Titre: Title:	Defining linguistic antipatterns towards the improvement of source code quality
Auteurs: Authors:	Venera Arnaoudova
Date:	2010
Type:	Rapport / Report
Référence: Citation:	Arnaoudova, Venera (2010). Defining linguistic antipatterns towards the improvement of source code quality. Rapport technique. EPM-RT-2010-07.

EPM–RT–2010-07

DEFINING LINGUISTIC ANTIPATTERNS TOWARDS THE IMPROVEMENT OF SOURCE CODE QUALITY

Venera Arnaoudova
Département de Génie informatique et génie logiciel
École Polytechnique de Montréal

Septembre 2010

Poly

ÉCOLE POLYTECHNIQUE MONTRÉAL

EPM-RT-2010-07

Defining linguistic antipatterns towards the improvement of source code quality

Venera Arnaoudova
Département de génie informatique et génie logiciel
École Polytechnique de Montréal

Septembre 2010

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

EPM-RT-2010-07
Defining linguistic antipatterns towards the improvement of source code quality
par : Venera Arnaoudova
Département de génie informatique et génie logiciel
École Polytechnique de Montréal

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Defining linguistic antipatterns towards the improvement of source code quality

by

Venera Arnaoudova

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

in the

Département de génie informatique et génie logiciel

September 2010

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Abstract

Département de génie informatique et génie logiciel

Doctor of Philosophy

by Venera Arnaoudova

Previous studies showed that linguistic aspect of source code is a valuable source of information that can help to improve program comprehension. The proposed research work focuses on supporting quality improvement of source code by identifying, specifying, and studying common negative practices (*i.e.*, linguistic antipatterns) with respect to linguistic information. We expect the definition of linguistic antipatterns to increase the awareness of the existence of such bad practices and to discourage their use. We also propose to study the relation between negative practices in linguistic information (*i.e.*, linguistic antipatterns) and negative practices in structural information (*i.e.*, design antipatterns) with respect to comprehension effort and fault/change proneness. We discuss the proposed methodology and some preliminary results.

Contents

List of Figures

List of Tables

Abbreviations

CK	**C**hidamber and **K**emerer
IR	**I**nformation **R**etrieval
LM	**L**anguage **M**odel
LSI	**L**atent **S**emantic **I**ndexing
NL	**N**atural **L**anguage
numHEHCC	**Num**ber of terms with **H**igh **E**ntropy and **H**igh **C**ontext **C**overage

Chapter 1

Introduction

Program comprehension is preliminary to any maintenance activity because developers must first identify and understand relevant code fragments before performing any activity. ISO/IEC and IEEE define maintenance as the modification of a software product after delivery to correct faults, improve performance (or other quality attributes), or to adapt the product to a modified environment (ISO/IEC 14764:2006(E); IEEE Std 14764-2006). Maintenance is not a uniform activity and as the type of required changes may vary, four different types of maintenance can be identified [1]. Corrective maintenance includes all changes made to a system after deployment to correct problems. Preventive maintenance includes all changes made to a system after deployment to prevent faults to become failures. Adaptive maintenance includes all changes made to a system after deployment to support operability in a different environment. Perfective maintenance includes all changes made to a system after deployment to address new requirements. Adaptive and perfective types of maintenance are shown in the literature to consume a significantly large proportion of all maintenance effort. Corrective maintenance is reported to consume a relatively small proportion of the overall maintenance effort. The rest of the effort is consumed while applying preventive maintenance.

Source code of good quality in terms of comments and identifiers eases program comprehension because developers use identifiers to build their mental models of the code under analysis. Several studies showed that identifiers impact program comprehension (*e.g.*, [2–4]) and code quality [5].

Poorly-chosen identifiers could be misleading and also increase the risk of faults. Fault-prone entities, *i.e.*, classes, methods, and attributes, in object-oriented programs have been characterized by their internal characteristics (*e.g.*, [6–8]). However, fault proneness is a complex phenomenon hardly captured by a single characteristic, such as complexity

or size. We believe that linguistic information will provide additional information, complementary to the one captured by structural measures, and consequently will improve the characterization of fault prone entities.

Our research hypothesis is that poor linguistic information (*e.g.*, lack of relevant information, out-dated information, inconsistency with the rest of the software artifacts) impact negatively the quality of the code and the overall quality of the system. Thus, in this work, we are interested in 1) studying the relation between source code identifiers and code quality, 2) identifying common linguistic negative practices that increase the effort for comprehension, and 3) studying the relation between linguistic and design negative practices.

The rest of this document is organized as follows: In Chapter 2, we provide the necessary background for this research. In Chapter 3, we discuss our motivation and the problem statement we are addressing. Chapter 4 gives an overview of the proposed methodology. In Chapter 5, we discuss preliminary results with regards to program identifiers and fault proneness. In Chapter 6, we discuss ongoing work on the definition of linguistic antipatterns. In Chapter 7, we present related work. Chapter 8 defines current and future activities with respect to this research. We present preliminary conclusions in Chapter 9.

Chapter 2

Background

Before defining linguistic negative practices, in this chapter, we provide a brief background on common best practices in software engineering (design patterns) and common worst practices in software engineering (antipatterns).

2.1 Design Patterns

Design patterns were initially introduced by Christopher Alexander in the domain of architecture and urban design. "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such way that you can use this solution a million times over, without ever doing it the same way twice" [9].

Software design patterns were later popularized by Gamma et *al.* [10] to document recurring problems and their respective solutions that developers encounter during the development and maintenance phases. Design patterns are based on experience and the goal of identifying and document them is the reuse of knowledge and experience of people, that encountered similar problems in the past.

Design patterns are documented using the following template:

- **Pattern name**: A name describing the problem and solution. The name will become part of the vocabulary of developers and thus it should be chosen with a particular care.
- **Problem**: The problem describes the problem and the context in which one should apply the pattern.

- **Solution**: A design pattern provides a general solution to the problem that may be applied in different situations.
- **Consequences**: The consequences describe the expected benefits of applying the pattern as well as trade-offs. The purpose of the consequences is to help developers to decide whether it worth applying the pattern in the particular situation.

2.2 Design Antipatterns

Antipatterns are defined by Brown et *al.* [11] as recurring solutions with negative impact on software systems. While design patterns aim at promoting best practices, the goal of documenting antipatterns is to make developers aware of situations where a solution can have negative consequences, because being aware of eventual negative consequences help to avoid them.

An antipattern can be documented in three different forms, namely pseudo-, mini-, and full antipattern templates. The pseudo-antipattern template contains only name and problem. This is not a very common template because it does not provide all necessary information (*e.g.*, what would be a better solution). The mini-antipattern template contains name, problem, and solution sections. It is more complete than the pseudo-antipattern template and it is used as a non-formal antipattern description.

When documented in details, an antipattern is presented in its full template, composed of:

- **Antipattern name**: The name of the antipattern is intended to be pejorative.
- **Also Known As**: Other names the antipattern is known as.
- **Most Frequent Scale**: The level at which the antipattern is defined. Scale can be one or more of the following: idiom, micro-architecture, framework, application, system, enterprise, or global/industry.
- **Refactored Solution Name**: The name with which the refactored solution is known as.
- **Refactored Solution Type**: The type of the refactored solution corresponds to the type of action that results from this solution. It can be software (a new software should be purchased), technology (results in the adoption of a new technology), process (a process is defined), role (the solution results in assigning responsibilities).

- **Root Causes**: Enumerates causes of the antipattern. Can be one or more from the following: haste, apathy, narrow-mindedness, sloth, avarice, ignorance, pride, or responsibility (the universal cause).
- **Unbalanced Forces**: Point to the primal forces that have been ignored or underestimated. Primal forces are: management of functionality (meeting the requirements), management of performance (meeting required speed of operation), management of complexity (defining abstractions), management of change (controlling evolution of software), management of IT resources (controlling use and implementation of people and IT artifacts), management of technology transfer (controlling technology change).
- **Anecdotal Evidence [optional]**: Situations or expressions heard with the antipattern.
- **Background [optional]**: Background information that may be of interest.
- **General Form of this Antipattern**: A general characterization of the antipattern. Often in terms of diagrams.
- **Symptoms and Consequences**: List the symptoms (the apparent negative implications) and the consequences (anticipated negative implications).
- **Typical Causes**: The typical causes of the antipattern.
- **Known Exceptions**: Situations where the antipattern is known not to imply negative consequences.
- **Refactored Solutions**: The refactored solution in terms of steps to be undertaken.
- **Variations [optional]**: Variations of the antipattern and the refactored solution.
- **Example**: An example of how the solution is applied on an instance of the antipattern.
- **Related Solutions**: Clarifications with respect to related antipatterns. This section also includes references, terminology and resources.
- **Applicability to Other Viewpoints and Scales**: Description of the relevance of the antipattern on different levels.

Chapter 3

Motivation and problem statement: linguistic antipatterns

Studies in the literature showed that identifiers are among the most important sources of information to understand source code entities. Deißenböck and Pizka observed that 70% of the source code of Eclipse 3.0 consists of identifiers [2]. Haiduc and Marcus [3] studied several open-source systems and found that about 40% of the system domain terms were used in the source code. The lack of comments, poor coding standards, ambiguous or poorly selected identifiers impair code evolvability and increase the risk of introducing faults while performing evolution tasks.

From [2, 3, 12], we conjecture that linguistic information extracted from source code might highlight other aspects, not captured by structural metrics. We believe that those new aspects can help to understand and explain why certain entities are likely to pose program comprehension challenges for developers. Thus, linguistic information can help to locate methods, classes or code fragments likely difficult to understand. The more difficult to understand they are, the more difficult will be to change and evolve them without introducing defects. Several studies highlighted the importance of choosing the right identifiers and the impact of identifiers on program comprehension (*e.g.*, [2, 3, 13]) and code quality [5, 14, 15].

In this research work, we investigate the relation between, on the one hand, linguistic information in source code and, on the other hand, the quality of the software and the comprehension effort. We are interested in studying common practices, from linguistic aspect, in the source code that decrease the quality of the software or increase the comprehension effort of developers and maintainers. From those practices, we plan to define a new family of linguistic antipatterns that would be described in terms of symptoms, consequences, and alternative solutions. Based on the observation that proper

identifiers improve quality [2], we we believe that 1) identifiers should provide relevant information, 2) identifiers and documentation should be always up-to-date, and 3) there should be a consistency between linguistic information in source code and information in other software artifacts. Finally, we will investigate whether negative practices related to linguistic information are related to other types of negative practices such as design decisions.

Chapter 4

Methodology

To verify our hypothesis, we break down our methodology into four main steps. First, we plan to study the relation between linguistic information and the quality of source code. We then plan to improve the quality of source code by identifying, studying, and documenting common negative practices that increase comprehension effort. Next, we plan to investigate the automatic detection of linguistic antipatterns. Finally, we propose to study whether there exists a relation between linguistic antipatterns and design antipatterns. If a relation exists, linguistic antipatterns could be used to improve the quality of source code by increasing the accuracy of existing techniques detecting traditional design antipatterns. On the other hand, if linguistic and design antipatterns appear not to be related, the absence of a relation will open new venues in supporting program comprehension and software quality.

4.1 Linguistic Information and Source Code Quality

In this part, we are interested in the following research questions:

RQ1: Do physical and conceptual dispersions capture additional information with comparison to existing metrics used for fault and change explanation, such as LOC?

RQ2: Do physical and conceptual dispersions help to explain faults and changes in entities?

Inspired by previous work on the impact of source code identifiers on program comprehension [2, 3, 13] and code quality [5, 14, 15], we plan to investigate the relation between the way terms composing identifiers are used and the quality of the source code with respect to change and fault proneness. Term dispersion can be analyzed from two

aspects, namely physical and conceptual dispersions. Physical dispersion measures the degree to which a term is scattered across identifiers of different entities. Conceptual dispersion indicates how related these entities are. We believe that terms, highly used in unrelated entities (different contexts), may increase the odds ratio of those entities being faulty.

4.2 Linguistic Antipatterns

The research question we will answer here is:

RQ3: Are common linguistic negative practices *i.e.*, linguistic antipatterns related to software quality and program comprehension?

To define linguistic antipatterns, we plan to investigate negative common practices in several manners. We will start our investigation by analyzing the consistency of linguistic information extracted from a program with other aspects (*e.g.*, structural, dynamic). Next, we plan to study the applicability of ambiguity as defined in Natural Language (NL), *i.e.*, the taxonomy of ambiguities in NL Requirements Specifications defined by Berry and Kamsties [16]. Negative impact will be measured in terms of 1) change and fault proneness of entities containing linguistic antipatterns, and 2) developer comprehension level and effort. To measure comprehension level and effort, we will use devices such as eye tracking systems. We will perform a case study on subjects, asking them to perform modifications on systems, some of which containing linguistic antipatterns while others not. From those negative practices, from the symptoms that will allow us to identify them, and from the suggestions how to improve them, we will define linguistic antipatterns inspired from the design antipattern template of Brown [11]. Next, we plan to construct, with the help of minimum three software experts, an oracle of linguistic antipatterns of two systems — ArgoUML[1] and Rhino[2]. To validate the contribution of linguistic antipatterns on the quality of the source code, we will refactor the manually detected linguistic antipatterns and we will compare the quality of the source code on the original system and on the refactored system.

4.3 Linguistic Antipatterns Detection

The question that we are investigating in this part of the methodology is the following:

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

RQ4: Do linguistic antipatterns detection can help to improve code quality and program comprehension?

We believe that automatic antipattern detection can be beneficial from two aspect. First, it will allow developers to maximize the benefit of the defined linguistic antipatterns, by automatically identifying a set of bad practices in software systems. Second, it may allow a better understanding and a refinement of linguistic antipatterns, as different source code properties may happen to play an important role in the characterization of practices as antipatterns.

To detect linguistic antipatterns we will define rules based on existing or new measures and techniques such as the CK metrics suite [17] for structural information; Maximal Weighted Entropy (MWE) [18], Conceptual Cohesion of Classes (C3) [15], and Conceptual Coupling of a Class (CoCC) [14] for linguistic information; DEtection & CORection (DECOR) [19] for design antipattern detection. We also plan to investigate the potential benefit of using n-gram probability estimations on linguistic antipattern detection. N-gram probability estimations are used in Language Models (LM) to estimate the likelihood of discovering a piece of information given some previous knowledge.

We will validate the automatic detection of linguistic anti-patterns against the oracles built by the experts.

4.4 Linguistic and Design Antipatterns

The research question that we are interested in is:

RQ5: Is there a relation between linguistic and design antipatterns?

The criminological theory of Broken Windows [20] states that if a broken window of a building is left unrepaired, soon other windows will be broken, regardless of the neighborhood. Deißenböck and Pizka [2] comment on the application of the theory of Broken Windows on source code naming, affirming that source code identifiers containing negative characteristics have higher risk of fast deterioration. Inspired from this observation we have reasons to believe that code fragments containing design antipatterns have higher risk of containing other types of negative characteristics, and thus including linguistic antipatterns. To verify this hypothesis, we plan to study whether a relation exists between linguistic and design antipatterns. To validate the findings, we will analyze the contribution of linguistic antipatterns on automatic detection of design antipatterns for

two systems, GanttProject and Xerces, for which design antipatterns have been manually validated [19, 21]. The contribution will be measured in terms of improved precision and recall.

Figure 4.1 summarizes the flow of our methodology.

4.5 Expected contributions

We expect that our work contributes to improve the quality of source code and the quality of the overall system. We expect **RQ1** and **RQ2** to confirm the existence of a relation between term dispersion and fault proneness. Defining and detecting linguistic antipatterns is expected to increase the awareness of the existence of bad practices and consequently discourage and decrease their use (**RQ3** and **RQ4**). We also expect that there exists a relation between linguistic and design antipatterns (**RQ5**), confirming the Broken Windows theory [20], and thus we expect that linguistic antipatterns will increase the accuracy of existing techniques for design antipatterns.

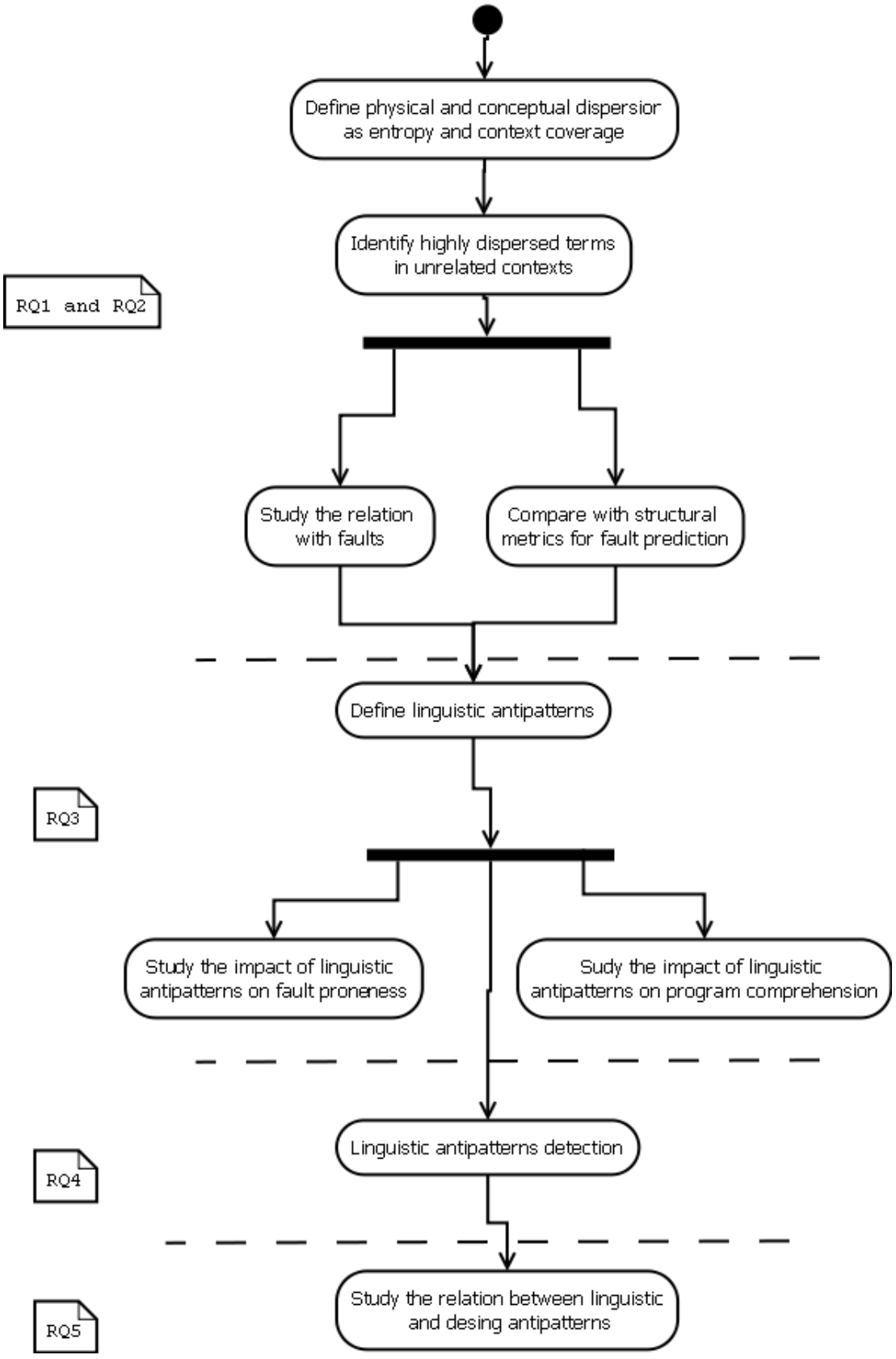

FIGURE 4.1: Flow of our methodology.

Chapter 5

Linguistic Information and Source Code Quality

In this chapter, we present our preliminary results that have been published in the Proceedings of the 26^{th} International Conference on Software Maintenance (ICSM'10) - ERA Track [22]. We address one aspect of research questions **RQ1** and **RQ2**, which is fault proneness.

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. 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. Entropy measures the *physical dispersion* of terms in a program: the higher the entropy, the more scattered across the program the terms. Context coverage measures the *conceptual dispersion* of terms: the higher their context coverage, the more unrelated the methods using them. 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.

The rest of this chapter is organized as follows. Section 5.1 introduces background definitions and defines the novel measure. Section 5.2 provides an overview of our approach. Section 5.3 describes our empirical study, reports, and discusses its results. Section 5.4 provides more details regarding the automation.

5.1 Definitions

In this section, we detail the computations of term entropy and context-coverage. With no loss of generality, we focus on methods and attributes because they are "small" contexts of identifiers. We include 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.

5.1.1 Term Entropy

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.

5.1.2 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 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:

$$CC(t_k) = 1 - \frac{1}{\left(\begin{array}{c} |C| \\ 2 \end{array} \right)} \sum_{\substack{i=1 \ldots |C|-1 \\ j=i+1 \ldots |C| \\ e_i, e_j \in C}} sim(e_i, e_j)$$

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. Note that the number of summations is $\left(\begin{array}{c} |C| \\ 2 \end{array} \right)$ 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.

5.1.3 Aggregated Metric

In this preliminary investigation 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, 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.

5.2 Approach

We now present a 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.

5.2.1 Research Questions

Entropy and context coverage likely capture features different from size or other classical object-oriented metrics, such as the CK metrics suite [17]. However, it is well known that size is one of the best fault predictors [6, 26, 27] 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 – 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.
- **RQ_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, 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 $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$ will have the same answer in all version of the two programs and will be universally true. Nevertheless, as previous authors [2, 4, 5, 28], 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.

5.2.2 Analysis Method

To statistically analyze $\mathbf{RQ}_1$, 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) [29] 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 $\mathbf{RQ}_1$ 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 ($\mathbf{RQ}_2$) 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) [29] 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) [29] 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).

TABLE 5.1: Correlation test for ArgoUML v0.16 and Rhino v1.4R3.

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

5.3 Case Study

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 [30, 31], 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 [30, 32].

5.3.1 Results

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

5.3.1.1 RQ_1 – Metric Relevance

Table 5.1 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 5.2). 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 5.2 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 5.3 reports the interaction model built between fault proneness (explained variable) and the explanatory variables LOC

TABLE 5.2: 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 5.3: 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

and $numHEHCC$. In both models, $M_{ArgoUML}$ and M_{Rhino}, the intercept is relevant as 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. However, in both models $M_{ArgoUML}$ and M_{Rhino}, the LOC coefficient is, at least, one order of magnitude smaller than the $numHEHCC$ coefficient. This difference 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 5.3 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}.

5.3.1.2 $\mathbf{RQ}_2$ – Relation to Faults

To answer $\mathbf{RQ}_2$, 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 5.4: 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 5.5: 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 5.6: 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 5.7: 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 5.4 and 5.5 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 5.6 and 5.7, 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, the latter seems to suggest that the real difference is between not containing high entropy and high context coverage terms and

TABLE 5.8: 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

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.

5.3.2 Discussion

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

5.3.2.1 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 $100 \leq k \leq 200$ [33, 34]. Therefore following such a heuristic approach we set the LSI subspace dimension equal to 100.

5.3.2.2 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.

5.3.2.3 Statistical Computations

All statistical computations were performed in R[1]. 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.

5.3.2.4 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 [6, 26, 27] 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.

5.3.3 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 5.8; for a fixed $numHEHCC$

[1] 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[2]) 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 5.8 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 or incorrect term entropy and context coverage values. We use manually-validated faults that have been used in previous studies [30]. 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 [32]. 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 [30, 32]. Also, we discussed the relation and lack

[2]Although the coefficient for LOC is not significant, it was taken into account for the calculation of odds because it has been shown in the literature that LOC is an important measure for fault prediction.

	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 5.1: 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 5.1. 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 [35], 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.

5.4 Automation

To create the term by entity matrix (for the purpose of this study, methods and attributes of Java classes are considered as documents and are reffed to as entities), Java classes need to be parsed and the identifiers need to be extracted. Next, then the identifiers are split into terms and the entropy and context coverage are then calculated for each term. We have used Java and R[3] to provide the automation. See below for a brief explanation.

[3] http://cran.r-project.org/

System	Total number of Java files	Number of not parsed files	Percentage
Rhino v1.4R3	75	1	0.01
Rhino v1.5R1	100	3	0.03
Rhino v1.5R2	105	2	0.02
Rhino v1.5R3	104	2	0.02
Rhino v1.5R4.1	107	1	0.01
ArgoUML v0.10.1	777	64	0.08
ArgoUML v0.12	850	64	0.08
ArgoUML v0.14	1077	57	0.05
ArgoUML v0.16	1124	53	0.04

TABLE 5.9: Unparsed files for both ArgoUML and Rhino.

5.4.1 Parsing

We used Java grammar 1.5 and JavaCC[4] to generate a Java parser that extracts the identifiers. To verify the completeness of the grammar, we have parsed 11 versions of Rhino and 11 versions of ArgoUML with our parser, Table 5.9 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 for Rhino 1.4R3. Since the percentage of not parsed files for each version was less than 0.08 percent, we decided to proceed with our parser instead of using existing parsers.

5.4.2 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*. We then apply two filters on the dictionary. First, we remove terms with a length less than two because their semantics is often unclear and because they most likely correspond to loop indexes (*e.g.*, I, j, k). Second, 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] https://javacc.dev.java.net/

5.4.3 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.

5.4.4 Execution

We download several versions of Rhino for which faults were documented by Eaddy et *al.* [30] from the Mozilla Web site[5]. Versions of ArgoUML were downloaded from the Tigris Community Web site[6]. 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 [30, 32] to tag methods and attributes and relate them with entropy and context coverage values. The following paragraphs detail each step.

5.4.4.1 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.* [30] for 11 versions of Rhino. We obtain the mapping which corresponds to Rhino v1.4R3 by extracting, for each fault, its reporting date/time[7] and its fixing date/time. Then, we keep only those faults that fall under one of the following two cases: (i) the reporting date of the fault was before the release date

[5] https://developer.mozilla.org/
[6] http://argouml.tigris.org/
[7] https://bugzilla.mozilla.org/query.cgi

of v1.4R3 and its fixing date was after the release date of the same version and (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 [32]. 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.

5.4.4.2 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 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.

5.5 Conclusion

In this chapter, we presented our preliminary results targeting **RQ1** and **RQ2** (Section 4.1) with respect to fault proneness. 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.

As part of our future work related to **RQ1** and **RQ2**, we plan to study the relation between term dispersion and change proneness, as well as to relate and study the interaction of entropy and context coverage with a larger suite of object-oriented metrics.

Chapter 6

Linguistic Antipatterns

There may be several causes for poor linguistic information. Thus we expect that there are different types of linguistic antipatterns such as lack of relevant information, outdated information, inconsistency with the rest of the software artifacts.

In this chapter, we explore linguistic antipatterns from the aspect of inconsistency with the design artifacts and we present an example of our ongoing work on linguistic antipatterns.

We started our investigation by analyzing the consistency between, on the one hand, linguistic information found in source code identifiers and comments, and, on the other hand, the design of a system in which identifiers and comments have been extracted. We know from experience that a considerable amount of design patterns are implemented but not documented. For developers who do not know the design pattern, the system may become more difficult to understand because of the extra code related to the pattern. Thus, the intention of reusing a solution (*i.e.*, applying a design motif) may have a negative impact on the system, by increasing its complexity. From this negative practice, we defined the linguistic antipattern that follows.

- **Antipattern name**: Invisible use of design pattern.
- **Most Frequent Scale**: Application.
- **Refactored Solution Name**: Pattern documentation.
- **Root Causes**: Haste, apathy, sloth.
- **Background**: The use of design patterns have many advantages such as reuse of design and knowledge, common language with other developers. Several works in the literature encourage the use of patterns during software development and

maintenance (*e.g.*, [10, 36–39]), other perform experiments to verify the benefit of their use (*e.g.*, [40, 41]).

Sometimes, the use of design patterns in a software system can be identified easier than others for various reasons: i) because developers use some patterns more often than others and thus those patterns are more well-known by developers and maintainers (*e.g.*, *Singleton*), ii) because the name of some patterns or the roles of the participants are part of the names of some classes (*e.g.*, *Visitor*), or iii) because some patterns are documented in terms of comments or other types of documentation (*e.g.*, in JHotDraw[1] patterns appears in the documentation). Thus, documenting the use of patterns plays an important role to take better advantage of their benefits. However, more often than not, developers assume that design patterns are well-known and their implementation can be easily identified from the structure of the design. We concur with Kerievsky that the use of design patterns can make code look more complex for people that are not aware of the used pattern [42].

- **General Form of this Antipattern**: This antipattern is characterized by the absence of the design pattern in the linguistic information of the source code, *i.e.*, the pattern name or the roles of the participants do not exist, or is not explicit enough, neither in the participants names, nor in the comments/documentation.
- **Symptoms and Consequences**:
  - The class does not seem to be part of the domain concepts.
  - It is not clear what is the role of the class in the overall design because of the lack of comments or unproper naming (the name of the class does not provide enough information about its purpose/role).
  - Comments and documentation do not indicate the use of any design pattern.
  - The name of the class does not suggest the use of design patterns.
- **Typical Causes**:
  - Lack of documentation: The design pattern is not documented.
  - Sloppy naming: Names do not include any hint about the pattern implementation.
- **Refactored Solutions**: The refactored solution involves renaming and documentation.

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

1. Rename the participants in the design pattern to include the name of the pattern or the roles they are playing in the pattern if they are self explanatory enough.
2. Document the use of the design pattern in terms of comments or other type of documentation (*e.g.*, javadoc). Other participants in the design pattern should be also enumerated and their roles should be explicitly stated.

Special cases:

1. It may happen that a class participate in more than one patterns. In this case, adding the roles of a class to its name may result in more complex and difficult to understand name. Thus, it is preferable to document the pattern through comments or other types of documentation only.
2. Sometimes, pattern participants are part of a library and the source code is not available for modification. In this case, the only possible documentation of the pattern is via the other participants.

- **Example**: The *Adapter* design pattern brings a solution when two classes are not compatible with each other. It converts one interface to another, the latter being expected by and compatible with the client. There are two types of implementations of the *Adapter* pattern, namely class (see Figure 6.1) and object (see Figure 6.2). More details on the two implementations are provided by Gamma et *al.* [10].

  Consider a fraction of a class diagram depicted in Figure 6.3, which is an instance of the object *Adapter* design pattern. The example is taken from JRefactory[2] v2.6.24. If a developer that knows the structure of the *Adapter* design pattern is analyzing the code, he could identify the implementation only if the code is reverse-engineered to obtain the class diagram or by browsing through the source code of class `MoveItemAdapter`. To identify the other participants in the pattern, the developer should manually navigate through the classes interacting with `MoveItemAdapter` and identify their roles. However, even if the developer is familiar with the *Adapter* pattern, identifying the roles of the participants is not straightforward because the actual implementation seems to be a variation of the pattern and extra dependencies exist between the classes. Moreover, if the developer is not familiar with the *Adapter* pattern, he does not have any hint that a pattern is used and will spend some considerable time and effort to understand the logic behind the design.

  Figure 6.4 depicts the partial class diagram after performing the suggested refactoring. To reflect the role of each class in the pattern some classes were renamed:

[2] `http://jrefactory.sourceforge.net/`

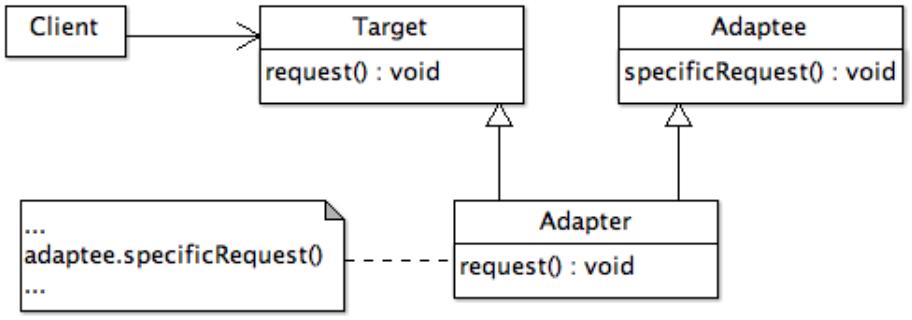

FIGURE 6.1: Class implementation of the *Adapter* design pattern.

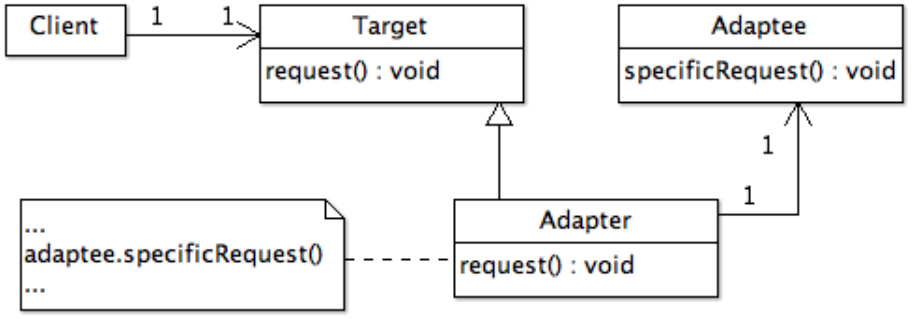

FIGURE 6.2: Object implementation of the *Adapter* design pattern.

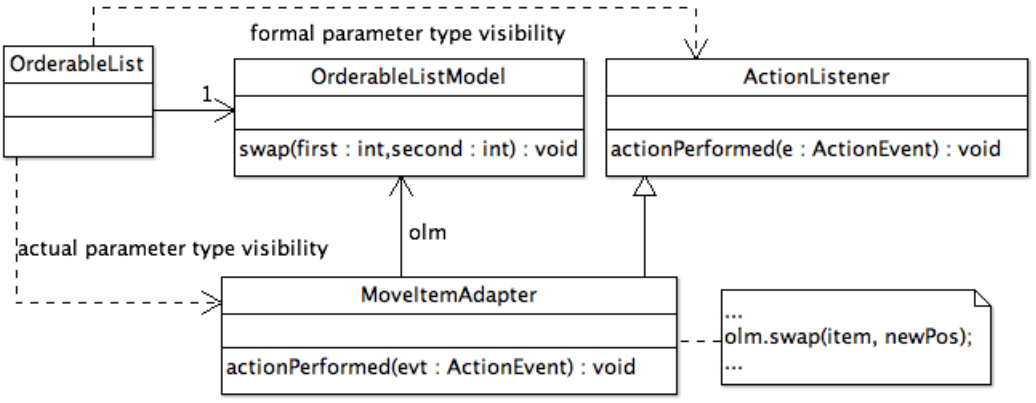

FIGURE 6.3: Instance of Invisible Use of Design Pattern.

class `OrderableListModel` was renamed to `OrderableListModelAdaptee`; class `OrderableList` was renamed to `OrderableListClient`. Note that class `ActionListener` can not be renamed because it is part of the `awt` Java library. A documentation in terms of comments is also added to all participants explicitly stating the role of each class and the rest of the participants.

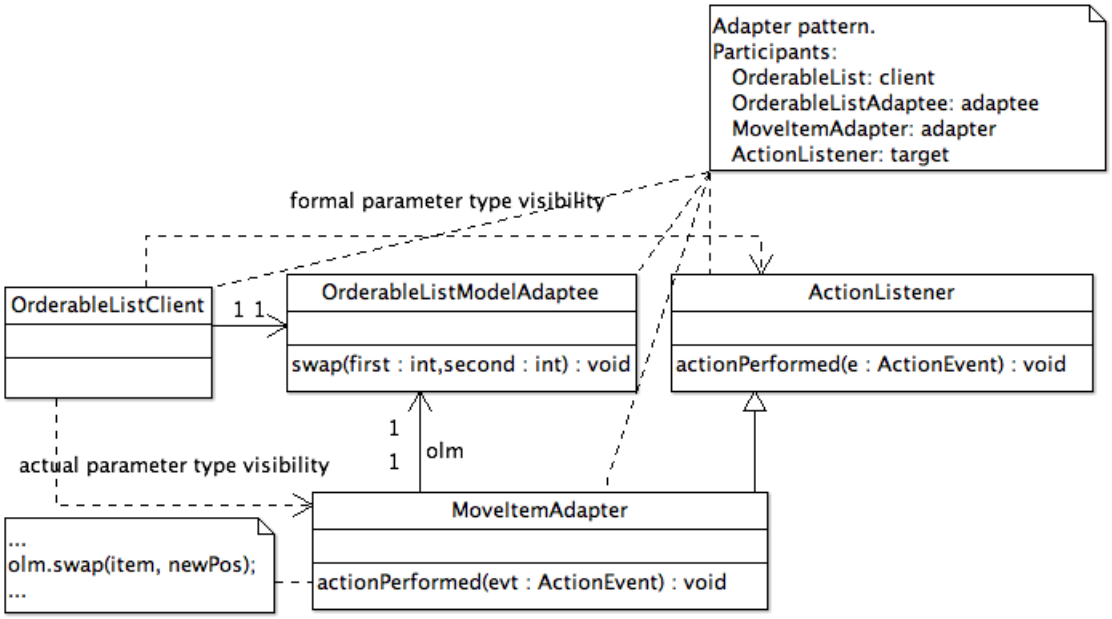

FIGURE 6.4: Instance of Invisible Use of Design Pattern - refactored.

Chapter 7

Related work

Our work is related to the following main categories of works: Information Retrieval (IR) techniques (Section 7.1), fault proneness (Section 7.2) and source code identifiers (Section 7.3) are related to **RQ1** and **RQ2** (Section 4.1); existing antipatterns definitions (Section 7.4), and research studying ambiguity in NL (Section 7.5) are related to **RQ3** (Section 4.2); antipattern detection techniques (Section 7.6) are related to **RQ4** (Section 4.3). To the best of our knowledge there is no previous work on studying the eventual relation between linguistic information and design antipatterns (**RQ5**, Section 4.4). Few works discuss the Broken Windows theory in software engineering (Section 7.7).

7.1 Entropy and IR-based Metrics

Several metrics based on entropy exist. Olague et *al.* [43] 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.* [44] combined entropy with component-dependency graphs to measure component cohesion. Entropy was also used by Snider [45] 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. Etzkorn *et al.* [46] presented a new measure for object-oriented programs that examines the implementation domain content of a class to measure its complexity. Patel et *al.* [47] and Marcus et *al.* [15] used Vector Space Model (VSM) and Latent Semantic Indexing (LSI) [24], respectively, to measure the semantic cohesion of a class. They used IR

methods 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 [14] defined new coupling metrics based on semantic similarity. Binkley et *al.* [28] 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.

Liu et *al.* [18] use linguistic information to propose a new measure, namely Maximal Weighted Entropy (MWE), to measure the cohesion of classes. The measure is based on Latent Dirichlet Allocation (LDA) and it is used to improve the results of models for software fault prediction based on structural cohesion. The authors validated the results on a C++ software system, namely Mozilla[1].

The measure that we define in this work is at finer grain (terms of source code identifiers) and combines entropy with context.

7.2 Metrics and Fault Proneness

Several researchers studied the correlations between static object-oriented metrics, such as the CK metrics suite [17], and fault proneness. For example, Gyimóthy et *al.* [6] compared the accuracy of different metrics from the 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.* [12] 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.* [48] showed that the previous correlations between object-oriented metrics and fault-proneness are mostly due to the correlations between the metrics and size. Hassan [49] 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.

Our work studies the importance of linguistic information for fault proneness explanation. In the future, we plan to relate linguistic information and the metrics defined and used in the literature for fault prediction.

[1] http://www.mozilla.org/

7.3 Linguistic Information in Source Code

Sridhara et *al.* [50] study the applicability of a set of English-based semantic similarity tools on source code identifiers and comments. The authors are interested in six types of word relations and they believe that those relations can improve automated software comprehension and analysis tools.

Lexical ambiguity, which is defined as a type of linguistic ambiguity, has been studied by Deißenböck and Pizka in source code identifiers [2]. The authors formalize the definition of concise and consistent identifiers names, and enforce the user to follow the formal rules through a tool prototype.

Caprile and Tonella proposed refactoring strategies for source code identifiers based on standard lexicon of terms and their arrangement [4].

Kuhn et *al.* introduce semantic clustering to identify topics in source code based on Latent Semantic Indexing (LSI) and clustering source code documents with similar vocabulary [51].

Haiduc and Marcus [3] 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% [52]. Thus, naming conventions are crucial for improving the source code understandability. Butler et *al.* [5] 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.* [53], 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.* [54]. These latter studies also showed that, in many cases, abbreviated identifiers are as useful as full-word identifiers. Recently, Binkley et *al.* [13] performed an empirical study of the impact of identifier style on code readability and showed that Camel-case identifiers allow more accurate answers.

Concern location techniques such as [55–57] aim at improving comprehension by extracting concepts from source code identifiers and comments.

We plan to build on these previous works for the definition and detection of linguistic antipatterns.

7.4 Antipatterns definition

Antipatterns became popular with the help of Brown et *al.* who defined a catalog of antipatterns from three different point of views, namely architectural, design, and management [11]. As described in Section 2.2, solution with negative consequences focuses on the structure of the software or on the management, whereas the type of the refactored solutions can be software, technology, process, or role. In our research, we focus on the linguistic aspect of software, thus the negative solution and the refactored solutions will focus on renaming and documentation. The linguistic aspect of source code is not addressed in depth by Brown et *al.* The authors provide brief recommendations to choose meaningful names for the participants of a design pattern and, when possible, try to incorporate the name of the pattern in the names of the participants. They give an example with the `Strategy` pattern used for a text compositing algorithm where names `SimpleLayoutStrategy` or `TeXLayoutStrategy` seem to be suitable choices. Brown et *al.* also recommend to decide on naming conventions for operations (*e.g.*, use the prefix `create-` for the Factory pattern) and be consistent. The authors also suggest that pattern should be part of documentation.

Later Brown co-authored two other books on antipatterns. The first book lists a catalog of antipatterns related to Software Configuration Management (SCM) [58]. The authors document antipatterns that can be encountered when managing the evolution of software projects (such as change and revision management). In the second book, Brown et *al.* [59] list set of project management antipatterns from three perspectives, namely people, technology and process.

Laplante and Neill [60] extend the catalog of Brown et *al.* [11] by enriching the catalog of management antipatterns and by creating a new category of environmental antipatterns. The former focuses on problems occurring with managers who fail in their leadership task, whereas the latter is not about individual problems but rather a company strategy, or a group of employees with negative consequences.

The work by Shoemaker [61] is driven by the importance of communication in software development process and the negative impact that may result from a failure to communicate customer needs. The author defines a set of best and worst practices while writing requirements. The antipatterns fall into both elicitation and analysis phases and are mostly defined from a point of view of a programmer.

Other related works [62–65] concentrate on antipatterns related to a specific technology or language.

Dudney et *al.* [62] publish a catalog of J2EE antipatterns. Through this catalog, the authors aim at helping developers to build better J2EE applications by documenting commonly made mistakes and solutions suggesting how to fix them. The antipatterns spread through various domains some of which are quite general (such as distributed computing, application scale, persistence, and service-based architecture), whereas others are J2EE specific (such as JSP, Entity Beans, and J2EE services).

Karwin [63] defines a set of SQL antipatterns and groups them into four categories, namely logical database design antipatterns, physical database design antipatterns, query antipatterns, and application development antipatterns. Logical database design antipatterns deal with common mistakes made while designing and organizing a database. Physical database design antipatterns are concerned with the actual table definition and the choice of types for the data. Query antipatterns deal with data insertion and retrieval. Finally, application development antipatterns document common mistakes made when using SQL in the context of other programming languages.

Tate [64] introduces a set of antipatterns in server-side Java programming. The antipatterns document misuse of base Java and J2EE concepts such as servlets, JSPs, and Enterprise JavaBeans (EJBs). The author also describe more general antipatterns such as misuse of XML, lack of coding standards, and memory leaks.

After Bitter Java, Tate co-authored Bitter EJB [65] in which the authors define a catalog of EJB antipatterns. As in previous related works, the authors also tackle more general issues such as persistence, performance, and testing.

Our work extends previous works on antipatterns by enriching the catalog with a new category of linguistic antipatterns defined following the template presented by Brown et *al.* [11].

7.5 Natural Language Processing

Ceccato et *al.* [66] discuss the identification and measurement of different types of ambiguity (lexical, syntactic, semantic, and pragmatic) in NL texts. The authors propose a prototype tool limited to the identification of lexical ambiguity only.

Willis et *al.* [67] define and study nocuous ambiguity, which is a subtype of coordination ambiguity occurring when the same expression can be interpreted differently by different persons. The authors are not interested in disambiguation of the expressions but rather

identifying nocuous ambiguity together with an ambiguity threshold (the latter being the degree to which the user can tolerate misunderstandings) and inform the user about a potential misunderstanding.

We plan to study the applicability of ambiguities defined in NL on source code identifiers for the definition of linguistic antipatterns.

7.6 Antipatterns Detection

Marinescu [68] proposes detection strategies for defining metrics-based rules to identify design antipatterns. The approach is high level and allows to software engineers to capture deviations from what they define as good design. The author defined detection strategies for more that ten design smells and validated the approach on two versions of a medium size business application.

Munro [69] also defines a detection mechanism based on a set of software metrics. The author provides feedback on the design based of pre-defined interpretation rules. The author validated the approach on one small and one medium scale systems detecting two design smells.

Moha et *al.* [70] use structural information to detect design anti-patterns. The authors provide an automatic generation of detection algorithms from specifications written in a domain specific language for four antipatterns (*Blob*, *Functional Decomposition*, *Spaghetti Code*, and *Swiss Army Knife*). Precision and recall were calculated on nine software systems, and results are promising: 100% recall and a precision greater than 50%. The authors detect design antipatterns based on structural information expressed in terms of code smells.

We plan to detect linguistic antipatterns based on linguistic measures and their consistency with structural/dynamic measures. We also plan to apply the work by Moha et *al.* [70] for the detection of design patterns to verify the Broken Windows theory.

7.7 Broken Windows Theory in Software Engineering

Hunt and Thomas define a list of tips that may be thought of as patterns for programmers [71]. One of these tips is the "Don't Live with Broken Windows" in which the authors recommend to fix broken windows (such as bad design decisions or badly written code) as soon as they appear in order to keep control over the situation.

Deienböck and Pizka [2] performed an experiment with graduate and undergraduate students over one year and three months respectively. Students were initially asked to develop a clone detection system using basic line-based mechanism. After a while the requirements changed and the line-based detection was to be replaced with a unit-based technique with units at different levels of granularity. Students implemented the modification, but even few months after, the identifier `line` was present in almost all modules. Consequently, *line* and `unit` became synonyms. The problem came after a new requirement, which is to provide the line-based technique as an optional detection mechanism. Identifiers named `line` were confusing because in some places they meant `line` while in others, they meant `unit`. The authors observed the effect of the Broken Windows theory as students started being more careless about the naming of new variables, because they considered the program as a mess already.

We plan to verify the Broken Windows theory by studying the co-occurrence of linguistic and design antipatterns.

Chapter 8

Research Plan

8.1 RQ1, RQ2 (Summer 2010 - Fall 2010)

These research questions investigate the relation between linguistic information and code quality. We can break them down into two activities as follows:

- 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* 26^{th} *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).

8.2 RQ3 (Winter 2011, Summer 2011)

This research question study the relation between, on one hand, linguistic antipatterns and, on the other hand, software quality and program comprehension. We plan to answer this research question through the following activities:

- Conduct an experiment with software developers on systems (some of which contain linguistic antipatterns, while others not) and measure the degree of effort that subjects provide to understand the systems.

  Possible publication: Empirical Software Engineering Journal.

- Study the relation between linguistic antipatterns and code quality.

  Possible publication: Software Quality Journal.

8.3 RQ4 (Fall 2011)

This research question deals with the automatic detection of linguistic antipatterns.

Possible publication: IEEE International Conference on Automated Software Engineering (ASE'12).

8.4 RQ5 (Winter 2012)

This research question studies the relation between linguistic and design antipatterns.

Possible publication: International Conference on Software Engineering (ICSE'2012).

Chapter 9

Conclusion

In this work we are interested in investigating the importance of linguistic information on system quality and comprehension effort that developers should provide to understand a piece of code. To this end, we broke down our research methodology into five research questions, presented in Chapter 4.

Up to now, we addressed one aspect of **RQ1** and **RQ2** (Section 4.1) by studying the relation between, on the one hand, fault proneness and, on the other hand, the conceptual and physical dispersions of terms extracted from source code identifiers (Chapter 5). We showed that terms highly used in different contexts increase the odds ratio of the entities containing them being buggy.

Currently, we are investigating **RQ3** (Section 4.2) by identifying, specifying, and studying negative linguistic practices from different aspects. In Chapter 6, we present an example of a linguistic antipattern in terms of inconsistency of linguistic information with design artifacts.

Preliminary findings that linguistic information is related to code quality are encouraging and show that the proposed research worth carrying on.

Bibliography

[1] Burton Swanson and Bennet P. Lientz. *Software Maintenance Management: A study of the management of computer application software in 487 data processing organizations*. Addison Wesley, 1980.

[2] Florian Deißenböck and Markus Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, September 2006.

[3] Sonia Haiduc and Andrian Marcus. On the use of domain terms in source code. In *Proceedings of the* 16^{th} *International Conference on Program Comprehension (ICPC'08)*, pages 113–122. IEEE Computer Society, June 2008.

[4] Bruno Caprile and Paolo Tonella. Restructuring program identifier names. In *Proceedings of* 16^{th} *International Conference on Software Maintenance (ICSM'00)*, pages 97–107. IEEE Computer Society, 2000.

[5] 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 (WCRE'09)*, pages 31–35. IEEE Computer Society, October 2009.

[6] 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 (TSE'05)*, 31(10):897–910, October 2005.

[7] Michelle Cartwright and Martin Shepperd. An empirical investigation of an object-oriented software system. *IEEE Transactions on Software Engineering (TSE'00)*, 26(8):786–796, August 2000.

[8] Taghi M. Khoshgoftaar, Edward B. Allen, Kalai S. Kalaichelvan, and Nishith Goel. Early quality prediction: A case study in telecommunications. *IEEE Software*, 13 (1):65–71, January 1996.

[9] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[11] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., March 1998.

[12] 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 (PROMISE'07)*, page 9. IEEE Computer Society, 2007.

[13] David Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. To CamelCase or Under_score. In *Proceedings of the* 17^{th} *International Conference on Program Comprehension (ICPC'09)*, pages 158–167. IEEE Computer Society, May 2009.

[14] Denys Poshyvanyk and Andrian Marcus. The conceptual coupling metrics for object-oriented systems. In *Proceedings of* 22^{nd} *International Conference on Software Maintenance (ICSM'06)*, pages 469–478. IEEE Computer Society, 2006.

[15] 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 (TSE'08)*, 34(2):287–30, March/April 2008.

[16] Daniel M. Berry and Erik Kamsties. *Perspectives on Software Requirements*, chapter Ambiguity in Requirements Specification, pages 7–44. Kluwer Academic Publishers, 2003.

[17] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering (TSE'94)*, 20(6):476–493, June 1994.

[18] Yixun Liu, Denys Poshyvanyk, Rudolf Ferenc, Tibor Gyimóthy, and Nikos Chrisochoides. Modeling class cohesion as mixtures of latent topics. In *Proceedings of the* 25^{th} *International Conference on Software Maintenance (ICSM'09)*, pages 233–242. IEEE Computer Society, 2009.

[19] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering (TSE'10)*, 36(1):20–36, January-February 2010.

[20] George L. Kelling and James Q. Wilson. Broken windows: The police and neighborhood safety. *The Atlantic Monthly*, March 1982.

[21] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. A Bayesian approach for the detection of code and design smells. In *Proceedings of the* 9^{th} *International Conference on Quality Software (ICQS'09)*, pages 305–314. IEEE Computer Society, 2009.

[22] 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} *International Conference on Software Maintenance (ICSM'10) - ERA Track*. IEEE Computer Society, 2010.

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

[24] 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.

[25] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. *The Goal Question Metric Paradigm*. John Wiley & Sons, 1994.

[26] Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51(3):245–273, May 2000.

[27] Yuming Zhou and Hareton Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on Software Engineering (TSE'06)*, 32(10):771–789, October 2006.

[28] 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. IEEE Computer Society, 2007.

[29] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & All, 4th edition edition, 2007.

[30] Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering (TSE'08)*, 34(4):497–515, July 2008.

[31] 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 the 6^{th} joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 385–394. ACM, 2007.

[32] 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, February 2010.

[33] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of 25^{th} International Conference on Software Engineering (ICSE'03)*, pages 125–135. IEEE Computer Society, 2003.

[34] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering traceability links in software artefact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology (TOSEM'07)*, 16(4):13, September 2007.

[35] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition edition, 1998.

[36] James W. Cooper. *The Design Patterns Java Companion*. Addison Wesley, 1998.

[37] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall, 2nd edition edition, 2001.

[38] Alan Shalloway and James Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Addison-Wesley Professional, 1st edition edition, 2001.

[39] Oscar Nierstrasz, Stéphane Ducasse, and Serge Demeyer. *Object-Oriented Reengineering Patterns*. Square Bracket Associates, 2009.

[40] Lutz Prechelt, Barbara Unger, Walter F. Tichy, Peter Brössler, and Lawrence G. Votta. A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering (TSE'01)*, 27(12):1134–1144, December 2001.

[41] Lutz Prechelt, Barbara Unger, Michael Philippsen, and Walter F. Tichy. Two controlled experiments assessing the usefulness of design pattern documentation in

program maintenance. *IEEE Transactions on Software Engineering (TSE'02)*, 28 (6):595–606, June 2002.

[42] Joshua Kerievsky. *Refactoring to Patterns*. Addison Wesley, 2004.

[43] 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 *Proceedings of the International Conference on Software Engineering Research and Practice (SERP'06)*, pages 442–452. CSREA Press, 2006.

[44] 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 (IITA'08)*, pages 697–700. IEEE Computer Society, 2008.

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

[46] Letha H. Etzkorn, Sampson Gholston, and William E. Hughes. A semantic entropy metric. *Journal of Software Maintenance: Research and Practice*, 14(5):293–310, July 2002.

[47] Sukesh Patel, William Chu, and Rich Baxter. A measure for composite module cohesion. In *Proceedings of* 14^{th} *International Conference on Software Engineering (ICSE'92)*, pages 38–48. ACM, 1992.

[48] Kalhed El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering (TSE'01)*, 27(7):630–650, July 2001.

[49] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the* 31^{st} *International Conference on Software Engineering (ICSE'09)*, pages 78–88. IEEE Computer Society, 2009.

[50] Giriprasad Sridhara, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Identifying word relations in software: A comparative study of semantic similarity tools. In *Proceedings of the* 16^{th} *International Conference on Program Comprehension (ICPC'08)*, pages 123–132. IEEE Computer Society, 2008.

[51] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, March 2007.

[52] Gregory Butler, Peter Grogono, Rajjan Shinghal, and Indra Tjandra. Retrieving information from data flow diagrams. In *Proceedings of* 2^{nd} *Working Conference on Reverse Engineering (WCRE'95)*, pages 84–93. IEEE Computer Society, 1995.

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

[54] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? A study of identifiers. In *Proceedings of* 14^{th} *International Conference on Program Comprehension (ICPC'06)*, pages 3–12. IEEE Computer Society, 2006.

[55] Giuliano Antoniol and Yann-Gaël Guéhéneuc. Feature identification: A novel approach and a case study. In Tibor Gyimóthy and Vaclav Rajlich, editors, *Proceedings of the* 21^{st} *International Conference on Software Maintenance (ICSM'05)*, pages 357–366. IEEE Computer Society, September 2005.

[56] Meghan Revelle. Supporting feature-level software maintenance. In *Proceedings of the* 16^{th} *Working Conference on Reverse Engineering (WCRE'09)*, pages 287–290. IEEE Computer Society, 2009.

[57] Scott Grant, James R. Cordy, and David B. Skillicorn. Automated concept location using independent component analysis. In *Proceedings of the* 15^{th} *Working Conference on Reverse Engineering (WCRE'08)*, pages 138–142. IEEE Computer Society, 2008.

[58] William J. Brown, Hays W. McCormick, and Scott W. Thomas. *AntiPatterns and Patterns in Software Configuration Management*. John Wiley & Sons, 1999.

[59] William J. Brown, Hays W. "Skip" McCormick III, and Scott W. Thomas. *AntiPatterns in Project Management*. John Wiley & Sons, 2000.

[60] Phillip A. Laplante and Colin J. Neill. *AntiPaterns: Identification, Refactoring, and Management*. Auerbach Press, 2006.

[61] Martin L. Shoemaker. *Requirements Patterns and Antipatterns: Best (and Worst) Practices for Defining Your Requirements*. Addison-Wesley Professional, 2007.

[62] Bill Dudney, Stephen Asbury, Joseph K. Krozak, and Kevin Wittkopf. *J2EE Antipatterns*. Wiley Publishing, Inc., 2003.

[63] Bill Karwin. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*. The Pragmatic Bookshelf, 2010.

[64] Bruce A. Tate. *Bitter Java*. Manning Publications Co., 2002.

[65] Bruce Tate, Mike Clark, Bob Lee, and Patrick Linskey. *Bitter EJB*. Manning Publications Co., 2003.

[66] Mariano Ceccato, Nadzeya Kiyavitskaya, Nicola Zeni, Luisa Mich, and Daniel M. Berry. Ambiguity identification and measurement in natural language texts. Technical Report DIT-04-111, Department of Information and Communication Technology, University of Trento, December 2004.

[67] Alistair Willis, Francis Chantree, and Anne De Roeck. Automatic identification of nocuous ambiguity. *Research on Language and Computation*, 6(3-4):355–374, December 2008.

[68] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the* 20^{th} *International Conference on Software Maintenance (ICSM'04)*, pages 350–359. IEEE Computer Society, 2004.

[69] Matthew James Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. In *Proceedings of the* 11^{th} *International Software Metrics Symposium (METRICS'05)*. IEEE Computer Society, 2005.

[70] Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, Laurence Duchien, and Alban Tiberghien. From a domain analysis to the specification and detection of code and design smells. *Formal Aspects of Computing (FAC'10)*, 22(3-4):345–361, 2010.

[71] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison Wesley, 1999.

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