Mining the Lexicon Used by Programmers during Sofware Evolution

Giuliano Antoniol[1], Yann-Gaël Guéhéneuc[2],
Ettore Merlo[1], and Paolo Tonella[3]
[1]Soccer Lab., DGI, École Polytechnique de Montréal, Canada
[2]GEODES Lab., DIRO, University of Montreal, Canada
[3]Fondazione Bruno Kessler—IRST, Trento, Italy
antoniol@ieee.org, guehene@iro.umontreal.ca,
ettore.merlo@polymtl.ca, and tonella@itc.it

Abstract

Identifiers represent an important source of information for programmers understanding and maintaining a system. Self-documenting identifiers reduce the time and effort necessary to obtain the level of understanding appropriate for the task at hand. While the role of the lexicon in program comprehension has long been recognized, only a few works have studied the quality and enhancement of the identifiers and no works have studied the evolution of the lexicon. In this paper, we characterize the evolution of program identifiers in terms of stability metrics and occurrences of renaming. We assess whether an evolution process similar to the one occurring for the program structure exists for identifiers. We report data and results about the evolution of three large systems, for which several releases are available. We have found evidence that the evolution of the lexicon is more limited and constrained than the evolution of the structure. We argue that the different evolution results from several factors including the lack of advanced tool support for lexicon construction, documentation, and evolution.

1 Introduction

Identifiers play a central role in software maintenance because, in a very concise way, they convey clues on the semantics of the software systems, *i.e.*, the semantics of the entities that they label. Although not always recognized by programmers [1, 21], the relevance of identifiers in the program comprehension process has been investigated for a long time [1, 3, 5, 22]. Identifier structure and quality have also been investigated [2, 4, 11], because programmers resort to identifiers to quickly acquire knowledge about program entities and their mapping into domain entities [16, 18]. "Good" identifiers save programmers from reading the entire code segment associated with an entity when attempting to understand the role of the entity in the whole system and its relevance to the software maintenance task at hand.

However, to the best of our knowledge, no previous study attempted to study the evolution of the lexicon of identifiers over time, considering, in particular, their stability over different releases. The lexicon plays a central role during program comprehension insomuch that part of the training of new personnel focuses on its acquisition [20].

Refactoring has been advocated as fundamental to periodically improve the internal structure of a system to facilitate its future evolution (*preventive* maintenance [10]). The investigated *hypothesis* of this work is that a similar evolution process may exist for identifiers, *i.e.*, identifiers may be periodically improved, in particular since the advent of identifier-related refactoring such as *renaming* [8]. We formulate the following research questions to test our general hypothesis about the evolution of identifiers:

- **RQ1**: How does the stability of the lexicon of identifiers compare to the stability of the program structure as the program evolves?
- **RQ2**: What is the frequency of changes to program entities (in particular renaming) due to identifier refactoring?

RQ1 concerns the software evolution process. The evolution of the software structure and size has been already investigated [12], while the evolution of the lexicon less so. We compare lexical vs. structural stabilities during evolution to assess whether they obey similar rules and whether they evolve alike. Finally, from available data, we infer rules to interpret the co-evolution of identifiers and structure.

RQ2 focuses on the quality of the lexicon and its improvement and implicitly on tool support for the lexicon evolution process. It is difficult to measure objectively the quality of identifiers, so we make the hypothesis that one of the possible reasons for a change to the lexicon is to improve the quality of its identifiers.

These two research questions are relevant because during maintenance the construction of a mental model of (part of) the system and the reconstruction of traceability links with other artifacts (requirements, test cases...) represent major sources of difficulties that are strongly affected by the quality of identifiers. The maintainers' productivity depends on the quality of identifier. As a consequence, knowledge about the evolution of the lexicon has implications for the software development and maintenance process, for the training of novices, and for the design of support tools. Moreover, the lexicon of large systems may be substantial, for example we identify in Eclipse more than 124000 unique identifiers, as detailed in Section 3.

We address the two research questions by analyzing the history of three large open-source systems that have evolved for a long time and for which a number of successive versions is available. For each system, first we consider and locate program entities and their identifiers at different levels of granularity, namely at file, class, and function levels; then we extract their lexicon by segmenting identifiers and projecting them onto natural language words. This phase is followed by a normalization of the words performed through stemming. Finally, we study the evolution of the systems by means of lexical and structural stability metrics.

The primary contributions of this paper can be summarized as follows:

- A novel and formal characterization of program lexicon and program structure stability.
- Evolution plots and statistical tests of stability metric results computed from three real world large software projects.
- Rules to qualify the evolution of a program according to the changes to its lexicon and to relate the structural and lexical stabilities with the system maturity and evolution phases.

The remaining of this paper is organized as follows: Section 2 describes the metrics used to measure the lexical and structural stabilities, based on a data model spanning multiple granularity levels. Section 3 presents and discusses experimental results. Related work and conclusion follow and end the paper.

2 Data Mining and Analysis

This section introduce the model, metrics, and analyses that are later performed on three systems.

2.1 Data Model

Objects of our analysis are software systems, comprising their *structures* and *lexica*. Lexica are described by the words in the identifiers and in the comments, while program structures are described by program organizations and control flows. Lexica and structures vary depending on the granularity at which we consider the program: systems, files, classes, methods, functions, global variables, or statements.

At any level of granularity, we model an object of study as a set of *container entities* and *contained entities*. When the evolution of a software system is analyzed, program entities are identified in different releases either by *name* or by *structural similarity*. If two entities (*e.g.*, two functions) have the same name in two successive releases, we assume that they are the same entity that has evolved from one release to the next. If an entity is not found in a previous release by name, it is either a new entity or a renamed entity. To decide between these two possibilities, following an approach similar to Antoniol et al.'s [2], we apply two constraints: (1) There exists an entity in the previous release that cannot be matched by name with any entity in the current release; (2) There exists an entity in the previous release that is structurally similar to an entity in the current release, with similarity above a given threshold. If an entity satisfies both constraints, then we assume that the entity was *renamed* between previous and current releases.

At different levels of granularity, we consider different data for the structure and the lexicon. We adopt the data model depicted in Figure 1. A system or a directory contains files. A file contains classes, functions, or global variables (assuming a mixed procedural and object-oriented programming style as common in C++ systems). A class contains members (attributes and methods) and sub-type relationships with other classes or interfaces.

The structure of a function is represented by a vector of metric values computed on the function body. The lexicon is obtained from the function name and from the identifiers used in the function body and comments.

Attributes and super-types can only be present or absent, their associated structure being arrays containing a single boolean value. The lexicon of an attribute is given by the words in its name. The computation of words out of identifiers requires a normalization step,

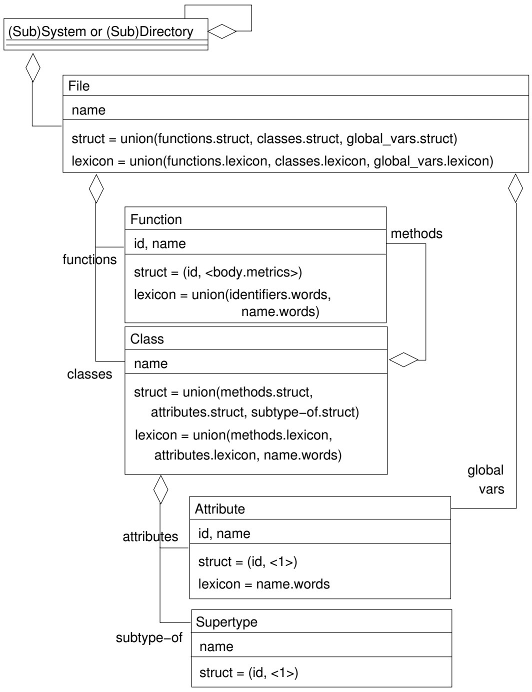

(Sub)System or (Sub)Directory
File
name
struct = union(functions.struct, classes.struct, global_vars.struct)
lexicon = union(functions.lexicon, classes.lexicon, global_vars.lexicon)
functions
Function
id, name
struct = (id, <body.metrics>)
lexicon = union(identifiers.words, name.words)
methods
classes
Class
name
struct = union(methods.struct, attributes.struct, subtype−of.struct)
lexicon = union(methods.lexicon, attributes.lexicon, name.words)
global vars
attributes
Attribute
id, name
struct = (id, <1>)
lexicon = name.words
subtype−of
Supertype
name
struct = (id, <1>)

Figure 1. Data model consisting of container entities and contained entities.

achieved with the Porter stemming algorithm [19].

The class structure is recursively defined using method structure, which is identical to that of functions, attribute, and sub-type relationship structures. The lexicon of a class is obtained from the method and attribute lexicon. Similarly, entities at higher levels of granularity, such as files and directories or subsystems, have a structure and a lexicon that are given by the union of the structure vectors and of the lexica of the contained entities.

The computation of structural data for the function body resorts to software metrics, such as cyclomatic complexity and size (number of uncommented lines of code). In particular, we used the list of structural metrics described in the work by Mayrand et al. on clone detection [15].

2.2 Stability Metrics

We propose two novel stability metrics, which capture the degree of stability of the structure and of the lexicon for a given programming entity, to address **RQ1**. For **RQ2**, we recognize occurrences of renaming by looking for containers that cannot be matched by lexicon in a previous release but that can be matched by structural similarity with an unmatched container in the current release. These stability metrics are generalizations of previous structural metrics [7].

Given two releases i and j, a granularity level G, and a program entity, let us indicate by $e_{i,G}$ and $e_{j,G}$ the two respective releases of the entity which can be either matched by lexicon or by structure. If the entity is not a container, it can be either a function, or an attribute, or a super-type and its structure is a vector of metric values. It is also possible to represent its lexicon using a term-frequency vector where each component is associated with a word and is equal to the number of occurrences of the word in the lexicon. Let us indicate by $vect(e_{i,G})$ any of the two vectors for the structural or lexical stability metrics. We compare the stability of $e_{i,G}$ with $e_{j,G}$ by computing the cosine between the respective vectors:

$$stab(e_{i,G},\ e_{j,G}) \ = \begin{cases} 0 & \begin{array}{l} if\ (vect(e_{i,G}) \ = \ \vec{\mathbf{0}}) \vee \\ \quad (vect(e_{j,G}) \ = \vec{\mathbf{0}})\ else \end{array} \\ \dfrac{\langle vect(e_{i,G}),\ vect(e_{j,G}) \rangle}{\mid vect(e_{i,G}) \mid \cdot \mid vect(e_{j,G}) \mid} \end{cases}$$

Thus, the lexical stability will be 0 when $e_{i,G}$ and $e_{j,G}$ have orthogonal structure or lexicon, while it will be 1 when the entities in the two releases have proportional vectors. If either $vect(e_{i,G})$ or $vect(e_{j,G})$ is $\vec{\mathbf{0}}$ (*e.g.*, the vector of a function missing in version i or j), stability is 0 by definition. We resort to co-linearity of structural information to identify renaming, by using a threshold $T = 1$ and by verifying that $stab_{struct}(e_{i,G}, e_{j,G}) \geq T$: entities are considered to structurally match each other if the cosine of the respective metrics vectors is equal to 1. This threshold value does not guarantee that the two entities have the same lexicon or structure but that the representations are co-linear, i.e., they have proportional metric values.

When the entity is a container, its structural and lexical stabilities are computed as the average of the corresponding stabilities of the contained entities. The lexical or structural stability over a sequence of releases of a container is defined as the sequence of the stability values of the contained entities at consecutive releases $i-1$, i, i.e., average stability of successive pairs $e_{i-1,G}$, $e_{i,G}$.

2.3 Renaming

The relative number of renaming occurring at version i is defined by considering all entities $e_{i,G}$ at a given granularity level G (*e.g.*, functions) and determining those which existed in the previous release but were named differently. As already stated, we characterize a renaming using structural stability (and a threshold $T = 1$) computed for entities that cannot be matched by name. The fraction of renamed entities is then defined as:

$$ren(i, G) = \frac{|DN(e_{i,G}, i-1)|}{|e_{i,G}|}$$

with:

- $e_{i,G}$: entities at granularity G existing in release i.
- $DN(e_{i,G}, j)$: subset of $e_{i,G}$ containing all entities having a different name in (previous) version $j(= i-1)$. Formally, an entity e belongs to $DN(e_{i,G}, j)$ *iff* it satisfies the two following conditions: $e \in DN(e_{i,G}, j) \Leftrightarrow$
  - $\nexists a \in e_{j,G} \mid e.name = a.name$; and
  - $\exists a \in e_{j,G}, \forall e' \in e_{i,G} \mid e'.name \neq a.name \wedge stab_{structure}(e, a) \geq T$

for a given threshold T.

2.4 Stability Analysis

Qualitatively, we visually address **RQ1** by inspecting the plots that represent structural and lexical stabilities over the releases of a system under study. We also quantitatively address **RQ1** by means of statistical tests. The null hypothesis is that there is no statistically-significant difference between the probability distribution of the lexical stability compared to that of the structural stability. Our alternative hypothesis is that structural and lexical stability follow different rules and are characterized by significantly different median values when considering the evolution of a program over time. We use the non-parametric Wilcoxon paired test [9].

RQ2 requires counting the absolute number of renaming occurrences $DN(e_{i,G}, j)$, which can be found for the entities considered at the granularity level G, and comparing this number with the number of entities kept unchanged from the previous release. In our experiments, although we use a threshold value $T = 1$, we nonetheless obtain false positives for small-size entities. For example, small methods that cannot be matched by lexicon are very likely to have similar metric values (similar cyclomatic complexity, similar size, and so on). We remove these false positives by adding an extra constraint on the minimum size for the entities. For methods, we set the minimum at 10 lines of code. Renaming involving entities, the sizes of which are below 10, are skipped (false negatives are possible). By applying this extra constraint in our experiments, the reported occurrences of renaming have been manually verified to be improved.

System	Language	Size	Versions	Identifiers
Eclipse	Java	2.9 MLOC	19	124187
Mozilla	C++	4.4 MLOC	24	55244
CERN/Alice	C++	0.825 MLOC	13	9002

Table 1. Features of the analyzed systems. Size was computed on the latest release.

3 Experimental Results

We now apply the previously-defined analyses on three large software systems.

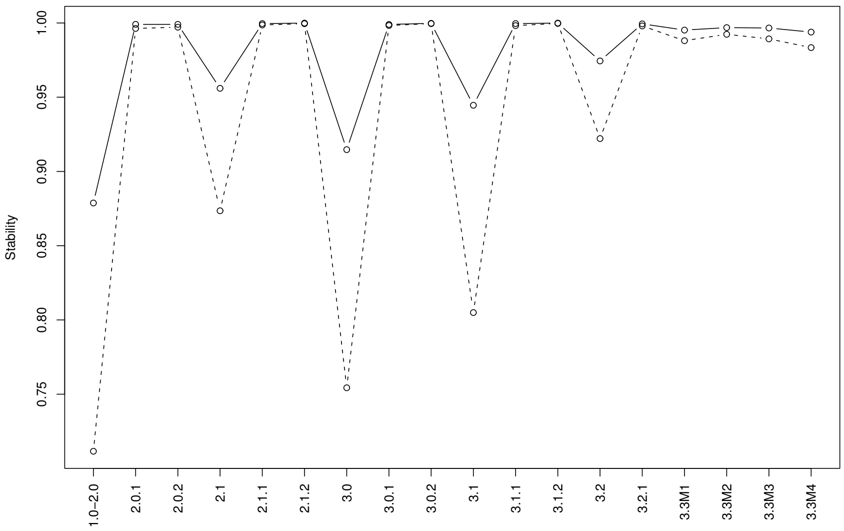
3.1 Subject Systems

As subject systems, we choose *Eclipse*, *Mozilla*, and *Alice*. *Eclipse* is a widely used integrated development environment, supporting programming in Java and other languages through a rich set of functions and graphical facilities. It is a large size program written in Java. We analyzed 19 versions. *Mozilla* is one of the most popular Web browsers. Written in C++, it represents also a large code base, of which 24 releases have been considered in this work. Releases were selected according to the Mozilla road-map[1] to ensure the presence of the main trunk and consistency between release time and release tag, i.e., a higher tag correspond to a more recent release. The third application was provided by the *Alice* experiment currently under construction at CERN (the European Organization for Nuclear Research). It is also a large C++ system, available in 13 versions. It is devoted to the reconstruction, simulation, and analysis of particle trajectories gathered from high-energy physics experiments run on the large hadron collider in Geneva. Table 1 shows data related to the subject systems. We choose three large and different subject systems to decrease possible external factors influencing our study, in particular the context of development (industry-backed open-source, open-source, academics), the age of the system (5 years, more than 15 years, and a couple of years), and the domain (development environment, Web browser, data analysis).

3.2 Stability Plots

The granularity chosen for the stability analysis is the class level ($G = Class$). Figure 2 shows the average lexical stability (solid line) and the average structural stability (dashed line), computed for the available versions of Eclipse. The first point in the diagram rep-

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



1.0-2.0	2.0.1	2.0.2	2.1	2.1.1	2.1.2	3.0	3.0.1	3.0.2
< 2.2e-16	4.377e-08	8.505e-09	5.551e-15	2.942e-11	8.475e-10	0.00662	1.477e-11	3.498e-10
3.1	3.1.1	3.1.2	3.2	3.2.1	3.3M1	3.3M2	3.3M3	3.3M4
< 2.2e-16	0.0004912	0.001291	6.457e-13	1.230e-07	< 2.2e-16	7.527e-14	3.171e-13	3.109e-15

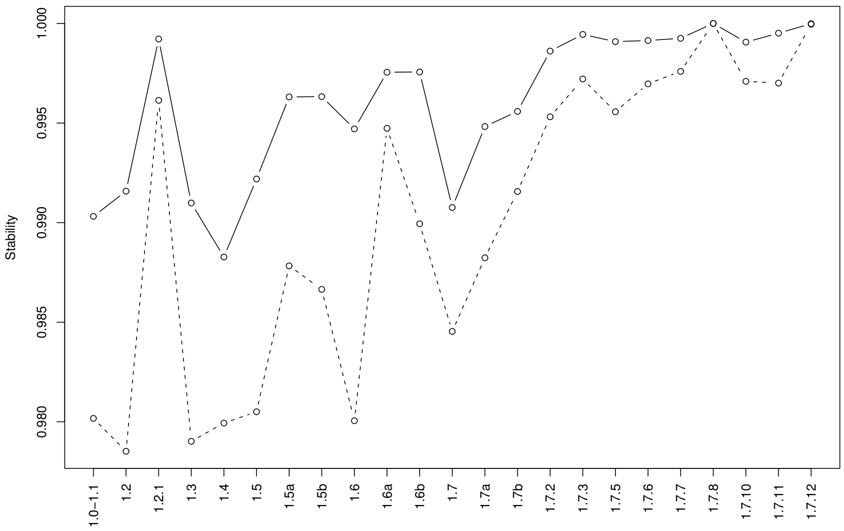
Figure 2. Lexical (solid line) and structural (dashed line) stability for Eclipse. Below the plot, probability that lexical and structural stability data are generated by the same probability distribution, according to the Wilcoxon non-parametric test.

resents the average stability of version 2.0 compared to 1.0. The next point represents the stability of the version indicated on the horizontal axis, compared to the previous version. For example, the second point is the stability of version 2.0.1 compared to 2.0; the last point is the stability of version 3.3M4 vs. 3.3M3.

The evolution of Eclipse highlights the existence of releases where major changes occurred, resulting in lower stabilities, and of releases where minor changes happened and stability values are close to 1. For example, version 2.0 compared to 1.0 shows high structural and lexical instabilities, resulting from major code changes involving both the structure and the lexicon of the code (structural stability is 0.71, lexical stability is 0.87). Other versions associated with major instabilities are: 2.1, 3.0, 3.1, 3.2. These observations are in accordance with the common software development process adopted with open-source systems, where major versions implement large number of changes, while minor versions contain mostly bug fixes.

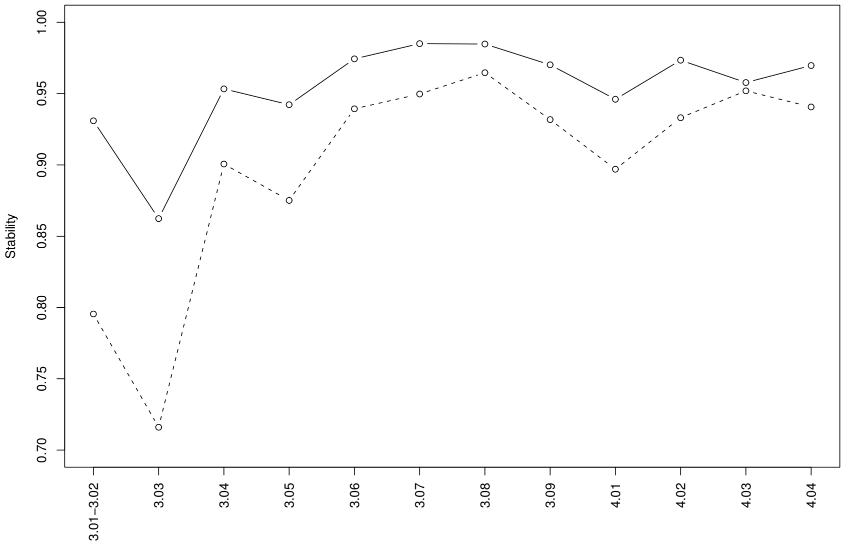
Looking at the full plot depicted in Figure 2, we notice a decrease in the number and amplitudes of the instabilities. Even if only major changes are considered, it is apparent that the program is moving toward maturity (compare structural stability at versions 3.0, 3.1, 3.2). However, such a trend is not necessarily monotonic, as apparent from the stability of version 3.0 compared to 2.1.

At the bottom of Figure 2, we show the results of the Wilcoxon test. The reported values are the probabilities that the values observed for the structural and lexical stabilities come from the same statistical distribution (p-value). It is commonly accepted that a p-value below 0.05 allows rejecting the null hypothesis with strong evidence. For every considered release of Eclipse, we reject the null hypothesis with strong evidence and we conclude that structural and lexical stabilities do not have the same distribution. Although the temporal evolution looks parallel (high structural (in)stability is associated with high lexical (in)stability), the two types of stability follow different rules and come from different maintenance processes. Even in cases where the average stability values are very close to each other (*e.g.*, version 3.0.2), the underlying distributions are statistically different, and this observation is true for all versions, including those involving only minor changes. Finally, we observe that the lexical stability is always higher than the structural



1.0-1.1 < 2.2e-16	1.2 < 2.2e-16	1.2.1 8.152e-06	1.3 < 2.2e-16	1.4 < 2.2e-16	1.5 < 2.2e-16	1.5a < 2.2e-16	1.5b < 2.2e-16	1.6 < 2.2e-16
1.6a < 2.2e-16	1.6b < 2.2e-16	1.7 < 2.2e-16	1.7a < 2.2e-16	1.7b < 2.2e-16	1.7.2 < 2.2e-16	1.7.3 6.789e-11	1.7.5 < 2.2e-16	1.7.6 < 2.2e-16
1.7.7 1.279e-07	1.7.8 < 2.2e-16	1.7.10 < 2.2e-16	1.7.11 1.439e-06	1.7.12 4.696e-13				

Figure 3. Lexical (solid line) and structural (dashed line) stability for Mozilla. Below the plot, probability that lexical and structural stability data are generated by the same probability distribution, according to the Wilcoxon non-parametric test.



3.01-3.02 5.124e-06	3.03 0.03817	3.04 0.0003216	3.05 0.005056	3.06 4.158e-06	3.07 3.329e-14	3.08 0.001288	3.09 1.121e-08	4.01 1.535e-09
4.02 < 2.2e-16	4.03 9.227e-15	4.04 2.709e-10						

Figure 4. Lexical (solid line) and structural (dashed line) stability for Alice. Below the plot, probability that lexical and structural stability data are generated by the same probability distribution, according to the Wilcoxon non-parametric test.

stability, indicating that programmers tend to modify the structure more than the lexicon. Even when the system undergoes major changes, programmers prefer to keep the lexicon stable, while they tolerate a higher number of structural changes.

In the stability plots reported for Mozilla (in Figure 3) and Alice (in Figure 4), we observe similar patterns as for Eclipse. The first considered version of Mozilla, 1.1, is already quite stable in comparison to the previous version, 1.0, with a structural stability of 0.98 and a lexical stability of 0.99. These values are substantially higher than those reported for Eclipse for the major releases (*e.g.*, 1.0–2.0). One can interpret this by considering that Mozilla derives from a previous system, Netscape. Thus, its code base has already achieved a relatively high maturity during the evolution of its previous incarnation. Hence, the 24 analyzed versions are already more mature and stable. Nevertheless, some versions are more instable than others, *i.e.*, major releases 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, and 1.7, and an overall trend towards higher stability is quite apparent. Lexical stability is always higher than structural stability, similarly to Eclipse, and the difference between the distributions is statistically significant.

The evolution of Alice is characterized by stability levels similar to Eclipse, with a trend towards higher stability, indicating that this system is also approaching maturity. A few instabilities departing from the trend can be noticed for example at versions 3.03, 3.05, 4.01. This observation suggests that the development of Alice may not follow a recognized software development process, where minor releases should implement only minor changes. The occurrence of major changes with versions 3.03, 3.05, 4.01 has been confirmed by the programmers working on Alice. Similarly to the other two systems, the plot shows a correlation between structural and lexical stabilities, with the latter always above the former. As this system evolve, major changes in the lexicon corresponded to major changes in the structure, but the lexicon was kept more stable than the structure, both for major and minor releases.

3.3 Renaming

Renaming analysis was performed at the method and function level (i.e., $G = method$), to increase the probability of observing interesting renaming patterns. Tables 2 and 3 show the numbers of renaming occurrences (last column) for each pair of consecutive versions. For comparison purposes, we report also the number of unchanged methods and of newly added methods. In all three systems, renamed entities represent a very small fraction of the total number of entities in each release. Even when compared to added entities, renamed entities represent only a small fraction. Mozilla and Alice are somehow extreme cases, with no renaming detected in any of the versions considered for these programs (false negatives are possible). This observation does not contradict our previous results on the lexical stabilities, the added entities being responsible for the instabilities.

Version	Unchanged	Added	Renamed
1.0–2.0	26744	39233	10
2.0.1	65875	290	1
2.0.2	66118	276	2
2.1	60450	28138	19
2.1.1	88534	138	1
2.1.2	88662	51	0
3.0	58125	52577	40
3.0.1	110692	226	0
3.0.2	110880	52	0
3.1	90874	40217	4
3.1.1	130951	326	1
3.1.2	131268	41	0
3.2	121128	32087	29
3.2.1	153110	542	1
3.3M1	151799	3211	10
3.3M2	153885	3533	5
3.3M3	155623	3990	8
3.3M4	156977	5206	11
Average	106760	11674	7

Table 2. Unchanged, added and renamed entities (methods) for Eclipse.

Considering the results shown in Tables 2 and 3, we notice the impact of the programming language on the programmers' tendency to rename entities. In Eclipse (a Java system), we observe a higher number of renaming occurrences than in the other two programs, written in C++, maybe because of the refactoring available in current Java development environments. Also, instability of the lexicon in Eclipse may result from the instability of its domain, with new concepts being added in major releases to provide new features, while the domains of the Web browser Mozilla and of the simulator Alice are better defined and bound. Overall, renaming represents a small fraction of the changes, thus confirming the high stability of the lexicon during the evolution of these software systems.

3.4 Discussion

With the above analyses, we are in a position to answer the research questions **RQ1** and **RQ2** and suggest rules on the evolution of structure and lexicon.

To answer **RQ1**, we compared the structural and lexical stabilities of three large programs during their evolution. The observation of the stability plots show a general trend toward higher stability levels. Both structural and lexical stabilities tend to increase over

Version	**Unchanged**	**Added**	**Renamed**
1.0–1.1	48586	1845	0
1.2	49293	2268	0
1.2.1	51532	9	0
1.3	49821	2421	0
1.4	49899	2494	0
1.5	51240	1597	0
1.5a	52043	555	0
1.5b	52051	777	0
1.6	51692	1492	0
1.6a	52368	559	0
1.6b	52365	517	0
1.7	51068	2615	0
1.7a	52604	935	0
1.7b	52672	804	0
1.7.2	53364	282	0
1.7.3	53003	28	0
1.7.5	52963	194	0
1.7.6	53137	42	0
1.7.7	53157	17	0
1.7.8	53174	0	0
1.7.10	53146	62	0
1.7.11	53179	49	0
1.7.12	53228	2	0
Average	51981	850	0

Version	**Unchanged**	**Added**	**Renamed**
3.01–3.02	1062	675	0
3.03	951	3050	0
3.04	3641	1085	0
3.05	4437	1139	0
3.06	5307	1072	0
3.07	6239	1319	0
3.08	7410	517	0
3.09	7748	2853	0
4.01	8539	3326	0
4.02	10791	2988	0
4.03	11856	2236	0
4.04	12851	2487	0
Average	6736	1895	0

Table 3. Unchanged, added and renamed entities (methods) for Mozilla (top) and for Alice (bottom).

time, with more mature projects (*i.e.*, Mozilla) having higher stability values. This comparison shows that the lexical stability is always higher than the structural stability: the lexicon is more stable during the evolution of a system. We explain these findings in that the lexicon forms an essential corner stone of the programmers' understanding and mental models. Based on the performed comparisons, structural and lexical stabilities evolve following a similar pattern, but yet they derive from different underlying statistical distributions. Thus, the lexicon evolves independently of the structure. Again, we explain these findings by the importance of the lexicon and the care with which programmers modify the lexicon independently of the structure of the system, to avoid having to change dramatically their own mental models. We derive a first rule: *The lexicon is more stable than the structure of a software system.*

To answer **RQ2**, we computed the number of renaming occurrences across different versions for each system. This computation shows that very few renaming occurrences took place during the evolution of the systems, thus confirming the importance of the lexicon. We conclude that changes to the lexicon are more rare, may be more critical to the correspondence between the system and its domain, and should be more carefully evaluated and analyzed. Also, the small number of renaming occurrences means that programmers are reluctant to modify their mental models by modifying the lexicon once they have developed such a model. On one hand, this observation suggests that the lexicon tends to consolidate and stabilize over time, requiring little adjustments. On the other hand, this observation indicates an inflexibility of the systems and of the programmers who cannot absorb lexical changes. Poor identifiers might become part of the application lexicon. This small number of renaming occurrences could also be the result of a careful domain analysis. We derive a second rule: *Changes to the lexicon are rare during the evolution of a software system.*

In addition to these two rules, we observe that all three systems evolve toward stability, although occasionally the trend is reverted at major releases. We explain this observation by the different nature of changes in major releases and in minor releases. Major releases often result from major efforts to improve or change the functionalities and quality of the system and thus its structure and lexicon. The amplitude of the instabilities tends to decrease over time. These observations lead us to conclude with a more general rule of software evolution: *A system becoming more stable has both structural and lexical stabilities converging toward 1.*

The statistical analyses of the obtained results indicate that lexical and structural stability values have different distributions. This observation underpins a different change process for these two aspects of the code. Moreover, we can relate the absence of renaming in C++ systems, compared to the occurrence of renaming in the analyzed Java system, to the different development environments available for these two languages. Java development environments offer advanced refactoring support, including entity renaming. Thus, this absence may indicate a strong need for tools that support the evolution of the lexicon in C++ (in particular, given the high number of unique identifiers in Mozilla, 55244). Moreover, current development environments should be enriched with tools devoted to the understanding, cross-referencing, documenting, and refactoring of the lexicon. Given the small fraction of renaming, we argue that such a support is still too limited and could be probably enhanced in several ways: glossary construction tools should be added to ease understanding; cross-referencing tools could act directly on the words composing the identifiers and include stemming to prevent lexical variations; abbreviation expansion tools could ease reading and understanding. Finally, documentation about the lexicon should be provided with reference to a domain ontology to enhance renaming with recommendations based on the existing terms and their relations.

3.5 Threats to validity

The main threats to the validity of this study are related to its *external* validity, *i.e.*, to the generalization of the results to other systems. We considered three large systems and thus replications with more systems is necessary to draw definitive conclusions that hold generally for systems of different size, programming language, or application domain. Moreover, the choice of the systems, in particular Eclipse, may have impacted the study. Eclipse is now widely used both in the open-source community and in industry and, thus, its developers must be careful when renaming public identifiers.

To minimize the *conclusion* threats to validity, associated with the possibility of relating different data distributions to different stability measures, we used a non-parametric statistical test that makes no assumption on the distribution of the observed values.

With regards to the *construct* threats to validity, concerning the measures taken and the metrics adopted in the experiments, we use a generic metrics for stability, *i.e.*, the cosine similarity. Different metrics (components of the vectors) may highlight different aspects of stability, possibly leading to different conclusions.

Finally, we could not identify any specific *internal* validity threat, *i.e.*, threat associated with system-specific factors affecting the lexical or structural stability of the selected systems.

4 Related Work

Previous works on program identifiers focused on their role in support to program understanding [1, 3, 5, 22]. Other work [2, 4, 11, 17] attempted to investigate the information carried by the words composing an identifier, their syntactic structure and quality.

The existence of so-called "hard words" that encode core concepts into identifiers was the main outcome of the study by Anquetil et al. [1]. An in-depth analysis of the internal identifier structure was conducted by Caprile et al. [3], while guidelines for the production of high quality identifiers have been provided by Deißenböck et al. [5]. Methods related to identifier refactoring were proposed by Caprile et al. [4] and Demeyer et al. [6].

Some studies [2, 13, 14] report how identifiers can be used to recover traceability links. Merlo et al. [17] analyzed informal information including identifiers and comments in programs.

The role of identifiers in mapping the domain model into the program model (*i.e.*, programming entities) was studied by Takang et al. [22]. A crucial role is recognized to be played by the program lexicon and the coding standards in the so-called naturalization process of software immigrants [20]. Lawrie et al. focused on the quality of the identifiers [11]. The results of the empirical study they conducted with over 100 programmers indicate that full words as well as recognizable abbreviations lead to better comprehension.

Assuming a major role of identifiers in program comprehension, the novelty of the present work compared to the existing literature is the analysis of the evolution of the lexicon over time, and its comparison with structural evolution.

5 Conclusion

We analyzed the versions of three large systems, namely Mozilla and Alice written in C++ and Eclipse written in Java, to assess the stabilities of their structures and lexica. Observations indicate that these systems evolved towards higher stability, but occasionally the trend is reverted, when instabilities are introduced to accommodate changes. We derive three rules from these observations: the lexicon is more stable than the structure of a software system; changes to the lexicon are rare during the evolution of a software system; and conclude that: a system becoming more stable has structural and lexical stabilities converging toward 1, *i.e.*, developers focus on adding new functionalities and correcting bugs rather than refactoring structure and lexicon.

We argue that the limited ability to evolve the lexicon of a software system is due to the cost of building a mental model of the system through its lexicon for programmers and the lack of support offered by development environments. These observations are confirmed by the higher number of occurrences of renaming for the Java system compared to the C++ systems. Actually, most Java development environments offer the *renaming* refactoring. We conclude that novel support tools for the lexicon are needed, such as glossary construction tools, cross-referencing tools, abbreviation expansion tools, and application domain ontologies.

The lexicon of a program represents a substantial investment for a software company, hence its value should be preserved and increased over time, to take full advantage of its beneficial effects on program comprehension. We will devote our future work to the replication of the experiments on other systems, considering other programming languages and styles, as well as different entity granularity. We will also complement the stability metrics with quality metrics of the identifiers.

Acknowledgments

The authors gratefully thank the programmers of the Alice project. Giuliano Antoniol was partially supported by NSERC, Canada Research Chair in Software Change and Evolution. Yann-Gaël Guéhéneuc and Ettore Merlo were partially supported by an NSERC Discovery Grant.

References

[1] N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of CASCON*, pages 213–222, December 1998.

[2] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28:970–983, Oct 2002.

[3] B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proc. of the Working Conference on Reverse Engineering (WCRE)*, pages 112–122, Atlanta, Georgia, USA, October 1999.

[4] B. Caprile and P. Tonella. Restructuring program identifier names. In *Proc. of the International Conference on Software Maintenance (ICSM)*, pages 97–107, 2000.

[5] F. Deißenböck and M. Pizka. Concise and consistent naming. In *Proc. of the International Workshop on Program Comprehension (IWPC)*, May 2005.

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

[7] M. O. Elish and D. Rine. Design structural stability metrics and post-release defect density: An empirical study. In *COMPSAC '06: Proceedings of the 30th Annual International Computer Software and Applications Conference*, pages 1–8, Washington, DC, USA, 2006. IEEE Computer Society.

[8] M. Fowler. *Refactoring: Improving the design of existing code.* Addison-Wesley Publishing Company, Reading, MA, 1999.

[9] M. Hollander and D. A. Wolfe. *Nonparametric statistical inference.* John Wiley & Sons, New York, 1973.

[10] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990 (1991 Corrected Edition).* The Institute of Electrical and Electronics Engineers, Inc., 1994.

[11] D. Lawrie, C. Morrel, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *Proc. of the International Conference on Program Comprehension (ICPC)*, pages 3–12, 2006.

[12] M. M. Lehman. Programs life cycles and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, September 1980.

[13] J. I. Maletic, G. Antoniol, J. Cleland-Huang, and J. H. Hayes. 3rd international workshop on traceability in emerging forms of software engineering (tefse 2005). In *ASE*, page 462, 2005.

[14] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE*, pages 125–137, 2003.

[15] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 244–253, Monterey CA, Nov 1996.

[16] A. V. Mayrhauser and A. M. Vans. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 22(6):424–437, 1996.

[17] E. Merlo, I. McAdam, and R. D. Mori. Feed-forward and recurrent neural networks for source code informal information analysis. *Journal of Software Maintenance*, 15(4):205–244, 2003.

[18] N. Pennington. *Comprehension Strategies in Programming. In: Empirical Studies of Programmers: Second Workshop. G.M. Olsen S. Sheppard S. Soloway eds.* Ablex Publisher Nordwood NJ, Englewood Cliffs, NJ, 1987.

[19] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

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

[21] H. Sneed. Object-oriented cobol recycling. In *Proc. of the Working Conference on Reverse Engineering (WCRE)*, pages 169–178, 1996.

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