minimize post release maintenance costs and provide quality software. The results of our assessment shows that using LBS together with structural metrics improves fault prediction in the majority of the cases.

LBS are problems which are introduced in the code while defining identifiers. In this work, we have proposed an approach which can assist developers in (re-)defining identifiers and avoid inconsistencies. The approach

exploits the natural language based concept extraction approach described above and uses three methods, *term prefix*, *neighboring concepts*, and *synonym*, plus an extension of the *neighboring concepts* method, called *concept prefixes* to provide a ranked list of terms to complete or replace an identifier being defined. The results of the evaluation conducted on identifier completion using two of the proposed methods, *term prefix*, and *neighboring concepts* with its variant *concept prefix*, show the applicability and usefulness of the approach in providing correct suggestions. This indicates that assisting developers in extracting knowledge already captured in the source code will allow them also to name concepts in the source code consistently.

7.2 Future works

In this work, we have presented two approaches to extract concepts and inter-concept relations from the source code and evaluated the usability of the concepts in a program understanding task, concept location. In the evaluation, we have used the results of the queries formulated with and without the use of the extracted concepts. Recently, Haiduc *et al.* [55, 54] have proposed query assessment metrics which are used to classify if a query is effective or not, prior to its execution. We believe combining Haiduc *et al.* 's query classification approach with our approach to select the effective queries will benefit developers by saving time and effort spent in formulating an effective query.

Besides the support the extracted concepts and the relations among them give to concept location, we believe that they can be used to support other tasks such as prediction and assessment of code quality, similar to the works of Poshyvanyk *et al.* [96] and Marcus *et al.* [82]. We also plan to investigate such dimensions in the future.

To filter domain concepts from the ontologies built using our approaches,

we have applied information retrieval (IR) techniques. While using such techniques, we have focused only on filtering the domain concepts and we have used a simple rule to keep the inter-concept relationship: *a relation is kept if concepts at both ends of the relation are found to be domain concepts*. In the future, we plan to investigate heuristics which can be used to infer possible relationships between domain concepts which are connected indirectly, through other relations connecting intermediate implementation concepts.

To improve the quality of identifiers, we have developed an approach which exploits the concepts extracted following our approach and we suggest terms which can be used for replacing or completing the identifiers being defined. In the future, we plan to investigate other approaches which can be used to avoid other types of LBS and we plan to integrate them with our identifier suggestion approach, to provide developers with a comprehensive identifier naming assistance tool. Besides, we plan to extend the LBS catalog with new bad smells that include also more types of identifiers.

Bibliography

[1] Surafel Lemma Abebe, Sonia Haiduc, Andrian Marcus, Paolo Tonella, and Giuliano Antoniol. Analyzing the evolution of the source code vocabulary. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, CSMR '09, pages 189–198, Washington, DC, USA, 2009. IEEE Computer Society.

[2] Surafel Lemma Abebe, Sonia Haiduc, Paolo Tonella, and Andrian Marcus. Lexicon bad smells in software. In *Proceedings of the 16th Working Conference on Reverse Engineering*, WCRE '09, pages 95–99, Washington, DC, USA, 2009. IEEE Computer Society.

[3] Surafel Lemma Abebe, Sonia Haiduc, Paolo Tonella, and Andrian Marcus. The effect of lexicon bad smells on concept location in source code. In *Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation*, WCRE '11, pages 125 –134. IEEE Computer Society, 2011.

[4] Surafel Lemma Abebe and Paolo Tonella. Natural language parsing of program element names for concept extraction. In *Proceedings of IEEE International Conference on Program Comprehension*, ICPC '10, pages 156–159. IEEE Computer Society, 2010.

[5] Surafel Lemma Abebe and Paolo Tonella. Towards the extraction of domain concepts from the identifiers. In *Proceedings of the 18th*

Working Conference on Reverse Engineering, WCRE '11, pages 77–86, Washington, DC, USA, 2011. IEEE Computer Society.

[6] Surafel Lemma Abebe and Paolo Tonella. Automated identifier completion and replacement. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*, CSMR '13, pages 263–272. IEEE Computer Society, 2013.

[7] Surafel Lemma Abebe, Paolo Tonella, Venera Arnaoudova, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Can lexicon bad smells improve fault prediction? In Rocco Oliveto and Denys Poshyvanyk, editors, *Proceedings of the 19th Working Conference on Reverse Engineering*, WCRE '12. IEEE Computer Society Press, 2012.

[8] Nicolas Anquetil and Timothy Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '98, pages 4–. IBM Press, 1998.

[9] Giuliano Antoniol, Yann-Gaël Guéhéneuc, Ettore Merlo, and Paolo Tonella. Mining the lexicon used by programmers during software evolution. In *Proceedings of IEEE International Conference on Software Maintenance*, ICSM '07.

[10] Erik Arisholm, Lionel C. Briand, and Magnus Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. In *Proceedings of the The 18th IEEE International Symposium on Software Reliability*, ISSRE '07, pages 215–224, Washington, DC, USA, 2007. IEEE Computer Society.

[11] Venera Arnaoudova, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. A new family of software anti-patterns: Lin-

guistic anti-patterns. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*, CSMR '13.

[12] Venera Arnaoudova, Laleh Eshkevari, Rocco Oliveto, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Physical and conceptual identifier dispersion: Measures and relation to fault proneness. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–5, Washington, DC, USA, 2010. IEEE Computer Society.

[13] Ricardo A. Baeza-yates and Chris H. Perleberg. Fast and practical approximate string matching. In *3rd Annual Symposium in Combinatorial Pattern Matching*, CPM '92, pages 185–192, London, UK, UK, 1992. Springer-Verlag.

[14] Robert M. Bell, Thomas J. Ostrand, and Elaine J. Weyuker. Does measuring code change improve fault prediction? In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, Promise '11, pages 2:1–2:8, New York, NY, USA, 2011. ACM.

[15] Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(1):pp. 289–300, 1995.

[16] T.J. Biggerstaff, B.G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93, pages 482 –498, Los Alamitos, CA, USA, may 1993. IEEE Computer Society Press.

[17] D. Binkley, H. Feild, D. Lawrie, and M. Pighin. Software fault prediction using language processing. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, TAICPART-MUTATION '07, pages 99–110, Washington, DC, USA, 2007. IEEE Computer Society.

[18] Dave Binkley, Matthew Hearn, and Dawn Lawrie. Improving identifier informativeness using part of speech information. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 203–206, New York, NY, USA, 2011. ACM.

[19] David M. Blei. Probabilistic topic models. *Communications of ACM*, 55(4):77–84, 2012.

[20] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, 2003.

[21] Cathal Boogerd and Leon Moonen. Assessing the value of coding standards: An empirical study. In *Proceedings of the 24th IEEE International Conference on Software Maintenance*, ICSM '08, pages 277 –286, Washington, DC, USA, 2008. IEEE Computer Society.

[22] Cathal Boogerd and Leon Moonen. Evaluating the relation between coding standard violations and faults within and across software versions. In *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 41–50, Washington, DC, USA, 2009. IEEE Computer Society.

[23] Thorsten Brants and Alex Franz. *Web 1T 5-gram Version 1. Linguistic Data Consortium, Philadelphia, PA.* Philadelphia, PA, 2006.

[24] Lionel C. Briand, John W. Daly, and Jürgen Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, July 1998.

[25] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Relating identifier naming flaws and code quality: An empirical study. In *Proceedings of the 16th Working Conference on Reverse Engineering*, WCRE '09, pages 31–35, Washington, DC, USA, 2009. IEEE Computer Society.

[26] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 156–165, Washington, DC, USA, 2010. IEEE Computer Society.

[27] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Mining java class naming conventions. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 93–102, Washington, DC, USA, 2011. IEEE Computer Society.

[28] Deng Cai, Qiaozhu Mei, Jiawei Han, and Chengxiang Zhai. Modeling hidden topics on document manifold. In *Proceeding of the 17th ACM conference on Information and knowledge management*, CIKM '08, pages 911–920, New York, NY, USA, 2008. ACM.

[29] B. Caprile and P. Tonella. Restructuring program identifier names. In *Proceedings of the 16th International Conference on Software Maintenance*, ICSM '00, pages 97 –107. IEEE Computer Society, 2000.

[30] Bruno Caprile and Paolo Tonella. Restructuring program identifier names. In *Proceedings of International Conference on Software Maintenance*, ICSM '00.

[31] Bruno Caprile and Paolo Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, WCRE '99, pages 112–, Washington, DC, USA, 1999. IEEE Computer Society.

[32] M.M. Carey and G.C. Gannod. Recovering concepts from source code with automated concept identification. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, ICPC '07, pages 27–36, Washington, DC, USA, 2007. IEEE Computer Society.

[33] S. R. Chidamber and C. F. Kemerer. *IEEE Transactions in Software Engineering*, 20(6):476–493, 1994.

[34] Brendan Cleary, Chris Exton, Jim Buckley, and Michael English. An empirical analysis of information retrieval based concept location techniques in software comprehension. *Empirical Software Engineering*, 14:93–130, 2009.

[35] Michael L. Collard, Jonathan I. Maletic, and Andrian Marcus. Supporting document and data views of source code. In *Proceedings of the ACM symposium on Document engineering*, DocEng '02.

[36] Anna Corazza, Sergio Di Martino, and Valerio Maggio. Linsen: An efficient approach to split identifiers and expand abbreviations. In *Proceedings of the 28th IEEE International Conference of Software Maintenance*, ICSM '12.

[37] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories*, MSR '10, pages 31–41. IEEE Computer Society, 2010.

[38] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[39] Florian Deißenböck and Markus Pizka. Concise and consistent naming. In *Proceedings of the 13th International Workshop on Program Comprehension*, IWPC '05, pages 97–106, Washington, DC, USA, 2005. IEEE Computer Society.

[40] Sergio Di Martino, Filomena Ferrucci, Carmine Gravino, and Federica Sarro. A genetic algorithm to configure support vector machines for predicting fault-prone components. In *Proceedings of the International Conference on Product-Focused Software Process Improvement*, PROFES '11, pages 247–261. Springer-Verlag, 2011.

[41] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, pages n/a–n/a, 2012.

[42] Karim O. Elish and Mahmoud O. Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649–660, 2008.

[43] J.-R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, and M. Dao. Automatic extraction of a wordnet-like identifier network from software. In *Proceedings of the 18th IEEE International Conference on Program Comprehension*, ICPC '10, pages 4–13, Washington, DC, USA, 2010. IEEE Computer Society.

[44] Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, Cambridge, MA, 1998.

[45] Zachary P. Fry, David Shepherd, Emily Hill, Lori L. Pollock, and K. Vijay-Shanker. Analysing source code: looking for useful verbdirect object pairs in all the right places. *IET Software*, 2(1):27–36, 2008.

[46] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Communications of ACM*, 30(11):964–971, November 1987.

[47] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the use of relevance feedback in ir-based concept location. In *Proceedings of the 25th IEEE International Conference on Software Maintenance*, ICSM '09, 2009.

[48] Jesús Giménez and Lluís Màrquez. Fast and accurate part-of-speech tagging: The svm approach revisited. In *Recent Advances in Natural Language Processing III*, RANLP '03.

[49] Jesús Giménez and Lluís Màrquez. SVMTool: A general POS tagger generator based on Support Vector Machines. In *Proceedings of 4th International Conference on Language Resources and Evaluation*, LREC '04.

[50] Scott Grant, James R. Cordy, and David Skillicorn. Automated concept location using independent component analysis. In *Proceedings of the 15th Working Conference on Reverse Engineering*, WCRE '08, pages 138–142, Washington, DC, USA, 2008. IEEE Computer Society.

[51] T. L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences*, 101(Suppl. 1):5228–5235, April 2004.

[52] Yann-Gaël Guéhéneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 172–181, Washington, DC, USA, 2004. IEEE Computer Society.

[53] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.

[54] S. Haiduc, G. Bavota, R. Oliveto, A. Marcus, and A. De Lucia. Evaluating the specificity of text retrieval queries to support software engineering tasks. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12.

[55] Sonia Haiduc, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and Andrian Marcus. Automatic query performance assessment during the retrieval of software artifacts. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE '12.

[56] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, Washington, DC, USA, 2009. IEEE Computer Society.

[57] Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori L. Pollock, and K. Vijay-Shanker. Amap: automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *Proceedings of the international working conference on Mining software repositories*, MSR '08, pages 79–88, New York, NY, USA, 2008. ACM.

[58] Emily Hill, Lori L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, 2009.

[59] Thomas Hofmann. Probabilistic latent semantic indexing. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '99, pages 50–57, New York, NY, USA, 1999. ACM.

[60] Einar W. Høst and Bjarte M. Østvold. Debugging method names. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*, ECOOP '09, pages 294–317, Berlin, Heidelberg, 2009. Springer-Verlag.

[61] Einar W. Høst and Bjarte M. Østvold. Debugging method names. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*, ECOOP '09, pages 294–317, Berlin, Heidelberg, 2009. Springer-Verlag.

[62] I. Hsi, C. Potts, and M. Moore. Ontological excavation: unearthing the core concepts of the application. In *Proceedings of the Working Conference on Reverse Engineering*, WCRE '03, pages 345 – 353. IEEE Computer Society, 2003.

[63] Thorsten Joachims. Advances in kernel methods. chapter Making large-scale support vector machine learning practical, pages 169–184. MIT Press, Cambridge, MA, USA, 1999.

[64] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Ken ichi Matsumoto, Bram Adams, and Ahmed E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *Proceedings of*

the IEEE International Conference on Software Maintenance, ICSM '10, pages 1–10. IEEE Computer Society, 2010.

[65] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, June 2012.

[66] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[67] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics*, ACL '03, pages 423–430, Morristown, NJ, USA, 2003. Association for Computational Linguistics.

[68] Dan Klein and Christopher D. Manning. Fast exact inference with a factored model for natural language parsing. In *Proceedings of Advances in Neural Information Processing Systems 15*, NIPS '03, pages 3–10. MIT Press, 2003.

[69] Segla Kpodjedo, Filippo Ricca, Philippe Galinier, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Design evolution metrics for defect prediction in object oriented systems. *Empirical Software Engineering*, 16(1):141–175, 2011.

[70] Kari Laitinen, Jorma Taramaa, Markky Heikkila, and Neil C. Rowe. Enhancing maintainability of source programs through disabbreviation. *Journal of Systems and Software*, 37(2):117–128, 1997.

[71] D. Lawrie and D. Binkley. Expanding identifiers to normalize source code vocabulary. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, ICSM '11.

[72] D. Lawrie, D. Binkley, and C. Morrell. Normalizing source code vocabulary. In *Proceedings of the 17th Working Conference on Reverse Engineering*, WCRE '10.

[73] Dawn Lawrie, Henry Feild, and David Binkley. Extracting meaning from abbreviated identifiers. In *Proceedings of the 7th IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '07.

[74] Dawn Lawrie, Henry Feild, and David Binkley. Syntactic identifier conciseness and consistency. In *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '06, pages 139–148, Washington, DC, USA, 2006. IEEE Computer Society.

[75] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, pages 303–318, 2007.

[76] D.J. Lawrie, H. Feild, and D. Binkley. Leveraged quality assessment using information retrieval techniques. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC '06, pages 149 –158, Washington, DC, USA, 0-0 2006. IEEE Computer Society.

[77] Dekang Lin. Dependency-based evaluation of MINIPAR. In *Proceedings of International Workshop on the Evaluation of Parsing Systems*, WEPS '98.

[78] Dekang Lin. LaTaT: Language and text analysis tools. In *Proceedings of 1st International Conference on Human Language Technology Research*, HLT '01, pages 1–6, Stroudsburg, PA, USA, 2001. Association for Computational Linguistics.

[79] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice Hall, 1994.

[80] J.I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, May 2001.

[81] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions Software Engineering*, 34(2):287–300, 2008.

[82] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions Software Engineering*, 34(2):287–300, 2008.

[83] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 214–223, Washington, DC, USA, 2004. IEEE Computer Society.

[84] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Journal of Computational Linguistics - Special issue on using large corpora: II*, 19(2):313–330, 1993.

[85] Girish Maskeri, Santonu Sarkar, and Kenneth Heafield. Mining business topics in source code using latent dirichlet allocation. In *Proceedings of the India software engineering conference*, ISEC '08.

[86] B.W. Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 405(2):442–451, 1975.

[87] Thilo Mende and Rainer Koschke. Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, PROMISE '09, pages 7:1–7:10, New York, NY, USA, 2009. ACM.

[88] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Constructing precise object relation diagrams. In *Proceedings of the International Conference on Software Maintenance*, ICSM '02, 2002.

[89] George A. Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, November 1995.

[90] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 284–292, New York, NY, USA, 2005. ACM.

[91] Joakim Nivre. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies*, IWPT '03.

[92] Joakim Nivre, Johan Hall, and Jens Nilsson. MaltParser: A data-driven parser-generator for dependency parsing. In *Proceedings of the 5th International Conference on Language Resources and Evaluation*, LREC '06.

[93] Jan Nonnen, Daniel Speicher, and Paul Imhoff. Locating the meaning of terms in source code research on "term introduction". In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, pages 99–108, Washington, DC, USA, 2011. IEEE Computer Society.

[94] Maksym Petrenko, Václav Rajlich, and Radu Vanciu. Partial domain comprehension in software evolution and maintenance. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 13–22, Washington, DC, USA, 2008. IEEE Computer Society.

[95] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, ICPC '07, 2007.

[96] Denys Poshyvanyk and Andrian Marcus. The conceptual coupling metrics for object-oriented systems. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, ICSM '06, pages 469–478, Washington, DC, USA, 2006. IEEE Computer Society.

[97] Denys Poshyvanyk, Maksym Petrenko, Andrian Marcus, Xinrong Xie, and Dapeng Liu. Source code exploration with google. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, ICSM '06, pages 334–338, Washington, DC, USA, 2006. IEEE Computer Society.

[98] V. Rajlich. Intensions are a key to program comprehension. In *Proceedings of IEEE 17th International Conference on Program Comprehension*, ICPC '09, pages 1–9, may 2009.

[99] Vaclav Rajlich and Prashant Gosavi. Incremental change in object-oriented programming. *IEEE Software*, 21(4):62–69, July 2004.

[100] Václav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *Proceedings of the 10th International Workshop on Program Comprehension*, IWPC '02, pages 271–280, Washington, DC, USA, 2002. IEEE Computer Society.

[101] D. Ratiu and F. Deissenboeck. How programs represent reality (and how they don't). In *Proceedings of 13th Working Conference on Reverse Engineering*, WCRE '06, pages 83 –92. IEEE Computer Society, oct. 2006.

[102] Daniel Ratiu, Martin Feilkas, and Jan Jürjens. Extracting domain ontologies from domain specific APIs. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, CSMR '08, pages 203–212, Washington, DC, USA, 2008. IEEE Computer Society.

[103] Filippo Ricca, Emanuele Pianta, Paolo Tonella, and Christian Girardi. Improving web site understanding with keyword-based clustering. *Journal Softw. Maint. Evol.*, 20:1–29, January 2008.

[104] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. How do professional developers comprehend software? In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE '12, pages 255–265, Piscataway, NJ, USA, 2012. IEEE Press.

[105] David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, AOSD '07.

[106] H. M. Sneed. Object-oriented cobol recycling. In *Proceedings of the 3rd Working Conference on Reverse Engineering*, WCRE '96, pages 169–, Washington, DC, USA, 1996. IEEE Computer Society.

[107] P. Tonella and S. L. Abebe. Code quality from the programmer's perspective. In *Proceedings of XII Advanced Computing and Analysis Techniques in Physics Research*, ACAT '08, 2008.

[108] Paolo Tonella and Alessandra Potrich. *Reverse Engineering of Object Oriented Code*. Springer-Verlag, Berlin, Heidelberg, New York, 2005.

[109] Sharath K. Udupa, Saumya K. Debray, and Matias Madou. Deobfuscation: Reverse engineering obfuscated code. In *Proceedings of the 12th Working Conference on Reverse Engineering*, WCRE '05, pages 45–54, Washington, DC, USA, 2005. IEEE Computer Society.

[110] Michael Uschold and Michael Gruninger. Ontologies and semantics for seamless connectivity. *SIGMOD Rec.*, 33:58–64, December 2004.

[111] V.N. Vapnik. *Statistical learning theory*. John Wiley & Sons, New York, 1998.

[112] Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. Comparing the effectiveness of several modeling methods for fault prediction. *Empirical Software Engineering*, 15(3):277–295, 2010.

[113] James Q. Wilson and George L. Kelling. The police and neighborhood safety: Broken windows. *Atlantic Monthly*, 127:29–38, 1982.

[114] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Computer Science. Springer Berlin Heidelberg, 2012.

[115] Xiaohua Zhou, Xiaodan Zhang, and Xiaohua Hu. Dragon toolkit: Incorporating auto-learned semantic knowledge into large-scale text retrieval and mining. In *Proceedings of the International Conference on Tools with Artificial Intelligence*, ICTAI '07, Washington, DC, USA, 2007. IEEE Computer Society.

[116] Yuming Zhou and Hareton Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on Software Engineering*, 32(10):771–789, 2006.

[117] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for Eclipse. In *Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 9–, Washington, DC, USA, 2007. IEEE Computer Society.

Appendix A

Publications

Journal papers

1. Abebe, S. L., Alicante, A., Corazza, A., and Tonella, P., *Supporting Concept Location through Identifier Parsing and Ontology Extraction.* Journal of Systems and Software. (Under review)

2. Abebe, S. L., and Tonella, P., *Extraction of Domain concepts from the Source Code.* Journal of Science of Computer Programming. (Under submission)

Conference papers

3. Abebe, S. L. and Tonella, P., *Automated Identifier Completion and Replacement*, in Proceedings of the 17th European Conference on Software Maintenance and Reengineering, Genova, Italy, 2013.

4. Abebe, S. L., Arnaoudova V., Tonella P., Antoniol, G., and Guhneuc, Y. G., *Can Lexicon Bad Smells Improve Fault Prediction?*, in Proceedings of the 19th Working Conference on Reverse Engineering, Kingston, Canada, 2012.

5. Abebe, S. L. and Tonella, P., *Towards the Extraction of Domain Concepts from the Identifiers*, in Proceedings of the 18th Working Confer-

ence on Reverse Engineering, Limerick, Ireland, 2011.

6. Abebe, S. L., Haiduc, S., Tonella, P., and Marcus, A., *The Effect of Lexicon Bad Smells on Concept Location in Source Code*, in Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation, Williamsburg, Virginia, USA, 2011.

7. Abebe, S. L. and Tonella, P., *Natural Language Parsing of Program Element Names for Concept Extraction*, in Proceedings of the International Conference on Program Comprehension, Braga, Minho, Portugal, 2010.

8. Abebe, S. L., Haiduc, S., Tonella, P., and Marcus, A., *Lexicon Bad Smells in Software*, in Proceedings of the 16th Working Conference on Reverse Engineering, Lille, France, 2009.

9. Abebe, S. L., Haiduc, S., Marcus, A., Tonella P. and Antoniol, G., *Analyzing the Evolution of the Source Code Vocabulary*, in Proceedings of the 13th European Conference on Software Maintenance and Reengineering, Kaiserslautern, Germany, 2009.

10. Tonella, P. and Abebe, S. L., *Code Quality from the Programmer's Perspective*, in Proceedings of Science, XII Advanced Computing and Analysis Techniques in Physics Research, Erice, Italy, 2008.