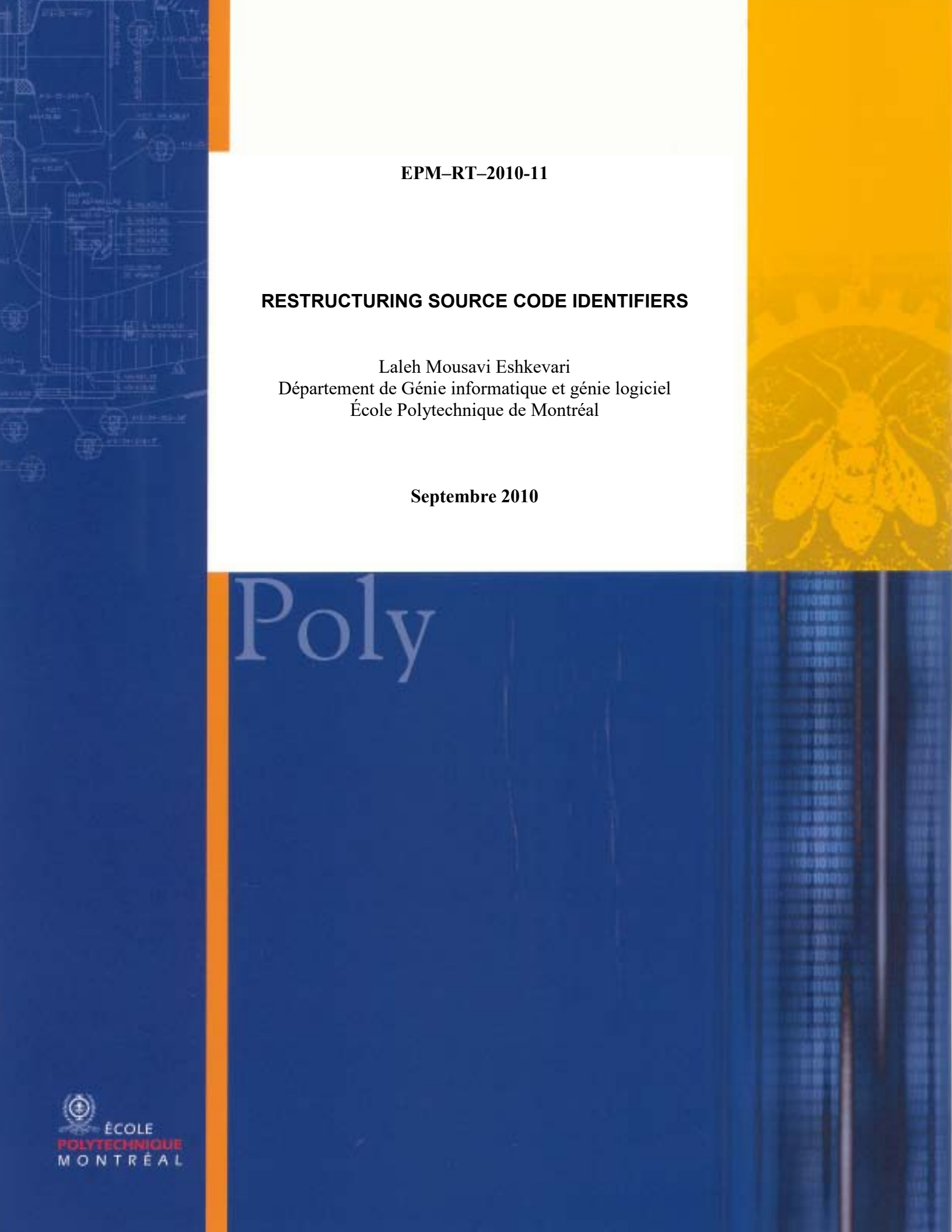
Titre: Title:	Restructuring Source Code Identifiers
Auteurs: Authors:	Laleh Mousavi Eshkevari
Date:	2010
Type:	Rapport / Report
Référence: Citation:	Eshkevari, Laleh Mousavi (2010). Restructuring Source Code Identifiers. Rapport technique. EPM-RT-2010-11.

EPM–RT–2010-11

RESTRUCTURING SOURCE CODE IDENTIFIERS

Laleh Mousavi Eshkevari
Département de Génie informatique et génie logiciel
École Polytechnique de Montréal

Septembre 2010



EPM-RT-2010-11

Restructuring Source Code Identifiers

Laleh Mousavi Eshkevari
Département de génie informatique et génie logique
École Polytechnique de Montréal

Septembre 2010

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

EPM-RT-2010-11
Restructuring Source Code Identifiers
par : Laleh Mousavi Eshkevari
Département de génie informatique et génie logiciel
École Polytechnique de Montréal

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Restructuring Source Code Identifiers

by

Laleh Mousavi Eshkevari

A proposal submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the

Département de génie informatique et génie logiciel

September 2010

Abstract

In software engineering, maintenance cost 60% of overall project lifecycle costs of any software product. Program comprehension is a substantial part of maintenance and evolution cost and, thus, any advancement in maintenance, evolution, and program understanding will potentially greatly reduce the total cost of ownership of any software products. Identifiers are an important source of information during program understanding and maintenance. Programmers often use identifiers to build their mental models of the software artifacts. Thus, poorly-chosen identifiers have been reported in the literature as misleading and increasing the program comprehension effort. Identifiers are composed of terms, which can be dictionary words, acronyms, contractions, or simple strings. We conjecture that the use of identical terms in different contexts may increase the risk of faults, and hence maintenance effort. We investigate our conjecture using a measure combining term entropy and term context-coverage to study whether certain terms increase the odds ratios of methods to be fault-prone. We compute term entropy and context-coverage of terms extracted from identifiers in Rhino 1.4R3 and ArgoUML 0.16. We show statistically that methods containing terms with high entropy and context-coverage are more fault-prone than others, and that the new measure is only partially correlated with size. We will build on this study, and will apply summarization technique for extracting linguistic information form methods and classes. Using this information, we will extract domain concepts from source code, and propose linguistic based refactoring.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software maintenance is defined as the modification of a software product after delivery to correct faults, improve performance (or other attributes) or to adapt the product to a modified environment (ANSI/IEEE standard 1219-1998). The objective of maintenance is to enhance and optimize an existing system while preserving its integrity [1]. The initial step of software maintenance is program comprehension and it demands effort and time, initially to understand the software system and then to identify possible problems to be fixed while providing a remedy for them without affecting the overall behavior of the system. Thus any advancement in maintenance, evolution, and program understanding will potentially greatly reduce the total cost of ownership of any software products.

Many studies in the domain of software engineering used object-oriented metrics, such as Chidamber and Kemerer (CK) metrics suite [2], to evaluate software quality [3, 4], maintenance efforts [5–8], to predict faults [9–11] and change proneness [12]. These metrics are calculated based on the structural data extracted from source code.

We believe that linguistic data extracted from source code might also help to improve code quality. Indeed, several studies showed the impact of identifiers on program comprehension (*e.g.*, [13–15]) and code quality [16].

Most of the refactoring proposed in the literature aim to improve code quality via code restructuring. To the best of our knowledge, the only work on linguistic refactoring is by Caprile *et al.* [13]. They proposed refactoring based on compliance of terms in identifiers with standard lexicon and grammars. The authors analyze method names in isolation, i.e, without considering the identifiers used in the body of the method.

We build on this work and use the linguistic information extracted from method body for refactoring. To do so, we apply summarization techniques. In natural languages,

summarization is used to identify the main topics of a given document while minimizing the redundancy. We apply this technique on method body to extract important topics and use them to propose refactoring for identifiers in method names and–or method bodies. For example, by comparing method name with the topics extracted we can verify if the method name is appropriate or it should be refactored in to reflect those topics. Therefore, our objective is to identify refactoring opportunities, that is, places where linguistic refactoring can be performed. Moreover, we plan to apply such refactorings on code and investigate the impact of refactoring on program comprehension. We believe that linguistic refactoring will enhance program comprehension.

The remaining part of this proposal is organized as follows: In Chapter 2 we discuss the problem and motivation behind this research and the research questions. Chapter 3 describes the proposed methods to address the research questions. Chapter 4 presents the details of our method for addressing first research question. Chapter 5 presents an example where summarization technique is applied on a method body. Chapter 6 defines current and future activities with respect to this research. In Chapter 7 we discuss related work, and finally Chapter 8 provides conclusion and future work.

Chapter 2

Motivation and Problem Statement: Identifiers and Refactoring

Identifiers are among the most important sources of information to understand source code [13, 14]. Haiduc and Marcus [15] studied several open-source programs and found that about 40% of the domain terms were used in the source code. Unfortunately, in collaboration environments the probability of having two developers use the same identifier for different entities is between 7% and 18% [17]. Thus, naming conventions are crucial for improving the source code comprehensibility. Furthermore, applying meaningful and consistent names for source code identifiers can improve the precision of information retrieval (IR) techniques in traceability links recovery between artifacts of different types. Finally, software systems with well-defined identifiers are easier to debug as the program is easer to read and understand.

Identifiers must be sufficiently distinctive yet must relate to one another and to the context in which they appear [14]. We concur with Deißenböck and Pizka's observation that proper identifiers improve quality and that identifiers should be used consistently [14].

We consider an identifier as an anomaly if it has frequency higher than a certain defined threshold, it is used in different context, or it contains terms that are synonym or polysemy. The reason is that we believe such identifiers increase the developers effort to understand the role of the associated identifier, which ultimately may lead to faults. For the purpose of this research, the context and the frequency will be taken to account as criteria for considering an identifier as an anomaly. We would like to investigate if

there is a relation between entities containing identifiers with anomalies and their fault proneness and change proneness. Moreover, we would like to automatically identify such identifiers. We believe that summarization techniques will enable us to extract the characteristic and functionality of entities (e.g., methods, attributes). Using the information gained through summarization, we then plan to refactor entities names toward names that better reflect their functionality.

Thus high level research question can be summarized as: ***How to pinpoint identifiers with anomalies and suggest linguistic-based refactoring of such identifiers.***

This research question can be divided into the following research questions:

- RQ1: How to identify identifiers with anomalies? We define metric to identify identifier with anomalies and investigate the relation between fault proneness and identifier anomalies.
- RQ2: How to identify linguistic feature implemented by methods in the system? We analyze whether identifiers in method signature and body can be used to derive the feature implemented by the method.
- RQ3: How to define linguistic refactoring based on feature identified? This research question relates to the previous question. We are interested to see if we can suggest refactoring strategies based on the feature identified for methods.
- RQ4: Do the proposed linguistic refactoring strategies improve program comprehension and visual effort? We are interested to evaluate whether the there is a significant difference in comprehensibility and visual effort between the original and refactored code

Chapter 3

Methodology

We plan to answer the research questions defined in the Chapter 2 in the following steps:

3.1 RQ1: Identifiers with Anomalies

We present a novel measure based on linguistic data and an empirical study to verify our if we can pinpoint identifiers with anomalies. The novel measure quantifies terms from two aspects: *term entropy* and *context-coverage*. Term entropy is derived from entropy in information theory and measures the "physical" dispersion of a term in a program, *i.e.*, the higher the entropy, the more scattered the term is across entities. Term context-coverage is based on an Information Retrieval method and measures the "conceptual" dispersion of the entities in which the term appears, *i.e.*, the higher the context coverage of a term, the more unrelated are the entities containing it.

We perform an empirical study relating terms with high entropy and high context-coverage to the fault-proneness of the methods and attributes in which they appear. We analyze two widely studied open source programs, ArgoUML[1] and Rhino[2] because sets of manually-validated faults for these two programs exist in the literature. We show that there is a statistically significant relation between the "physical" and "conceptual" dispersion of terms and fault proneness.

[1] http://argouml.tigris.org/
[2] http://www.mozilla.org/rhino/

3.2 RQ2: Linguistic-based Feature Identification

We draw inspiration from previous works on text summarization [18, 19] techniques. As explained in [19], the goal of summarization activity is to identify the main topics of a given document while minimizing redundancy. Program features are implemented via methods. Depending on the granularity of a feature, one method or a group of method implement it. We believe that by analyzing the vocabulary of the identifiers and comments used in a method body, we can extract the feature that the method implements. Summarization techniques enable us to identify important topics in method body, and those topics enable us to infer feature implemented in the method.

Natural language text summarization is consist of two steps: (1) decomposing a text into a term by sentence matrix and (2) applying an appropriate technique (e.g., IR or LSA techniques) for selecting candidate sentences. In our case, each method represents a document, while split identifiers of the method (*e.g.*, names of variables, method calls, user defined types, formal and actual parameters, comments) correspond to terms. Statements, method signature and the comments will correspond to sentences. Therefore, a method can be transformed into term by sentence matrix, where each element $a_{i,j}$ in the matrix is the weighted frequency of term i in sentence j. We would like to distinguish the terms according to the role that the corresponding identifier plays in the method. That is, to give more weights to terms coming from identifiers that are part of formal parameters and return statement.

By applying LSA technique on term by sentence matrix we can identify important topics. We plan to combine techniques used in [13, 20] (explained in detail in Chapter 7) to infer the feature implemented in method body by analyzing those identified topics. We proceed by labeling each method with the identified feature. Moreover, we plan to evaluate the precision of our topic extraction technique by conducting an empirical study on a system that has proved to have good internal quality, and consistent identifier naming (*e.g.*, JHotDraw[3]), and ask experts to validate the results of our technique. The objective would be to verify if topics identified by our techniques match the ones identified by experts.

3.3 RQ3: Linguistic-based Refactoring

Once methods are labeled with feature, we would like to provide recommendations for refactorings. We conjecture that identifiers (if selected wisely) should reflect the responsibility and characteristic of the entity that they are labeling. Moreover, it is a

[3] http://www.jhotdraw.org/

general rule that a method name should reflect its responsibility. The topics and the feature identified in the previous step will enable us to evaluate the appropriateness of the method name and to provide suggestions for better naming in case the original name is not well suited. The same approach can be used for refactoring class names. Moreover, we can also identify possibility for structural refactoring. For example, by analyzing the topics extracted from a method body, we can evaluate the degree to which these topics are related. In other words, we can evaluate the cohesion of a method and suggest extract method as a possible solution to increase the cohesion.

3.4 RQ4: Evaluation of Program Comprehension and Visual Effort

Finally, we would like to evaluate if the proposed refactoring strategies indeed increase code comprehension. To verify if the comprehensibility is improved, we plan to perform an experiment and ask experts to evaluate the degree of comprehensibility before and after refactoring. Subjects will be given two fragments of code (before and after refactoring) and will be asked to identify the purpose of the code only by reading the code. Moreover, we are interested in evaluating the effort of comprehension for code before and after refactoring. We will apply eye tracking techniques to evaluate visual effort. Visual effort will be computed as the amount and duration of eye movements (e.g., eye fixation) needed to read a given piece of a code and to be able to identify the purpose of the code. The goal of this study is to see if there is a significant difference in visual effort between the original and the refactored code. We expect to have less visual effort in refactored code than in the original code.

The expected contributions of this work can be summarized as following:

- A novel measure characterizing the "physical" and "conceptual" dispersions of terms.
- An empirical study showing the relation between the proposed measure and entities fault proneness.
- Extracting domain concepts from source code, which can improve establishing traceability links between requirements and implementation.
- Suggesting refactoring for method names toward semantic information that is implicit in method bodies.
- Evaluating the impact of this refactoring on program comprehension.

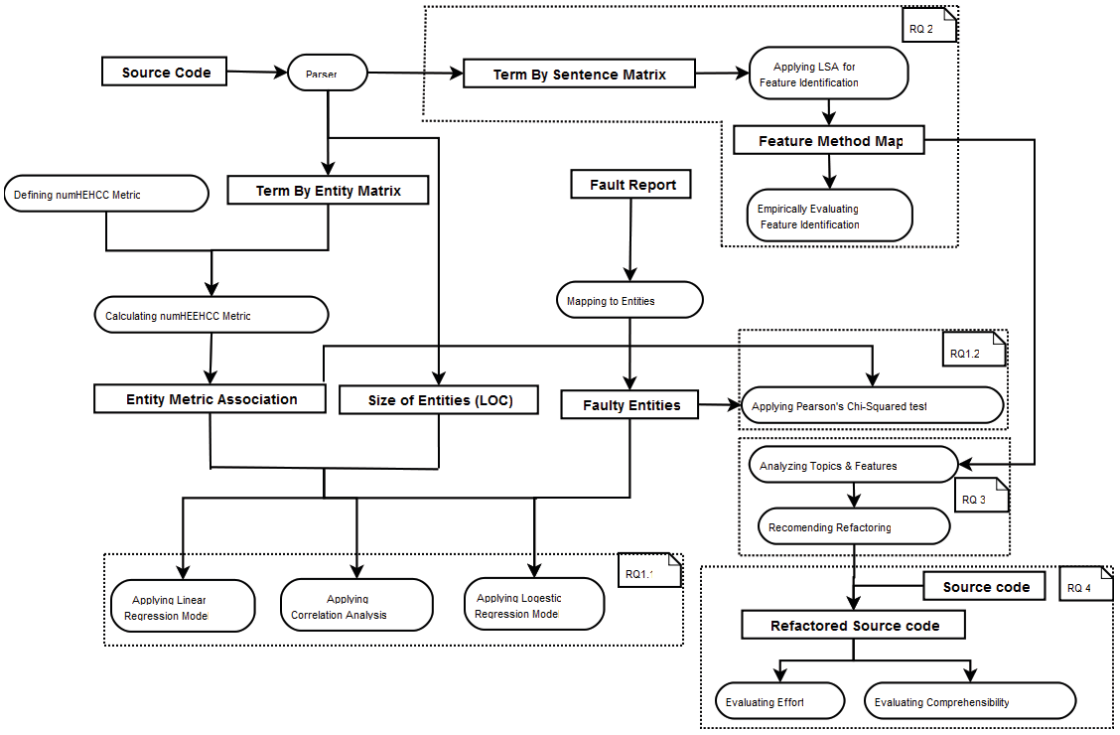


FIGURE 3.1: UML activity diagram illustrating our methodology.

Figure 3.1 illustrate our methodology that we will now detail in next chapters.

Chapter 4

RQ1: Identifiers with Anomalies

This section describes the details our method for answering the *RQ*1. It is a preliminary study and it is accepted for the next International Conference on Software Maintenance-ERA Track [21]. First, the necessary background will be discussed, then the details of the study will be explained.

4.1 Background

With no loss of generality, we focus on methods and attributes because they are "small" contexts of identifiers. Moreover, we consider attributes because they are often part of some program faults, *e.g.*, in Rhino they participate to 37% of the reported faults. However, the computation can be broaden by using classes or other entities as contexts for identifiers.

4.1.1 Data Extraction

We extract the data required to compute term entropy and context-coverage in two steps. First, we extract the identifiers found in class attributes and methods, *e.g.*, names of variables and of called methods, user defined types, method parameters. Extracted identifiers are split using a Camel-case splitter to build the term dictionary, *e.g.*, *getText* is split into *get* and *text*. Future work includes using a more versatile algorithm to extract terms from identifiers, such as in Madani *et al.* [22]. We then apply two filters on the dictionary. We remove terms with a length less than one because their semantics is often unclear and because they most likely correspond to loop indexes (*e.g.*, I, j, k). Next, we prune terms appearing in a standard English stop-word list augmented with programming language keywords.

Second, the linguistic data is summarized into a $m \times n$ frequency matrix, *i.e.*, a *term-by-entity* matrix. The number of rows of the matrix, m, is the number of terms in the dictionary. The number of columns, n, corresponds to the number of methods and attributes. The generic entry $a_{i,j}$ of the term-by-entity matrix denotes the number of occurrences of the i^{th} term in the j^{th} entity.

4.1.2 Term Entropy

Let us suppose a source of information that can emit four symbols, A, B, C, or D. We may have no prior knowledge of symbol frequency and, thus, need to observe emitted symbols to increase our knowledge on the distribution of the symbols. Once we observed a symbol, we gained some information, our uncertainty decreased and, thus, we increased our knowledge about the distribution of the symbols.

Shannon [23] measures the amount of uncertainty, or entropy, of a discrete random variable X as:

$$H(X) = -\sum_{x \in \varkappa} p(x) \cdot log(p(x))$$

where $p(x)$ is the mass probability distribution of the discrete random variable X and $\varkappa$ is its domain.

We consider terms as random variables with some associated probability distributions. We normalize each row of the term-by-entity matrix so that each entry is in $[0, 1]$ and the sum of the entries in a row is equals to one to obtain a probability distribution for each term. Normalization is achieved by dividing each $a_{i,j}$ entry by the sum of all $a_{i,j}$ over the row i. A normalized entry $\widehat{a}_{i,j}$ is then the probability of the presence of the term t_i in the j^{th} entity. We then compute term entropy as:

$$H(t_i) = -\sum_{j=1}^{n} (\widehat{a}_{i,j}) \cdot log(\widehat{a}_{i,j}) \quad i = 1, 2, \ldots, m$$

With term entropy, the more scattered among entities a term is, the closer to the uniform distribution is its mass probability and, thus, the higher is its entropy. On the contrary, if a term has a high probability to appear in few entities, then its entropy value will be low.

4.1.3 Term Context Coverage

While term entropy characterizes the "physical" distribution of a term across entities, context-coverage measures its "conceptual" distribution in the entities in which the term

appears. In particular, we want to quantify whether a same term is used in different contexts, *i.e.*, methods and–or attributes, with low textual similarity. Thus, the context coverage of term t_k (where $k = 1, 2, \ldots, m$) is computed as the average textual similarity of entities containing t_k:

$$CC(t_k) = 1 - \frac{1}{\binom{|C|}{2}} \sum_{\substack{i=1\ldots|C|-1 \\ j=i+1\ldots|C| \\ e_i, e_j \in C}} sim(e_i, e_j)$$

where $C = \{e_l | \widetilde{a}_{k,p} \neq 0\}$ is the set of all entities in which term t_k occurs and $sim(e_i, e_j)$ represents the textual similarity between entities e_i and e_j. The number of summations is $\binom{|C|}{2}$ because $sim(e_i, e_j) = sim(e_j, e_i)$. A low value of the context coverage of a term means a high similarity between the entities in which the term appears, *i.e.*, the term is used in consistent contexts.

To compute the textual similarity between entities, we exploit LSI, a space reduction-based method widely and successfully used in IR [24]. In particular, LSI applies a factor analysis technique to estimate the "latent" structure in word usage trying to overcome the main deficiencies of IR methods, such as synonym and polysemy problems. In particular, the non-normalized term-by-entity LSI projection into the entities subspace $\widetilde{a}_{i,j}$ captures the more important relations between terms and entities. The columns of the reduced term-by-entity matrix represent entities and can be thought of as elements of a vector space. Thus, the similarity between two entities can be measured by the cosine of the angle between the corresponding vectors.

4.1.4 Aggregated Metric

In our current work, we use the variable *numHEHCC* ("number of high entropy and high context coverage"), associated with all entities, to compute correlation, build linear as well as logistic models and contingency tables throughout the following case study:

$$numHEHCC(E_j) = \sum_{i=1}^{m} a_{ij} \cdot \psi(H(t_i) \geq th_H \wedge CC(t_i) \geq th_{CC})$$

where a_{ij} is the frequency in the term-by-entity matrix of term t_i and entity E_j ($j = 1, 2, \ldots, n$) and $\psi()$ is a function returning one if the passed Boolean value is true and zero otherwise.

Thus, $numHEHCC$ represents the overall number of times any term with high entropy (value above th_H) and high context coverage (value above th_{CC}) is found inside an entity.

4.2 Case Study

We performed a first case study of the term entropy and context-coverage measures following the Goal-Question-Metrics paradigm [25]. The *goal* of the study is to investigate the relation (if any) between term entropy and context-coverage, on the one hand, and entities fault proneness, on the other hand. The *quality focus* is a better understanding of characteristics likely to hinder program comprehension and to increase the risk of introducing faults during maintenance. The *perspective* is both of researchers and practitioners who use metrics to study the characteristic of fault prone entities.

The *context* of the study is two open-source programs: Rhino, a JavaScript/ECMAScript interpreter and compiler part of the Mozilla project, and ArgoUML, a UML modeling CASE tool with reverse-engineering and code-generation capabilities. We selected ArgoUML and Rhino because (1) several versions of these programs are available, (2) they were previously used in other case studies [26, 27], and (3) for ArgoUML (from version 0.10.1 to version 0.28) and for Rhino (from version 1.4R3 to version 1.6R5), a mapping between faults and entities (attributes and methods) is available [27, 28].

4.2.1 Case Study's Research Questions

Entropy and context coverage likely capture features different from size or other classical object-oriented metrics, such as the CK metrics suite [2]. However, it is well-known that size is one of the best fault predictors [29–31] and, thus, we first verify that $numHEHCC$ is somehow at least partially complementary to size.

Second, we believe that developers are interested in understanding why an entity may be more difficult to change than another. For example, given two methods using different terms, all their other characteristics being equal, they are interested to identify which of the two is more likely to take part in faults if changed.

Therefore, the case study is designed to answer the following research questions:

- **$RQ_{1.1}$ – Metric Relevance:** Do term entropy and context-coverage capture characteristics different from size and help to explain entities fault proneness? This

question investigates if term entropy and context-coverage are somehow complementary to size, and thus, quantify entities differently.

- **$\mathbf{RQ}_{1.2}$ – Relation to Faults:** Do term entropy and context-coverage help to explain the presence of faults in an entity? This question investigates if entities using terms with high entropy and context-coverage are more likely to be fault prone.

Fault proneness is a complex phenomenon impossible to capture and model with a single characteristic. Faults can be related to size, complexity, and–or linguistic ambiguity of identifiers and comments. Some faults may be better explained by complexity while other by size or linguistic inconsistency of poorly selected identifiers. Therefore, we do not expect that $\mathbf{RQ}_{1.1}$ and $\mathbf{RQ}_{1.2}$ will have the same answer in all version of the two programs and will be universally true. Nevertheless, as previous authors [13, 14, 16, 32], we believe reasonable to assume that identifiers whose terms have with high entropy and high context-coverage hint at poor choices of names and, thus, at a higher risk of faults.

4.2.2 Analysis Method

To statistically analyze $\mathbf{RQ}_{1.1}$, we computed the correlation between the size measured in LOCs and a new metric derived from entropy and context-coverage. Then, we estimated the linear regression models between LOCs and the new metric. Finally, as an alternative to the Analysis Of Variance (ANOVA) [33] for dichotomous variables, we built logistic regression models between fault proneness (explained variable) and LOCs and the proposed new metric (explanatory variables).

Our goal with $\mathbf{RQ}_{1.1}$ is to verify whether term entropy and context-coverage capture some aspects of the entities at least partially different from size. Thus, we formulate the null hypothesis:

> *H_{0_1}: The number of terms with high entropy and context-coverage in an entity does not capture a dimension different from size and is not useful to explain its fault proneness.*

We expect that some correlation with size does exist: longer entities may contain more terms with more chance to have high entropy and high context-coverage.

Then, we built a linear regression model to further analyze the strength of the relation in term of unexplained variance, *i.e.*, $1 - R^2$. This model indirectly helps to verify that entropy and context-coverage contribute to explain fault proneness in addition to size.

Finally, we performed a deeper analysis via logistic regression models. We are not interested in predicting faulty entities but in verifying if entropy and context-coverage help to explain fault proneness. The multivariate logistic regression model is based on the formula:

$$\pi(X_1, X_2, \ldots, X_n) = \frac{e^{C_0 + C_1 \cdot X_1 + \cdots + C_n \cdot X_n}}{1 + e^{C_0 + C_1 \cdot X_1 + \cdots + C_n \cdot X_n}}$$

where X_i are the characteristics describing the entities and $0 \leq \pi \leq 1$ is a value on the logistic regression curve. In a logistic regression model, the dependent variable π is commonly a dichotomous variable, and thus, assumes only two values $\{0, 1\}$, *i.e.*, it states whether an entity took part in a fault (1) or not (0). The closer $\pi(X_1, X_2, \ldots, X_n)$ is to 1, the higher is the probability that the entity took part in a fault. An independent variable X_i models information used to explain the fault proneness probability; in this study we use a metric derived from term entropy and the context-coverage, $numHEHCC$, and a measure of size (LOCs) as independent variables.

Once independent variables are selected, given a training corpus, the model estimation procedure assigns an estimated value and a significance level, p-value, to the coefficients C_i. Each C_i p-value provides an assessment of whether or not the i^{th} variable helps to explain the independent variable: fault proneness of entities.

Consequently, we expect that the logistic regression estimation process would assign a statistically relevant p-value to the coefficient of a metric derived from term entropy and context coverage, *i.e.*, lower than 0.05 corresponding to a 95% significance level.

With respect to our second research question ($\mathbf{RQ}_{1.2}$) we formulate the following null hypothesis:

> H_{0_2}*: There is no relation between high term entropy and context coverage of an entity and its fault proneness.*

We use a prop-test (Pearson's chi-squared test) [33] to test the null hypothesis. If term entropy and context coverage are important to explain fault proneness, then the prop-test should reject the null hypothesis with a statistically significant p-value.

To quantify the effect size of the difference between entities with and without high values of term entropy and context coverage, we also compute the *odds ratio* (OR) [33] indicating the likelihood of the entities to have such high values for our metric. OR is defined as the ratio of the odds p of a fault prone entity to have high term entropy and high context coverage to the odds q of this entity to have low entropy and context coverage: $OR = \frac{p/(1-p)}{q/(1-q)}$. When $OR = 1$ the fault prone entities can either have high or low term entropy and context coverage. Otherwise, if $OR > 1$ the fault prone entities

have high term entropy and high context coverage. Thus, we expect $OR > 1$ and a statistically significant p-value (*i.e.*, again 95% significance level).

4.2.3 Execution

We download several versions of Rhino for which faults were documented by Eaddy *et al.* [27] from the Mozilla Web site[1]. Versions of ArgoUML were downloaded from the Tigris Community Web site[2]. We selected the version of ArgoUML that has the maximum number of faulty entities (ArgoUML v0.16.) and one of the versions of Rhino, (Rhino v1.4R3).

The selected version of ArgoUML consists of 97,946 lines of Java code (excluding comments and blank lines outside methods and classes), 1,124 Java files, and 12,423 methods and fields. Version 1.4R3 of Rhino consists of 18,163 lines of Java code (excluding comments and blank lines outside methods and classes), 75 files, 1,624 methods and fields.

To create the term-by-entity matrix, we first parse the Java files of Rhino and ArgoUML to extract identifiers. We obtain terms by splitting the identifiers using a Camel-case split algorithm. We compute term entropy and context coverage using the approach presented in the previous section. We finally use existing fault mappings [27, 28] to tag methods and attributes and relate them with entropy and context coverage values. The following paragraphs detail each step.

4.2.3.1 Parsing

We used Java grammar 1.5 and JavaCC[3] to generate a Java parser that extracts the identifiers. To verify the grammar, we have parsed 11 versions of Rhino and 11 versions of ArgoUML with our parser, Table 4.1 shows the number of files which were not parsed using our parser for each version. Other versions of Rhino (1.5R5, 1.6R1, 1.6R2, 1.6R3, 1.6R4, 1.6R5) and ArgoUML (0.18.1, 0.20, 0.22, 0.24, 0.26, 0.26.2, 0.28) with 636 and 11,062 number of files respectively, all files were parsed.

For our case study we excluded the file which was not parsable. Since the percentage of files not been parsed using our parser for each version was less than 0.08 percent we have decided to proceed with our parser instead of using existing parsers.

[1] https://developer.mozilla.org/
[2] http://argouml.tigris.org/
[3] https://javacc.dev.java.net/

System	Total number of Java files	Number of files not parsed	Percentage
Rhino1.4R3	75	1	0.01
Rhino1.5R1	100	3	0.03
Rhino1.5R2	105	2	0.02
Rhino1.5R3	104	2	0.02
Rhino1.5R4.1	107	1	0.01
ArgoUML0.10.1	777	64	0.08
ArgoUML0.12	850	64	0.08
ArgoUML0.14	1077	57	0.05
ArgoUML0.16	1124	53	0.04

TABLE 4.1

4.2.3.2 Identifier Splitting

Identifier splitting is done in three steps : First, the identifiers are split on digits and special characters. Second, they are further split on lowercase to uppercase. Third, they are split on uppercase to lowercase (before the last uppercase letter). After splitting the identifiers to terms, we have applied two filters: first we have omitted the terms which have the length equal or less than two, then the terms are further filtered through stop words. The stop word list is a standard list to which we added Java specific terms and keywords.

4.2.3.3 Mapping Faults to Entities

We reuse previous findings to map faults and entities. For Rhino the mapping of faults with entities was done by Eaddy *et al.* [27] for 11 versions of Rhino. We obtain the mapping which corresponds to Rhino v1.4R3 by extracting, for each fault, its reporting date/time[4] and its fixing date/time. Then, we keep only those faults which fall under one of the following two cases: (i) the reporting date of the fault was before the release date of v1.4R3 and its fixing date was after the release date of the same version, (ii) the reporting date of the fault is after the release date of v1.4R3 and before the release date of the next version (v1.5R1). As for ArgoUML, we also use a previous mapping between faults and classes [28]. For each class marked as faulty, we compare its attributes and methods with the attributes and methods of the same class in the successive version and keep those that were changed and mark them as faulty.

4.2.3.4 Mapping Entities to Entropy and Context Coverage

We identify entities with *high* term entropy and context coverage values by computing and inspecting the box-plots and quartiles statistics of the values on all Rhino versions

[4]https://bugzilla.mozilla.org/query.cgi

TABLE 4.2: Correlation test for ArgoUML v0.16 and Rhino v1.4R3.

System	Correlation	p-values
ArgoUML	0.4080593	$\prec 2.2e-16$
Rhino	0.4348286	$\prec 2.2e-16$

and the first five versions of ArgoUML. The term context coverage distribution is skewed towards high values. For this reason we use 10% highest values of term context coverage to define a threshold identifying the high context coverage property. In other words, standard outlier definition was not applicable to context coverage. We do not observe a similar skew for the values of term entropy and, thus, the threshold for high entropy values is based on the standard outlier definition (1.5 times the inter-quartile range above the 75% percentile). We use the two thresholds to measure for each entity, the number of terms characterized by high entropy and high context coverage that it contains.

4.2.4 Results

We now discuss the results achieved aiming at providing answers to our research questions.

4.2.4.1 $RQ_{1.1}$ – Metric Relevance

Table 4.2 reports the results of Pearson's product-moment correlation for both Rhino and ArgoUML. As expected, some correlation exists between LOC and $numHEHCC$ plus the correlation is of the same order of magnitude for both programs.

Despite a 40% correlation a linear regression model built between $numHEHCC$ (dependent variable) and LOC (independent variable) attains an R^2 lower than 19% (see Table 4.3). The R^2 coefficient can be interpreted as the percentage of variance of the data explained by the model and thus $1 - R^2$ is an approximations of the model unexplained variance. In essence Table 4.3 support the conjecture that LOC does not substantially explain $numHEHCC$ as there is about 80% (85%) of Rhino (ArgoUML) $numHEHCC$ variance not explained by LOC. Correlation and linear regression models can be considered a kind of sanity check to verify that LOC and $numHEHCC$ help to explain different dimensions of fault proneness.

The relevance of $numHEHCC$ in explaining faults, on the programs under analysis, is further supported by logistic regression models. Table 4.4 reports the interaction model built between fault proneness (explained variable) and the explanatory variables LOC and $numHEHCC$. In both models, $M_{ArgoUML}$ and M_{Rhino}, the intercept is relevant as

TABLE 4.3: Linear regression models for Rhino v1.4R3 and ArgoUML v0.16.

	Variables	Coefficients	p-values
Rhino ($R^2 = 0.1891$)	Intercept	0.038647	0.439
	LOC	0.022976	$\prec 2e-16$
Argo (R^2=0.1665)	Intercept	-0.0432638	0.0153
	LOC	0.0452895	$\prec 2e-16$

TABLE 4.4: ArgoUML v0.16 and Rhino v1.4R3 logistic regression models.

	Variables	Coefficients	p-values
$M_{ArgoUML}$	Intercept	-1.688e+00	$\prec 2e-16$
	LOC	7.703e-03	$8.34e-10$
	numHEHCC	7.490e-02	$1.42e-05$
	LOC:numHEHCC	-2.819e-04	0.000211
M_{Rhino}	Intercept	-4.9625130	$\prec 2e-16$
	LOC	0.0041486	0.17100
	numHEHCC	0.2446853	0.00310
	LOC:numHEHCC	-0.0004976	0.29788

well as $numHEHCC$. Most noticeably in Rhino, the LOC coefficient is not statistically significant as well as the interaction term ($LOC : numHEHCC$). This is probably a fact limited to Rhino version 1.4R3 as for ArgoUML both LOC and the interaction term are statistically significant. In both models $M_{ArgoUML}$ and M_{Rhino} the LOC coefficient is, at least, one order of magnitude smaller than the $numHEHCC$ coefficient, which can partially be explained by the different range of LOC versus $numHEHCC$. On average in both programs method size is below 100 LOC and most often a method contains one or two terms with high entropy and context coverage. Thus, at first glance we can safely say that both LOC and $numHEHCC$ have the same impact in term of probability. In other words, the models in Table 4.4 clearly show that LOC and $numHEHCC$ capture different aspects of the fault proneness characteristic. Base on the reported results we can conclude that although some correlation exists between LOC and $numHEHCC$, statistical evidence allows us to reject, on the programs under analysis, the null hypothesis H_{0_1}.

4.2.4.2 $\mathbf{RQ}_{1.2}$ – Relation to Faults

To answer $\mathbf{RQ}_{1.2}$, we perform prop-tests (Pearson's chi-squared test) and test the null hypothesis H_{0_2}. Indeed, (i) if prop-tests revel that *numHEHCC* is able to divide the population into two sub-populations and (ii) if the sub-population with positive values for *numHEHCC* has an odds ratio bigger than one, then *numHEHCC* may act as a risk indicator. For entities with positive *numHEHCC* it will be possible to identify those

Table 4.5: ArgoUML v0.16 confusion matrix.

ArgoUML	numHEHCC ≥ 1	numHEHCC $= 0$	Total
Fault prone	381	1706	2087
Fault free	977	9359	10336
Total	1358	11065	12423
p-value $\prec 2.2e-16$			
Odds ratio $= 2.139345$			

Table 4.6: Rhino v1.4R3 confusion matrix.

Rhino	numHEHCC ≥ 1	numHEHCC $= 0$	Total
Fault prone	6	8	14
Fault free	172	1438	1610
Total	178	1446	1624
p-value $= 0.0006561$			
Odds ratio $= 6.270349$			

Table 4.7: ArgoUML v0.16 confusion matrix.

ArgoUML	numHEHCC ≥ 2	numHEHCC $= 1$	Total
Fault prone	198	183	381
Fault free	511	466	977
Total	709	649	1358
p-value $= 0.9598$			
Odds ratio $= 0.9866863$			

Table 4.8: Rhino v1.4R3 confusion matrix.

Rhino	numHEHCC ≥ 2	numHEHCC $= 1$	Total
Fault prone	3	3	6
Fault free	75	97	172
Total	78	100	178
p-value $= 1$			
Odds ratio $= 1.293333$			

terms leading to high entropy and high context coverage, identifying also the contexts and performing refactoring actions to reduce entropy and high context coverage.

Tables 4.5 and 4.6 show the confusion matrices for ArgoUML v0.16 and Rhino v1.4R3, together with the corresponding p-value and odds ratios. As the tables show, the null hypothesis H_{0_2} can be rejected.

We further investigate, with Tables 4.7 and 4.8, the relation between $numHEHCC$ and odds ratio. These contingency tables compute the odds ratio of entities containing two or more terms with high entropy and high context coverage with those entities which only contain one high entropy and high context coverage term. They are not statistically significant, but the odds ratio is close to one, which seems to suggest that the real

TABLE 4.9: Odds change due to LOC ($numHEHCC$=1) and $numHEHCC$(LOC=10) for ArgoUML v0.16 and Rhino v1.4R3.

Changing variable	Δ	Odds change ArgoUML	Odds change Rhino
LOC	1	1.007448705	1.003657673
	10	1.077034036	1.037184676
	50	1.449262781	1.200274163
numHEHCC	1	1.074742395	1.270879652
	2	1.155071215	1.61513509
	10	2.056097976	10.99117854
	50	36.74675785	160406.2598

difference is between not containing high entropy and high context coverage terms and just containing one or more. The results allow us to conclude, on the analyzed programs, that there is a relation between high term entropy and context-coverage of an entity and its fault proneness.

4.2.5 Discussion

We now discuss some design choices we adopted during the execution of the case studies aiming at clarifying their rationale.

4.2.5.1 LSI subspace dimension

The choice of LSI subspace is critical. Unfortunately, there is not any systematic way to identify the optimal subspace dimension. However, it was observed that in the application of LSI to software artifacts repository for recovering traceability links between artifacts good results can be achieved setting $100 \leq k \leq 200$ [34, 35]. Therefore, following such a heuristic approach, we set the LSI subspace dimension equal to 100.

4.2.5.2 Java Parser

We developed our own Java parser, using a Java v1.5 grammar, to extract identifiers and comments from source code. Our parser is robust and fast (less than two minutes to parse any version of the studied programs, in average) but when applied, few files could not be parsed. Unparsed files include those developed on earlier versions of both ArgoUML and Rhino because of the incompatibility between the different versions of Java grammar.

4.2.5.3 Statistical Computations

All statistical computations were performed in R[5]. The computations took about one day for both programs, where the most expensive part of the computation in terms of time and resources was the calculation of the similarity matrix. We believe that neither extensibility nor scalability are issues: this study explains the fault phenomenon and is not meant to be performed on-line during normal maintenance activities. In the course of our evaluation, we realized that the statistical tool R yields different results when used in different software/hardware platforms. We computed the results of our analysis on R on Windows Vista/Intel, Mac OS X (v10.5.8)/Intel, and RedHat/Opteron, and we observed some differences. All results provided in this paper have been computed with R v2.10.1 on an Intel computer running Mac OS. We warn the community of using R and possibly other statistical packages on different platforms because their results may not be comparable.

4.2.5.4 Object-oriented Metrics

We studied the relation between our novel metric, based on term entropy and context coverage, and LOC, which is among the best indicator of fault proneness [29–31] to show that our metric provides different information. We did not study the relation between our metric and other object-oriented metrics. Of particular interest are coupling metrics that could strongly relate to term entropy and context coverage. However, we argue, with the following thought-experiment, that term entropy and context coverage, on the one hand, and coupling metrics, on the other hand, characterize different information. Let us assume the source code of a working software system, with certain coupling values between classes and certain entropy and context coverage values for its terms. We give this source code to a simple obfuscator that mingles identifiers. The source code remains valid and, when compiled, results in a system strictly equivalent to the original system. Hence, the coupling values between classes did not change. Yet, the term entropy and context coverage values most likely changed.

4.2.6 Threats to Validity

This study is a preliminary study aiming at verifying that our novel measure, based on term entropy and context coverage, for two known programs (ArgoUML v0.16 and Rhino 1.4R3), is related to the fault proneness of entities (methods and attributes) and, thus, is useful to identify fault prone entities. Consider Table 4.9; for a fixed $numHEHCC$

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

value (one) an increase of ten for LOC will not substantially change the odds (7.7% for ArgoUML; 3.7% for Rhino[6]) while an increase of 50 increases the odds but not significantly (44.9% for ArgoUML; 20% for Rhino) in comparison to the variation of $numHEHCC$ (for a fixed value of LOC=10). For instance, in the case of ArgoUML for a fixed size of entities, one unit increase of $numHEHCC$ has almost the same odds effect than an increase of 10 LOCs. In the case of Rhino, for a fixed size of entities, one unit increase of $numHEHCC$ has more effect than an increase of 50 LOCs. Table 4.9 suggests that indeed an entity with ten or more terms with high entropy and context coverage dramatically change the odds and, thus, the probability of the entities to be faulty. Intuition as well as reported evidence suggest that term entropy and context coverage are indeed useful.

Threats to *construct validity* concern the relationship between the theory and the observation. These threats in our study are due to the use of possibly incorrect fault classifications and–or incorrect term entropy and context coverage values. We use manually-validated faults that have been used in previous studies [27]. Yet, we cannot claim that all fault prone entities have been correctly tagged or that fault prone entities have not been missed. There is a level of subjectivity in deciding if an issue reports a fault and in assigning this fault to entities. Moreover, in the case of ArgoUML, we used the mapping of faults to classes provided in [28]. In order to map the faults to entities we compared faulty classes with their updated version in the consecutive release, and we marked as faulty those entities that were modified. However, the changes could be due to a maintenance activity other than fault fixing, such as refactoring. Our parser cannot parse some Java files due to the incompatibility between the different versions of Java grammar, but errors are less than 4.7% in the studied program and thus do not impact our results. Another threat to validity could be the use of our parser to compute the size of entities. In the computation we took into account the blank lines and comments inside method bodies. We also used a threshold to identify "dangerous" terms and compute $numHEHCC$. The choice of threshold could influence the results achieved. Nevertheless, analyses performed with other thresholds did not yield different or contrasting results.

Threats to *internal validity* concern any confounding factor that could influence our results. This kind of threats can be due to a possible level of subjectiveness caused by the manual construction of oracles and to the bias introduced by the manual classification of fault prone entities. We attempt to avoid any bias in the building of the oracle by reusing a previous independent classification [27, 28]. Also, we discussed the relation and lack

[6] Although the coefficient for LOC is not significant, it was taken into account for the calculation of odds.

	BUG~LOC*Num			BUG~LOC+Num	cor(LOC,Num)	Num~LOC	prop.test	
	P-val of Num	P-val of LOC:Num	P-val of LOC	P-val of Num		R-squared	P-val	OR
RHINO								
14R3	**0.00310 ****	**0.29788**	**0.17100**	**0.427**	**0.4348286**	**0.1891**	0.0006561	6.2703488372093
15R1	0.4370	0.4948	0.0615	0.820	0.4834306	0.2337	0.4715	1.89127552373376
15R2	0.819	0.683	0.049 *	0.716	0.5671881	0.3217	0.5862	2.49775784753363
15R3	0.15351	0.21787	0.00438 **	0.6078	0.607882	0.3695	0.3078	1.51700024113817
15R41	0.0236 *	0.1150	0.0267 *	0.864	0.6144398	0.3775	0.3539	0.838453601539334
15R5	0.760	0.373	2.24e-05 ***	0.0624	0.59409	0.3529	6.366e-05	2.22070461204808
16R1	0.0225 *	0.1808	7.7e-09 ***	0.00133 **	0.693804	0.4814	9.558e-06	6.23550087873462
16R2	0.460	0.443	8.03e-05 ***	0.118	0.7183954	0.5161	5.117e-07	19.52023988006
16R3	0.106	0.462	0.734	0.543	0.7185065	0.5163	0.8762	3.87020648967552
16R4	0.00143 **	0.10138	0.87921	0.00367 **	0.718443	0.5162	0.09403	3.23936696340257
16R5	0.000130 ***	0.037970 *	0.949355	0.000395 ***	0.718443	0.5162	0.02984	3.89301634472511
ARGO								
10.1	5.13e-06 ***	0.0167 *	1.72e-05 ***	0.00331 **	0.3585247	0.1285	< 2.2e-16	3.77434873842059
12	0.01742 *	0.24144	0.00382 **	0.02525 *	0.3519112	0.1238	0.3846	1.09600293996844
14	0.5714	0.0947	3.65e-05 ***	0.0123 *	0.2833291	0.08028	0.003658	1.33146385542169
16	1.30e-05 ***	0.000210 ***	1.22e-09 ***	0.0201 *	0.4080593	0.1665	< 2.2e-16	2.13598761718464
18	0.00902 **	0.02080 *	< 2e-16 ***	0.503	0.3211758	0.1032	< 2.2e-16	2.17367145721925
20	5.46e-12 ***	0.000145 ***	9.03e-15 ***	2.46e-09 ***	0.3134616	0.09826	< 2.2e-16	1.81275400847108
22	0.5926	0.0581	2.94e-14 ***	0.626	0.1448166	0.02097	0.4695	1.16523298174674
24	< 2e-16 ***	2.94e-14 ***	< 2e-16 ***	<2e-16 ***	0.1510216	0.02281	< 2.2e-16	3.2243140738716
26	0.226	0.293	0.136	0.512	0.1614367	0.02606	0.684	1.47006660323501
26.2	0.000191 ***	4.02e-05 ***	3.39e-08 ***	0.2067	0.161324	0.02603	0.01221	1.34535919396442

FIGURE 4.1: Summary of all results for different versions of ArgoUML and Rhino.

thereof between term entropy and context coverage and other existing object-oriented metrics.

Threats to *external validity* concern the possibility of generalizing our results. The study is limited to two programs, ArgoUML 0.16 and Rhino 1.4R3. Results are encouraging but it pays to be cautious. Preliminary investigation on the ten ArgoUML and eleven Rhino releases show that $numHEHCC$ is complementary to LOC for fault explanation. The results of both ArgoUML and Rhino are summarized in Figure 4.1. Overall, although the approach is applicable to other programs, we do not know whether or not similar results would be obtained on other programs or releases. Finally, although we did not formally investigate the measures following the guidelines of measurement theory [36], we derived them from well-known definitions and relations and we plan to study their formal properties as part of our future work while addressing the threats to external validity.

4.3 Conclusion

We presented a novel measure related to the identifiers used in programs. We introduced term entropy and context-coverage to measure, respectively, how rare and scattered across program entities are terms and how unrelated are the entities containing them. We provide mathematical definitions of these concepts based on terms frequency and combined them in a unique measure. We then studied empirically the measure by relating terms with high entropy and high context-coverage with the fault proneness of the entities using these terms. We used ArgoUML and Rhino as object programs

because previous work provided lists of faults. The empirical study showed that there is a statistically significant relation between attributes and methods whose terms have high entropy and high context-coverage, on the one hand, and their fault proneness, on the other hand. It also showed that, albeit indirectly, the measures of entropy and context coverage are useful to assess the quality of terms and identifiers.

Future work includes empirical studies of the relations between high entropy and context coverage with other evolution phenomena, such change proneness. It also includes using the measures to provide hints to the developers on the best choice of identifiers while programming. We also plan to relate *numHEHCC* and other term entropy and context coverage derived metrics with a larger suite of object-oriented metrics and study interaction between OO metrics and metric proposed in this work.

Chapter 5

RQ2: linguistic feature identification

As explained in Chapter 3 for the RQ2 our goal is to identify the important topics that are implicit in the method body. For identifying such topics we will apply summarization technique. After reading summary of a given text written in natural language, we are able to identify its domain as well as its important topics. However, we are not yet able to infer the title of the text just by reading its summary. As title of a text introduce the domain and the focus of the text, we expect that the title and the body of the summary share a lot of terms. We believe that the same fact should hold when summarizing method body. It is important to note that understanding method summary is more challenging than understanding summaries in natural language as the sentences in the summary are the program statements. But we have to take in to account that these statements contain the mosts important topics of the method. To verify if indeed this is the case, we applied summarization technique on a method body. We choose JHotDrow as it has good internal quality and consistent identifier naming. In this system we select method `findpoint` in class `ShortestDistanceConnector`, as this method is one of the large method in the system. Figure 5.1 illustrates the method body.

We applied the summarization technique proposed by Gong *et al.* [19] in the following steps:

1. Decompose method body into a term by sentence matrix.
2. Apply Singular Value Decomposing (SVD) technique to decompose the term by sentence matrix into three matrix: singular value matrix, right and left singular vector matrix.

```
protected Point findPoint(ConnectionFigure connection, boolean getStart) {
    Figure startFigure = connection.start().owner();
    Figure endFigure = connection.end().owner();
    Rectangle r1 = startFigure.displayBox();
    Rectangle r2 = endFigure.displayBox();
    Insets i1 = startFigure.connectionInsets();
    Insets i2 = endFigure.connectionInsets();
    Point p1, p2;
    Point start = null, end = null, s = null, e = null;
    long len2 = Long.MAX_VALUE, l2;
    int x1, x2, y1, y2; // connection points
    int xmin, xmax, ymin, ymax;
    // X-dimension
    // constrain width connection insets
    int r1x, r1width, r2x, r2width, r1y, r1height, r2y, r2height;
    r1x = r1.x + i1.left;
    r1width = r1.width - i1.left - i1.right-1;
    r2x = r2.x + i2.left;
    r2width = r2.width - i2.left - i2.right-1;
    // find x connection point
    if (r1x + r1width < r2x) {
        x1 = r1x + r1width;
        x2 = r2x;
    } else if (r1x > r2x + r2width) {
        x1 = r1x;
        x2 = r2x + r2width;
    } else {
        xmax = Math.max(r1x, r2x);
        xmin = Math.min(r1x+r1width, r2x+r2width);
        x1 = x2 = (xmax + xmin) /2;
    }
    // Y-Dimension
    // constrain with connection insets
    r1y = r1.y + i1.top;
    r1height = r1.height - i1.top - i1.bottom-1;
    y1 = r1y + r1height;
    y2 = r2y;
} else if (r1y > r2y + r2height) {
    y1 = r1y;
    y2 = r2y + r2height;
} else {
    ymax = Math.max(r1y, r2y);
    ymin = Math.min(r1y+r1height, r2y+r2height);
    y1 = y2 = (ymax + ymin) /2;
}
// find shortest connection
for (int i = 0; i < 4; i++) {
    switch(i) {
    case 0:
        // EAST-WEST
        p1 = Geom.east(r1);
        p2 = Geom.west(r2);
        s = new Point(p1.x, y1);
        e = new Point(p2.x, y2);
        break;
    case 1:
        // WEST-EAST
        p1 = Geom.west(r1);
        p2 = Geom.east(r2);
        s = new Point(p1.x, y1);
        e = new Point(p2.x, y2);
        break;
    case 2:
        // NORTH-SOUTH
        p1 = Geom.north(r1);
        p2 = Geom.south(r2);
        s = new Point(x1, p1.y);
        e = new Point(x2, p2.y);
        break;
    }
    l2 = Geom.length2(s.x, s.y, e.x, e.y);
    if (l2 < len2) {
        start = s;
        end = e;
        len2 = l2;
    }
}
if (getStart)
    return start;
return end;
}
```

FIGURE 5.1: JHotDrow, method findPoint in class ShortestDistanceConnector

3. Set k (number of sentences to be added to the summary) to 1.
4. Select a column vector (sentence) in the right singular matrix whose kth element is the largest.
5. Add this column vector (sentence) to the summary.
6. If k reaches a predefined number stop, otherwise increment k and go to step 4.

We set k to ten and fifteen and the candidates sentences are shown in Figure 5.2. The order of sentences in the figure shows their importance. On the one hand terms such as height, length, top, bottom, geom, and figure suggest that the context is related to geometry. One the other hand, we can infer from terms such as shortest, find, and connection that the context should be related to graph. When analyzing the method manually, we can see that the method return the start or end dimension of a figure based on a condition. The summarized sentences infer a context close to the actual purpose of method, but we need a way to highlight terms such as find, start, and end to correctly

```
k=10
// SOUTH-NORTH
protected Point findPoint(ConnectionFigure connection, boolean getStart)
r2height = r2.height - i2.top - i2.bottom-1;
Insets i1 = startFigure.connectionInsets();
p1 = Geom.north(r1);
Insets i2 = endFigure.connectionInsets();
r1height = r1.height - i1.top - i1.bottom-1;
long len2 = Long.MAX_VALUE, l2;
Rectangle r2 = endFigure.displayBox();
int xmin, xmax, ymin, ymax;
--------------------------------
K=15
// SOUTH-NORTH
protected Point findPoint(ConnectionFigure connection, boolean getStart)
r2height = r2.height - i2.top - i2.bottom-1;
Insets i1 = startFigure.connectionInsets();
p1 = Geom.north(r1);
Insets i2 = endFigure.connectionInsets();
r1height = r1.height - i1.top - i1.bottom-1;
long len2 = Long.MAX_VALUE, l2;
Rectangle r2 = endFigure.displayBox();
int xmin, xmax, ymin, ymax;
x1 = x2 = (xmax + xmin) /2;
// find shortest connection
// WEST-EAST
//xmin = Math.min(r1x+r1width, r2x+r2width);
if (getStart)
```

FIGURE 5.2: Summarization of method findPoint in class ShortestDistanceConnector

identify the purpose of method. It is an on going work, and we plan to apply different techniques such as the ones proposed in [18, 37] to find the technique that works the best for our purpose. Moreover, we plan to apply different weighting methods (e.g. Binary weight, term frequency inverse document frequency (tf-idf) weight) for the terms in the term by sentence matrix.

Chapter 6

Research Plan

In the following a detail plan for completing the study will be proposed.

6.1 RQ1, (Summer 2010 - Fall 2010):

This research question investigates the relation between source code identifiers and fault proneness of entities. We address this research question in two steps:

- Study the relation between terms extracted from source code identifiers and fault proneness.
  Publication: V. Arnaoudova, L. Eshkevari, R. Oliveto, Y.-G. Guéhéneuc, and G. Antoniol, "Physical and Conceptual Identifier Dispersion Measures and Relation to Fault Proneness," in Proceedings of the 26th International Conference on Software Maintenance (ICSM'10) - ERA Track. IEEE Computer Society, 2010.
- Compare physical and conceptual dispersion to other metrics used for fault explanation.
  Possible publication: IEEE Transactions on Software Engineering (TSE).

6.2 RQ2, (Winter 2011 - Summer 2011):

The objective of this research question is to extract topic(s) from method bodies and to infer the features implemented via methods. We address this research question in the two following steps:

- Topics and feature extraction.
  Possible publication: International Conference on Software Maintenance (ICSM) 2011.

- Conduct an empirical study with software developers to evaluate the proposed feature extraction technique.
  Possible publication: Empirical Software Engineering Journal.

6.3 RQ3, (Fall 2011):

This research question deals with defining linguistic refactoring. The following journal will be target for addressing this research question.

- Catalog of linguistic refactoring.
  Possible publication: Journal of Systems and Software

6.4 RQ4, (Winter 2012 - Summer 2012):

This research question evaluates the impact of proposed refactoring (previous research question) on program comprehension and visual effort. The following two steps we will address this research question.

- Conduct an experiment to evaluate visual efforts when reading code before and after refactoring. **Possible publication:** International Conference on Software Maintenance (ICSM) 2012.

- Conduct an experiment with software developers to evaluate program comprehension. Subject will be given two versions of a system (before and after refactoring) and asked explain the purpose of a given code.
  Possible publication: International Conference on Software Engineering (ICSE) 2012.

Chapter 7

Related work

Our study relates to Information Retrieval (IR), fault proneness, and the quality of source code identifiers.

7.1 Entropy and IR-based Metrics

Several metrics based on entropy exist. Olague *et al.* [38] used entropy-based metrics to explain the changes that a class undergoes between versions of an object-oriented program. They showed that classes with high entropy tend to change more than classes with lower entropy. Yu *et al.* [39] combined entropy with component-dependency graphs to measure component cohesion. Entropy was also used by Snider [40] to measure the structural quality of C code by comparing the entropy of legacy program with that of a rewrite of the same program aimed at producing a well-structured system. The rewritten program had a much lower entropy that the legacy program.

IR methods have also been used to define new measures of source code quality. In [41] Etzkorn *et al.* presented a new measure for object-oriented programs that examines the implementation domain content of a class to measure its complexity. The content of methods in a class has also been exploited to measure the conceptual cohesion of a class [42–44]. In particular, IR methods were used to compute the overlap of semantic information in implementations of methods, calculating the similarities among the methods of a class. Applying a similar LSI-based approach, Poshyvanyk and Marcus [45] defined new coupling metrics based on semantic similarity. Binkley *et al.* [32] also used VSM to analyze the quality of programs. Split identifiers extracted from entities were compared against the split identifiers extracted from the comments of the entities: the higher the similarity, the higher the quality of the entities. The metric was also

applied to predict faults and a case study showed that the metric is suitable for fault prediction in programs obeying code conventions.

7.2 Metrics and Fault Proneness

Several researchers studied the correlations between static object-oriented metrics, such as the CK metrics suite [2], and fault proneness. For example, Gyimóthy *et al.* [29] compared the accuracy of different metrics from CK suite to predict fault-prone classes in Mozilla. They concluded that CBO is the most relevant predictor and that LOC is also a good predictor. Zimmermann *et al.* [46] conducted a case study on Eclipse showing that a combination of complexity metrics can predict faults, suggesting that the more complex the code is, the more faults in it. El Emam *et al.* [47] showed that the previous correlations between object-oriented metrics and fault-proneness are mostly due to the correlations between the metrics and size. Hassan [48] observed that a complex code-change process negatively affects programs. He measured the complexity of code change through entropy and showed that the proposed change complexity metric is a better predictor of faults than other previous predictors.

7.3 Identifiers and Program Comprehension

Marcus [15] studied several open-source programs and found that about 40% of the domain terms were used in the source code. Unfortunately, in collaborative environments, the probability of having two developers use the same identifiers for different entities is between 7% and 18% [17]. Thus, naming conventions are crucial for improving the source code understandability. Butler *et al.* [16] analyzed the impact of naming conventions on maintenance effort, *i.e.*, on code quality. They evaluated the quality of identifiers in eight open-source Java libraries using 12 naming conventions. They showed that there exists a statistically significant relation between flawed identifiers (*i.e.*, violating at least one convention) and code quality.

The role played by identifiers and comments on source code understandability has been empirically analyzed by Takang *et al.* [49], who compared abbreviated identifiers with full-word identifiers and uncommented code with commented code. They showed that (1) commented programs are more understandable than non-commented programs and (2) programs containing full-word identifiers are more understandable than those with abbreviated identifiers. Similar results have also been achieved by Lawrie *et al.* [50]. These latter studies also showed that, in many cases, abbreviated identifiers are as useful

as full-word identifiers. Recently, Binkley *et al.* [51] performed an empirical study of the impact of identifier style on code readability and showed that Camel-case identifiers allow more accurate answers. Recently Sharif *et al.* replicated the same study with different types of subjects [52]. The result of their study indicates that there is no difference between the two styles in terms of accuracy. However, a significant improvement in time and visual effort was reported for underscored identifiers.

Hill *et al.* proposed a technique for static code search [20]. Natural language phrases are created for each program entities (e.g. methods, fields, and constructors). These phrases are returned as a result of a query if the query words have the same order as the words in the phrases. The authors present their algorithm for phrase extraction technique. The extraction process consists of four steps: (1) the entity name is split to space delimited phrase, (2) then entity name will be categorized as noun phrase, verb phrase, or prepositional phrase, (3) verb, direct/indirect object and prepositions are identified for verb phrases (4) additional phrases are created based on the arguments of methods. The authors performed an empirical study to compare the proposed approach to an existing work in terms of efforts and effectiveness. The results of empirical study showed that the newly developed technique significantly outperforms the existing work. Kuhn *et al.* apply information retrieval technique to extracts topics form source code identifiers and comments. The authors combined LSA and clustering techniques for semantic clustering of source code entities (e.g., packages, methods, classes). First similarity between the entities is computed and then the entities are clustered based on their similarity values. These clusters partition the source code in to different topics. The authors proceed by applying LSA techniques in each cluster to label the clusters by the most relevant topic. The authors then used visualization technique to identify parts of source code implementing those topics. Packages in the system are visualized as rectangular each containing small squares corresponding to classes. Classes are colored differently based on the cluster they belong to. They applied their technique on several software systems from different application domain and programming languages. The results of their study showed the effectiveness of their approach for providing knowledge about an unfamiliar system to the developers.

7.4 Refactoring

Refactoring is proposed by Martin Fowler [53] and is defined as a change made to the structure of a system without changing its observable behavior. Refactoring is a transformation activity. The main idea is to redistribute classes and class features (attributes, methods) across the class hierarchy in order to improve readability and maintainability

and to facilitate future adaptations and extensions. It is done in small steps that are interleaved with tests and is not bound to implementation. Different types of refactoring is proposed by the author, for example: Composing methods, Moving features between objects, Making method calls simpler, etc. Although structural refactoring performed at code level improves code quality, we believe that linguistic refactoring can also improve code quality specifically program comprehension.

Caprile *et al.* proposed a semi-automatic approach for refactoring identifiers [13]. To refactor an identifier, first it is split to its terms and then each term is mapped to its standard form based on a dictionary of standard form and a synonym dictionary. In cases where a term cannot be map to a term in neither of the dictionaries human intervention is needed. This process is referred to as lexicon standardization. In the next step, arrangement of the terms in the identifier is standardized based on a grammar proposed by the authors. This grammar works only for function identifiers (methods, functions, or procedures names). According to this grammar a function is categorized into one of the three groups of: action, property checker, and transformation. Thus, a function identifier has specific term arrangement based on the category to which the function belongs. The authors applied the proposed technique on a file compression application called *gzip* that is written in C and on a Java program implementing the user interface of the tool they developed. The result of study showed that function identifiers were refactored to more meaningful standard forms. In this work vocabulary of the method signature was used for refactoring. However, we believe that linguistic information that contains in the method body should be taken into account as well.

7.5 Text Summarization

Gong and Liu [19] proposed two methods for text summarization. In both methods, a given document is transformed into a term by sentence matrix. The first method, which is based on IR technique, select candidate sentences for summary with the highest relevance score. The relevance score for each sentence is the result of an inner product of the sentence vector (column of term by sentence matrix) and the whole matrix (document). The second method applies Latent Semantic Analysis for selecting sentences for summary. Sentences with the largest index value with the most important singular value are selected for summary. The authors showed that these techniques have the same performance when their results were compared against the summarization of three human evaluators.

Steinberger and Jezek [18] proposed an LSA based technique for text summarization. Moreover, they proposed a technique to evaluate the quality of the summaries. The same

authors recently extended the above mentioned work to update document summarization [37]. The idea behind this work is to compare a new set of documents (to be summarized) with an older versions (that already have summaries), and mark topics in the corpus as: novel, significant, or redundant. Then, an updated version of a previous summary can be created by selecting sentences that contains just the novel and significant topics.

Similarly, Kireyev [54] proposed an LSA based technique for text summarization and update summaries. In [55] Haiduc *et al.* used text summarization technique for program comprehension. The main purpose of this study is to support developers during program comprehension activity by extracting knowledge from code. However, we process this knowledge and use it for suggesting linguistic refactoring.

Chapter 8

Conclusion

In software engineering, maintenance cost 60% of overall project lifecycle costs of any software product. Program comprehension is a substantial part of maintenance. Identifiers are among the most important sources of information for program comprehension activity. Hence, naming conventions are crucial for improving the source code comprehensibility. We introduced term entropy and context-coverage to measure, respectively, how rare and scattered across program entities are terms and how unrelated are the entities containing them. We perform an empirical study on two open source softwares, ArgoUML and Rhino, to investigate the relation between terms with high entropy and high context-coverage and the fault proneness of the entities using those terms. The empirical study showed that there is a statistically significant relation between attributes and methods whose terms have high entropy and high context-coverage, on the one hand, and their fault proneness, on the other hand.

The result of this empirical study motivates us to investigate linguistic refactoring. Identifier names if selected wisely should reflect the responsibility and characteristic of the entity they are labeling. We plan to first apply summarization techniques to extract important topics implicit in the method body. Next, by analysis these topics we plan to identify linguistic refactorings. For example, renaming a method name based on the topics extracted, as these topics suggest the characteristic and responsibility of the method. Finally we will perform an empirical study to evaluate if proposed refactoring can indeed improve program comprehension and visual effort.

Bibliography

[1] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the International Conference on Software Engineering (ICSE) track on The Future of Software Engineering*, 2000.

[2] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[3] Jagdish Bansiya and Carl G. Davis. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering*, 28(1): 4–17, 2002.

[4] Kuljit Kaur Chahal and Hardeep Singh. Metrics to Study Symptoms of Bad Software Designs. *Software Engineering Notes*, 34(1):1–4, 2009.

[5] Geert Poels and Guido Dedene. Evaluating the Effect of Inheritance on the Modifiability of Object-Oriented Business Domain Models. In *Proceedings of the* 5^{th} *European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2001.

[6] Lutz Prechelt, Barbara Unger, Michael Philippsen, and Walter Tichy. A Controlled Experiment on Inheritance Depth as a Cost Factor for Code Maintenance. *Journal of Systems and Software*, 65(2):115–126, 2003.

[7] R. Harrison, S. Counsell, and R. Nithi. Experimental Assessment of the Effect of Inheritance on the Maintainability of Object-Oriented systems. *Journal of Systems and Software*, 52(2-3):173–179, 2000.

[8] Melis Dagpinar and Jens H. Jahnke. Predicting Maintainability with Object-Oriented Metrics- An Empirical Comparison. In *Proceedings of the* 10^{th} *Working Conference on Reverse Engineering (WCRE)*, page 155. IEEE Computer Society, 2003.

[9] Hector M. Olague, Letha H. Etzkorn, Sampson Gholston, and Stephen Quattlebaum. Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile

Software Development Processes. *IEEE Transactions on Software Engineering*, 33 (6):402–419, 2007.

[10] Hector M. Olague, Letha H. Etzkorn, Sherri L. Messimer, and Harry S.Delugach. An Empirical Validation of Object-Oriented Class Complexity Metrics and Their Ability to Predict Error-Prone Classes in Highly Iterative, or Agile Software: A Case Study. *Journal of Software Maintenance and Evolution*, 20(3):171–197, 2008.

[11] S. Kanmani, V. Rhymend Uthariaraj, V. Sankaranarayanan, and P. Thambidurai. Object-Oriented Software Quality Prediction Using General Regression Neural Networks. *Software Engineering Notes*, 29(5):1–6, 2004.

[12] A. Gunes Koru and Jeff (Jianhui) Tian. Comparing high-change modules and modules with the highest measurement values in two large-scale open-source products. *IEEE Transaction on Software Engineering*, 31(8):625–642, 2005.

[13] B. Caprile and P. Tonella. Restructuring Program Identifier Names. In *Proceedings of the* 16^{th} *IEEE International Conference on Software Maintenance*, pages 97–107, San Jose, California, USA, 2000. IEEE CS Press.

[14] F. Deissenboeck and M. Pizka. Concise and Consistent Naming. *Software Quality Journal*, 14(3):261–282, 2006.

[15] S. Haiduc and A. Marcus. On the Use of Domain Terms in Source Code. In *Proceedings of the* 16^{th} *IEEE International Conference on Program Comprehension*, pages 113–122. IEEE CS Press, 2008.

[16] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Relating Identifier Naming Flaws and Code Quality: An Empirical Study. In *Proceedings of the* 16^{th} *Working Conference on Reverse Engineering*, pages 31–35. IEEE CS Press, October 2009.

[17] G. Butler, P. Grogono, R. Shinghal, and I. Tjandra. Retrieving Information From Data Flow Diagrams. In *Proceedings of the* 2^{nd} *Working Conference on Reverse Engineering*, pages 84–93. IEEE CS Press, 1995.

[18] Josef Steinberger and Karel Jezek. Text summarization and singular value decomposition. In *Proceedings of* 3^{rd} *International Conference on Advances in Information Systems (ADVIS)*, pages 245–254, 2004.

[19] Yihong Gong and Xin Liu. Generic text summarization using relevance measure and latent semantic analysis. In *Proceedings of the* 24^{th} *annual international ACM SIGIR conference on Research and development in information retrieval*, pages 19–25, 2001.

[20] Emily Hill, Lori Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the* 31^{st} *International Conference on Software Engineering (ICSE)*, pages 232–242. IEEE Computer Society, 2009.

[21] 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* 26^{th} *IEEE International Conference on Software Maintenance- ERA Track*, 2010.

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

[23] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications John Wiley & Sons., 1992.

[24] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[25] V. Basili, G. Caldiera, and D. H. Rombach. *The Goal Question Metric Paradigm, Encyclopedia of Software Engineering*. John Wiley and Sons, 1994.

[26] Lerina Aversano, Gerardo Canfora, Luigi Cerulo, Concettina Del Grosso, and Massimiliano Di Penta. An empirical study on the evolution of design patterns. In *Proceedings of the* 6^{th} *joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering (ESEC-FSE)*, pages 385–394, New York NY USA, 2007. ACM Press.

[27] Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Transaction on Software Engineering*, 34(4):497–515, 2008.

[28] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1):1–34, Jan 2010.

[29] 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.

[30] L. Briand, J. Wüst, John W. Daly, and V. Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51:245–273, 2000.

[31] 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.

[32] David Binkley, Henry Feild, Dawn Lawrie, and Maurizio Pighin. Software fault prediction using language processing. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 99–110, Washington, DC, USA, 2007. IEEE Computer Society.

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

[34] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of* 25^{th} *International Conference on Software Engineering*, pages 125–135, Portland, Oregon, USA, 2003.

[35] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artefact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology*, 16(4), 2007.

[36] N.E. Fenton and S.L Pfleeger. *Software Metrics: A Rigorous and Practical Approach (2nd Edition)*. Thomson Computer Press, Boston, 1997.

[37] Josef Steinberger and Karel Ježek. Update summarization based on latent semantic analysis. In *Proceedings of the* 12^{th} *International Conference on Text, Speech and Dialogue (TSD)*, pages 77–84, Berlin, Heidelberg, 2009. Springer-Verlag.

[38] Hector M. Olague, Letha H. Etzkorn, and Glenn W. Cox. An entropy-based approach to assessing object-oriented software maintainability and degradation - a method and case study. In *Software Engineering Research and Practice*, pages 442–452, 2006.

[39] Yong Yu, Tong Li, Na Zhao, and Fei Dai. An approach to measuring the component cohesion based on structure entropy. In *Proceedings of the* 2^{nd} *International Symposium on Intelligent Information Technology Application*, pages 697–700, Washington, DC, USA, 2008. IEEE Computer Society.

[40] Greg Snider. Measuring the entropy of large software systems. Technical report, HP Laboratories Palo Alto, 2001.

[41] L. H. Etzkorn, S. Gholston, and W. E. Hughes. A semantic entropy metric. *Journal of Software Maintenance: Research and Practice*, 14(5):293–310, 2002.

[42] Y. Liu, D. Poshyvanyk, R. Ferenc, T. Gyimóthy, and N. Chrisochoides. Modelling class cohesion as mixtures of latent topics. In *Proceedings of* 25^{th} *IEEE International Conference on Software Maintenance*, pages 233–242. IEEE CS Press, 2009.

[43] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.

[44] S. Patel, W. Chu, and R. Baxter. A measure for composite module cohesion. In *Proceedings of* 14^{th} *International Conference on Software Engineering*, pages 38–48, Melbourne, Australia, 1992. ACM Press.

[45] D. Poshyvanyk and A. Marcus. The conceptual coupling metrics for object-oriented systems. In *Proceedings of* 22^{nd} *IEEE International Conference on Software Maintenance*, pages 469 – 478, Philadelphia Pennsylvania USA, 2006. IEEE CS Press.

[46] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the* 3^{rd} *International Workshop on Predictor Models in Software Engineering*, May 2007.

[47] K. El Emam, S. Benlarbi, N. Goel, and S.N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transaction on Software Engineering*, 27(7):630–650, July 2001.

[48] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 78–88. IEEE Computer Society, 2009.

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

[50] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *Proceedings of* 14^{th} *IEEE International Conference on Program Comprehension*, pages 3–12. IEEE CS Press, 2006.

[51] D. Binkley, M. Davis, D. Lawrie, and C. Morrell. To CamelCase or Under score. In *Proceedings of* 17^{th} *IEEE International Conference on Program Comprehension*. IEEE CS Press, 2009.

[52] Bonita Sharif and Jonathan I. Maletic. An eye tracking study on camelcase and under_score identifier styles. *International Conference on Program Comprehension*, pages 196–205, 2010.

[53] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.

[54] Kirill Kireyev. Using latent semantic analysis for extractive summarization. In *Proceedings of Text Analysis Conference (TAC)*, 2008.

[55] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In *Proceedings of the* 32^{nd} *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 223–226, 2010.

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

École Polytechnique de Montréal

École affiliée à l'Université de Montréal

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

www.polymtl.ca

ÉCOLE
POLYTECHNIQUE
MONTRÉAL