Understanding Structural Complexity Evolution: a Quantitative Analysis

Antonio Terceiro, Manoel Mendonça, Christina Chavez
Software Engineering Lab (LES)
Department of Computer Science (DCC)
Federal University of Bahia (UFBA)
{terceiro, mgmendonca, flach}@dcc.ufba.br
Daniela S. Cruzes
Department of Computer and Information Science (IDI)
Norwegian University of Science and Technology (NTNU)
dcruzes@idi.ntnu.no

Abstract—Background: An increase in structural complexity makes the source code of software projects more difficult to understand, and consequently more difficult and expensive to maintain and evolve. Knowing the factors that influence structural complexity helps developers to avoid the effects of higher levels of structural complexity on the maintainability of their projects.

Aims: This paper investigates factors that might influence the evolution of structural complexity.

Method: We analyzed the source code repositories of 5 free/open source software projects, with commits as experimental units. For each commit we measured the structural complexity variation it caused, the experience of the developer who made the commit, the size variation caused by the commit, and the change diffusion of the commit. Commits that increased structural complexity were analyzed separately from commits that decreased structural complexity, since they represent activities of distinct natures.

Results: Change diffusion was the most influential among the factors studied, followed by size variation and developer experience; system growth was not necessarily associated with complexity increase; all the factors we studied influenced at least two projects; different projects were affected by different factors; and the factors that influenced the increase in structural complexity were usually not the same that influenced the decrease.

Conclusions: All the factors explored in this study should be taken into consideration when analysing structural complexity evolution. However, they do not fully explain the structural complexity evolution in the studied projects: this suggests that qualitative studies are needed in order to better understand structural complexity evolution and identify other factors that must be included in future quantitative analysis.

I. INTRODUCTION

Software maintenance and evolution are labor-intensive activities that consume a large part of the overall software lifecycle costs [1]. In this context, the assessment, comprehension and control of factors that impact maintenance and evolution activities are of paramount importance.

One such a factor, the structural complexity of a software system [2], [3], [4], [5], [6], [7], may exert strong influence on the maintenance effort: the higher is the structural complexity of the source code, the greater is the effort to comprehend, change and evolve it [6], [8]. In the specific case of free software\(^1\), projects with higher structural complexity are less likely to attract developers [8], [9]. This situation is specially harmful for projects that do not have corporate sponsors and depend solely on independent and volunteer developers to evolve.

There are empirical studies on the measurement of structural complexity and its evolution over time [6], [10], [11], but they fail to assess and provide empirical evidence about which factors influence the evolution of structural complexity itself. The structural complexity of a project may increase due to different factors, such as the maturity level of the project, growth of its codebase, the type of maintenance (perfective, corrective, evolutionary) the project has been subject to, development practices, past design decisions, and characteristics of the developers working on the project. The knowledge about the factors that impact structural complexity could be useful for developers and project managers to control the growth of structural complexity, or to drive its decrease by means of anti-regressive activities. This would contribute to keep an acceptable level of understandability and changeability and, therefore, to reduce maintenance effort and costs.

Our work aims to reduce this gap by studying different factors that may influence the variation in structural complexity, for instance, developers’ experience within a project, size variation and change diffusion. The choice of developers’ experience within the project as a factor reflects the importance of human factors on software development projects. We expect that developers’ experience influences software design activities, and in special, the evolution of structural complexity. Experience has been used previously to identify experts [12] and to predict defects [13], [14]. System size is an important factor in software evolution studies, and this work is not an exception. Besides, system size if often associated with software complexity [15], [16], [17]. Change diffusion – the number of different elements (subsystems, modules, files etc) involved in a change – represents how wide a change is with respect to a given codebase. Change diffusion has

\[^1\text{a.k.a. “open source software”}.\]
been characterized as a good predictor for defects [13], where the authors argued that “large diffusion indicates that the modularity of the code is not compatible with the change, because several modules have to be touched to implement the change. […] The diffusion of a change reflects the complexity of the implementation […].” Based on their ideas, we expect that change diffusion may have some influence over structural complexity evolution as well.

This paper reports the results of an empirical study performed in order to assess whether and how developers’ experience within a project, size variation and change diffusion influence the evolution of structural complexity. These three factors were studied in the context of historical data from five well-established free software development projects written in different programming languages. Our study can be regarded as a preliminary step towards the systematic identification of factors that can be used in the selection of treatments for taming structural complexity during maintenance activities.

The rest of the paper is organized as follows. Section II introduces basic concepts related to structural complexity, developer experience within the project, size variation and change diffusion. Section III presents the design of our study. Results are presented in Section IV, and discussed in Section V. Threats to validity are presented in Section VI. Section VII presents our closing remarks.

II. BACKGROUND

A. Structural Complexity

The concept of software complexity has been extensively studied in Software Engineering [2], [3], [4], [18], [19], [5], [6], [11], [10], [7], and yet there is no widely accepted approach for measuring it. Since the 70’s, there were several proposals for measuring complexity at the subroutine level [2], [3], [4], McCabe’s cyclomatic complexity [2] eventually became widely used and it’s still in use in more recent work [20], [8], [21].

Software complexity can also be studied at a higher level of abstraction. Researchers have proposed structured approaches for software architecture planning and documentation [18], [19], Design Structure Matrices [5] to represent complex software systems, highlighting dependencies and modules, measures for structural complexity based on inter-module coupling and intra-module cohesion [6], [11], [10], and the use of recursive aggregation to calculate structural complexity of higher levels of abstraction in terms of their own organization and the structural complexity of lower levels of abstraction [7].

Darcy et al. [6] propose that structural complexity should be measured by combining coupling and cohesion metrics together. Coupling and cohesion are complementary concepts: while coupling addresses inter-module relationships, cohesion is concerned with the relationship between the intra-module elements and their organization. The software design practitioners’ literature advocates that modules with low coupling and high cohesion are easier to understand and modify [22], [23].

The combination of coupling and cohesion into a structural complexity metric captures both the complexity created by excessive inter-module dependency and the complexity that results from modules with multiple, unrelated responsibilities. Moreover, coupling and cohesion are not limited to the object-oriented paradigm. Most programming paradigms have one or more concepts that stand for "module" – “classes”, “aspects”, “abstract data types”, or “source files” – for which coupling and cohesion measures can be calculated.

Our work uses Darcy’s metric for structural complexity for two reasons. First, their approach was based on a comprehensive literature review, in which they identified coupling and cohesion as factors that could be used to provide a meaningful measure of structural complexity. Second, they validated the assumption that higher structural complexity is associated with greater maintenance effort by means of a controlled experiment. The authors found out that neither coupling nor lack of cohesion alone could explain the decrease in comprehension performance of the developers; only when coupling and lack of cohesion were considered together as interacting factors, an association with greater maintenance effort could be observed.

A formalization of the metric proposed by Darcy et al. [6] can be given as follows. Given a project \( p \) and its set of modules \( M(p) \), the structural complexity of \( p \) is given by \( SC(p) \), which is defined as follows.

\[
SC(p) = \frac{\sum_{m \in M(p)} CBO(m) \times LCOM4(m)}{|M(p)|}
\]

This structural complexity measure is the average structural complexity across all the modules of the system. When a system grows in size and also has its structural complexity increased, this means that not only the system as a whole is larger and therefore more difficult to understand and modify, but every individual module, on average, is also more difficult to understand and modify.

For measuring cohesion, we use Chidamber and Kemerer’s CBO [24], calculated as the number of modules a given module depends on. For measuring lack of cohesion, we use Hitz and Montazeri’s LCOM4 [25]. The LCOM4 value for a module is the number of connected components of an undirected graph, where the nodes are the module’s subroutines (methods, functions etc.), and the edges indicate that two subroutines use at least one attribute/variable in common, or that one subroutines calls the other. These connected components represent independent parts of a module, and modules that have more than one of them have independent, distinct responsibilities.

Growing in size is inevitable for evolving software in a real-world setting, where there is a constant flow of new user requirements to be satisfied and environment changes to deal with. The challenge of modularization is then being able to grow in size while maintaining complexity under control.
B. Developers’ Experience

Developer’s experience has been covered in past literature in terms of two different dimensions. The first dimension is associated with developers’ previous work experience: developers are characterized in terms of “years of experience”, or in terms of “skill level”, which are represented in ordinal scales such as “low/medium/high” or in simple numeric scales such as “1 to 5”. This dimension of experience has been used, for example, as a predictor for project productivity [26], maintenance effort [6], and software product quality attributes [27]. Software development capability methods such as SPI and CMM have been criticized for not taking previous practical experience into account [28].

A second dimension of developer experience is related to developers’ activity in a given project. This dimension is characterized by using historical project information to provide a measure of how much experience each developer has on the ongoing project. This can be used, for example, to identify experts in certain parts of the project such as individual modules or subsystems [12], to make sure maintenance teams have at least someone knowledgeable in the maintained project [26], and a as predictor variable for defects [13], [14].

The experience of a developer within a project has been measured using commit data recorded in version control repositories, in terms of: (i) number of commits performed by the developer [14]; or (ii) the number of days since the first commit performed by the developer in the project’s version control repository [29].

C. System Size Variation

System size is definitively an attribute that must be observed by software researchers and practitioners. Besides the common sense observation that larger systems are more difficult to deal with, many believe that there is an association between size and complexity. Parnas [15] mentions system size as a factor that makes systems hard to change and evolve. In early software evolution studies, Lehman stated among his laws of software evolution both “Continuous Growth” and “Increasing Complexity” [16]. This association is also suggested in a paper by Jay et al. [17] that found a very strong correlation between system size and cyclostatic complexity across different programming languages, development methodologies and application domains.

Despite being often criticized, lines of code (LOC) remains as the most widely used measure of system size. Approaches for counting lines of code vary, but ignoring blank and comment lines is a common practice. System size variation may also be computed in terms of lines of code by calculating the difference in the LOC count between two versions of a given codebase.

D. Change diffusion

Small changes or changes that involve a small number of modules tend to require less cognitive effort from developers, while larger changes may involve more modules than a developer can easily reason about. Therefore, we expect that changes that span several modules increase the risk of developers modifying existing software design in a way that makes it more complex.

Change diffusion can be measured as the number of subsystems touched, the number of modules touched, or the number of lines of code touched. It has been characterized as a good predictor for software failures [13]. Change diffusion has also been used to construct prediction models based on change entropy, which outperformed models based on previous changes and previous faults for predicting the occurrence of faults in large free software projects [30].

III. Experimental Setup

In this study, our unit of analysis are changes made by developers to the project source code, also known as commits. Not all commits made by developers result in variation of structural complexity. Therefore, we have limited ourselves to examine only commits that increased or decreased the structural complexity of the studied systems.

In light of the three factors presented in Section II, the following research questions guided our study:

• RQ1: How does the developer’s experience within a project influence the increase or decrease in structural complexity caused by his commits?
• RQ2: How does the size variation introduced by commits influence the increase or decrease in structural complexity?
• RQ3: How does the change diffusion introduced by commits influence the increase or decrease in structural complexity?

A. Research Hypotheses

Our first hypotheses relate the three factors to commits that increase structural complexity.

H1: Developer experience influences the increase in structural complexity negatively. We expect commits made by more experienced developers to introduce less structural complexity in the codebase.

H2: Size variation influences the increase in structural complexity positively. Our intuition is that commits that add more code also add more structural complexity.

H3: Change diffusion influences the increase in structural complexity positively. We expect that broad changes touching many files introduce more structural complexity than narrow, localized changes.

The last hypotheses relate the three factors to commits that decrease structural complexity:

H4: Developer experience influences the decrease in structural complexity positively. When developers work in complexity-reducing activities, we expect that more experienced developers remove more structural complexity.

H5: Size variation influences the decrease in structural complexity negatively. In complexity-reducing activities, we expect that removing code should cause larger decrease in structural complexity.
**H₆:** Change diffusion influences the decrease in structural complexity positively. In complexity-reducing activities, we expect broader changes to be large reorganizations that make the code less complex, and therefore commits that touch more files are expected to produce a larger decrease in structural complexity.

**B. Instrumentation and Measurement**

In this study we are interested in the process through which a developer gains experience within a given project, and thus on the second dimension of experience mentioned in Section II-B. We are not interested in making absolute comparisons between developers, but rather to analyze developers’ behavior as they get more experienced in the context of a project they work on. Thus, we do not consider any previous experience the developer may have had, and represent experience in terms of time the developer has spent in that given project. We used the following measures of developer experience within the project:

- **n** – the number of commits the author had made to the project’s source code before the current commit (since he started working in the project).
- **d** – the number of days in which the author of the change had been contributing to the project before the given commit. This is measured by counting the number of days between the first change made by the developer and the date of the current change.

LOC is used for measuring size variation, since it is the most widespread size measure. For change diffusion, we use the number of files that are touched by a commit. They were represented by the following variables:

- **ΔLOC** – the variation in size of the project, measured in lines of code.
- **CF** – changed files, a measure of change diffusion. This variable is measured as the number of files affected (added, changed, removed) by the commit, as recorded by the version control system.

For measuring structural complexity, we use the metric by Darcy et al [6]. Since we are intested in both commits that increase and that decrease structural complexity, we represent structural complexity increase and decrease in their own variables:

- **ΔSC** – structural complexity variation. This is the difference between the structural complexity after the commit and the structural complexity before it. This is an auxiliary variable used to calculate the next two.
- **ΔSCᵢ** – increase in structural complexity. Defined as ΔSC, if ΔSC > 0. Whenever this variable was used in a test, only the data points where structural complexity actually increased in comparison with the previous commit (i.e. ΔSC > 0) were considered.
- **ΔSCᵣ** – decrease in structural complexity. Defined as |ΔSC|, if ΔSC < 0. Whenever this variable was used in a test, only the data points where the structural complexity actually decreased in comparison with the previous commit (i.e. ΔSC < 0) were considered.

![Figure 1](image-url) Variable values for commit at t₃.

Figure 1 describes visually the operational definitions for n, d and ΔSC, exemplifying their calculations for the commit at t₃. Each circle represents a commit in the project’s source code. We have information about its date (written below it), its author (represented by the different shades of gray), and also information about the state of the code after the commit (such as metrics values), represented by an attached tag. The commit at t₃, for example, is the third commit made by the light gray developer, so it has n = 2, which means that before making that commit, the developer had done two other commits. The value of ΔSC for that commit is then calculated by the difference between the current SC value and the SC value at the previous state of the code, represented by the commit in dark gray. ΔLOC is calculated similarly, but omitted in the figure. d is then calculated by the difference between the current time (t₃) and the time of the first commit that developer made to the project (t₀). CF is also not shown, but is obtained by querying the version control system for how many files were touched by each commit.

Commits that increase and decrease structural complexity were analyzed separately, since we believe they represent conceptually different activities. Commits that increase structural complexity represent the addition of new features and the correction of defects, while commits that decrease structural complexity are the result of anti-regressive work such as refactorings and other source code reorganizations.

**C. Analyzed projects**

We have studied historical data from five free software projects. The criteria for the project selection was the following: projects should be written in C, C++ or Java; projects must have had at least 18 months of development history available; the version control repository used by the project must be Git. Git is a distributed version control system (VCS) that stores explicitly the author name of each commit done in the project. Older VCS such as CVS and Subversion only store the login name of the user that added the commit to the VCS repository (committer), which is often not the same person who wrote the code (author). Since Git stores both author and committer data, the authorship information is more trustworthy, what is
projects were selected from GitHub. Descriptive information about the selected projects is presented in Table I. Clojure is a dynamic, functional programming language that is a dialect of Lisp. Its main implementation, also called Clojure, compiles Clojure code to JVM bytecode. Node is a platform for JavaScript programming with event-driven I/O based on the V8 JavaScript engine, mainly used for writing server-side applications in JavaScript. Redis and Voldemort are key-value storage systems, often considered to be in the category of “NoSQL databases”. Zeromq2 is a messaging library used for process communication in distributed applications.

All projects are infrastructure software, i.e. we did not analyze projects targeted at end users. This was not planned, and is probably due to our choice of looking for projects on GitHub: several free software applications targeted at end users, specially those part of umbrella projects like GNOME, KDE and Xfce, are hosted on their own servers.

Some of these projects started very small, such as Clojure with 59 LOC and Node with 617 LOC (column named ELOC0). Others started their version control history with 60% of their final size, such as Zeromq2.

We consider them to be medium-sized projects: their final size (ELOCf) ranges between 11,000 and 35,000 LOC. When we analyze their size in terms of number of modules, they present a high variance. Clojure is the project with the largest number of modules, containing 666 of them with an average module size of approximately 48 LOC/module. Redis is the project with the largest average module size, containing 48 modules with an average of 350 LOC/module.

These projects also had a significantly high number of distinct developers. Node was the project with more developers among them, with 164 developers recorded in its version control database. The one with fewer developers was Redis, with 32 developers. The developers do not have a uniform volume of participation in the projects, though.

Figure 2 shows the evolution of the studied projects in terms of their size in Lines of Code and its average structural complexity. The X axis shows the project time, starting at the first commit recorded in the version control repository and ending at the last commit that was considered in this paper (approximately until February/2011). Size and average structural complexity are depicted in the Y axis, with both measures normalized against their respective maximum values, so that they range from 0 to 1 in the chart. All projects have been growing in size since the beginning of the observed history, which indicates that they are being subject to an active evolution process. It is also interesting to note that all projects have increasing structural complexity over time, but with very different patterns.

From these charts we can also see that, while in some periods the structural complexity seems to fluctuate together.

---

**Table I: Descriptive Information for Studied Projects**

<table>
<thead>
<tr>
<th>Project</th>
<th>Language</th>
<th>Dev. time</th>
<th>#Devs</th>
<th>ELOC0</th>
<th>ELOCf</th>
<th>Modules</th>
<th>Avg. module size</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clojure</td>
<td>Java</td>
<td>5 years</td>
<td>58</td>
<td>59</td>
<td>31830</td>
<td>666</td>
<td>47.79</td>
<td>Compiler</td>
</tr>
<tr>
<td>Node</td>
<td>C++</td>
<td>2 years</td>
<td>164</td>
<td>617</td>
<td>15692</td>
<td>62</td>
<td>253.10</td>
<td>Compiler</td>
</tr>
<tr>
<td>Redis</td>
<td>C</td>
<td>2 years</td>
<td>32</td>
<td>4718</td>
<td>16801</td>
<td>48</td>
<td>350.02</td>
<td>Database</td>
</tr>
<tr>
<td>Voldemort</td>
<td>Java</td>
<td>2 years</td>
<td>35</td>
<td>9762</td>
<td>34663</td>
<td>434</td>
<td>79.87</td>
<td>Database</td>
</tr>
<tr>
<td>Zeromq2</td>
<td>C++</td>
<td>1.75 years</td>
<td>39</td>
<td>7260</td>
<td>11757</td>
<td>106</td>
<td>110.92</td>
<td>Messaging</td>
</tr>
</tbody>
</table>

---

2GitHub (github.com) is a Git-based hosting website largely used by free/open source software projects.
with the system size, that is not always the case. For example, we can notice how the Voldemort structural complexity curve looks very similar to its size curve. On the other hand, during some periods, the structural complexity drops or stays stable even though the system size is increasing: notice Node between time 0.6 and 1.0.

One can also notice periods of instability, in which even though one of the measures is evolving smoothly, the other is subject to sharp increases or decreases. In Redis, for example, between time 0.6 and 0.8 the size was growing approximately in a linear way, while the structural complexity exploded at approximately time 0.7.

D. Data extraction process

The version control repositories of the selected projects were processed with analizo\(^3\), a multi-language source code analysis and visualization toolkit. The procedure for each project is described below.

First, the project Git repository was cloned locally. Then, we listed all commit author names in the repository and created a mapping file to normalize their names. Since some authors use slightly different names during the project history, by doing this we avoid considering commits done by the same person as being made by different developers.

Then we determined programming language inclusion rules, to avoid analyzing code that is not part of the main project. For example Clojure and Node are compilers, and contain a lot of code in the language they compile (resp. Clojure and Javascript). We only analyzed their main code: Java for Clojure and C++ for Node. After that, we determined directory exclusion rules, to exclude directories that should not be included in the source code analysis, such as embedded dependencies, test code, etc. This was done by listing all the different source code tree configurations with analizo's tree-evolution tool and compiling a list of directories to be excluded for each project.

The repositories were then processed with analizo's metrics-history tool using the language and directory filter rules described previously, to obtain an initial project-specific dataset. This data set contained, for each commit that changed the source code, one row containing meta-information about the commit plus source code metrics for the corresponding source code (i.e. the resulting source code, after the commit was done). After that, the datasets were imported into a common relational database and post-processed to calculate the variables described previously \( (n, d, \Delta \text{LOC}, CF, \Delta SC, \Delta SC_i, \text{and } \Delta SC_d) \).

When calculating structural complexity for C++ and Java source code, classes were considered as "modules". In the case of C, a "module" is a pair of .c/.h source files.

IV. RESULTS

In this section, we present and interpret the experimental results according to the research hypotheses presented in Section III-A. All data analysis and statistical tests were performed using the R system [31].

An initial interesting fact that resulted from our data analysis was that the different ways of measuring experience within a project we propose in this paper are practically equivalent. Table II shows, for each project, the Spearman's \( \rho \) correlation between \( n \) (number of commits done by the developer so far) and \( d \) (number of days since the developer started in the project). One can see that there is a very strong linear correlation between these variables, what means we only need to look at one of them. For the rest of this study, we will only present and discuss the hypotheses using \( n \) as independent variable: their results are similar to the ones where we used \( d \) as independent variable.

<table>
<thead>
<tr>
<th>Project</th>
<th>Commits</th>
<th>( \rho(n, d) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>clojure</td>
<td>522</td>
<td>0.996</td>
</tr>
<tr>
<td>node</td>
<td>240</td>
<td>0.961</td>
</tr>
<tr>
<td>redis</td>
<td>117</td>
<td>0.984</td>
</tr>
<tr>
<td>voldemort</td>
<td>394</td>
<td>0.875</td>
</tr>
<tr>
<td>zeromq2</td>
<td>128</td>
<td>0.884</td>
</tr>
</tbody>
</table>

All correlations statistically significant with \( p < 0.001 \)

To test hypotheses \( H_3 \) to \( H_5 \), we built regression models in which the structural complexity increase is modeled as a linear combination of the experience of the developer making that change, the size variation and the change diffusion:

\[
\Delta SC_i = \alpha_0 + \alpha_1 n + \alpha_2 \Delta \text{LOC} + \alpha_3 CF
\]

Table III presents the resulting models for the commits that increase structural complexity.

We can see that \( H_3 \) (developer experience influences the increase in structural complexity negatively) holds only for Node, which is the only project in which the experience variable \( n \) has a statistically significant coefficient with a negative sign. Our interpretation is that the changes made by less experienced developers introduce more structural complexity in the project codebase, or alternatively that more experienced developers introduce less structural complexity. Even though that coefficient seems to have a low magnitude, one must note that the measures are in different scales.

\( H_2 \) (size variation influences the increase in structural complexity positively) holds for Clojure, Voldemort and Zeromq2. In all three projects, changes that introduce more lines of code also introduce more structural complexity. The same observation about the magnitude of the coefficient applies here: \( \Delta \text{LOC} \) and \( \Delta SC_i \) are measured in different scales.

Last, we have found that \( H_5 \) (change diffusion influences the increase in structural complexity positively) holds for all projects, except Zeromq2. In these projects changes that touch more files tend to introduce more structural complexity.

To test our hypotheses \( H_4 \) to \( H_6 \), we have built linear regression models similar to the ones presented above. For each project, we modeled the decrease in structural complexity

\(^3\)More information about Analizo can be found at analizo.org.
as a linear combination of experience of the developer, size variation and change diffusion:

$$\Delta SC_d = \alpha_0 + \alpha_1 n + \alpha_2 \Delta LOC + \alpha_3 CF$$

The results provided by these regression models are presented in table IV.

$H_4$ (developer experience influences the decrease in structural complexity positively) holds only for Clojure. Our interpretation is that in Clojure, the changes made by developers with more experience are the ones that cause larger decreases in structural complexity. In that project, anti-regressive work is done by the most experienced developers.

$H_5$ (size variation influences the decrease in structural complexity negatively) holds only for Zeromq2. In that project, the commits that produce larger decreases in structural complexity are the ones with lower number of lines of code added, or commits that actually remove lines of code. Voldemort also has a significant coefficient for $\Delta LOC$, but that coefficient is negative: in that project commits that add more lines of code are the ones producing larger decreases in structural complexity. This sounds counter-intuitive: maybe the Voldemort developers were able to reduce complexity by splitting existing modules into new ones that are larger but less coupled and/or more cohesive.

$H_6$ (change diffusion influences the decrease in structural complexity positively) holds for Node, Redis and Zeromq2. Our interpretation is that in those projects the commits that touch more files are the ones that cause a greater reduction in structural complexity.

V. DISCUSSION

The projects we analyzed in this study provided widely different results for all the studied factors. In order to identify differences between the projects and conjecture about the reasons for their structural complexity variation being influenced by some of the factors and not by the others, we want to look back at their descriptive data in table I.

Node was the only project in which higher levels of experience within the project were associated with smaller increases in structural complexity in complexity-increasing changes. The only notable difference between Node and all other projects is the fact that Node had a higher number of distinct contributors than the other projects. A possible explanation is that more experience only leads to smaller increases in structural complexity in projects with large teams, i.e. as far as structural complexity is concerned, larger teams would benefit more from having more experienced developers.

When looking at complexity-decreasing changes, we only found significant influence of experience in Clojure: in that project, changes made by developers with higher experience within the project produced a larger reduction in structural complexity. Clojure is the project with the longest development history among all of the studied projects: 5 years, against mostly 2 years for the other projects. A possible explanation is that in long-running projects, the more experienced developers are the ones that end up performing anti-regressive activities, such as refactorings.

Node and Clojure are the projects in which experience did play a role in influencing the amount of structural complexity added or removed by developers. Comparing them with the other projects, there are two aspects that called our attention:

- Both Node and Clojure started from very few lines of code ($ELOC_0$) in comparison with the their final size ($ELOC_f$), contrary to what happened to the other projects. A reasonable explanation for that would be that those projects had their important architectural decisions taken after they were first imported in the version control system. The same developers that made those decisions in the beginning would be the ones that now are able to introduce new code without adding so much complexity (in the case of Node), or to remove more complexity when doing reorganizations in the code (in the case of Clojure). The other projects could have had their important architectural decisions taken before they could
tracked, and the developers were left with no resources, no matter how experienced they were, to contain the structural complexity increase (or to drive its decrease).

- Both Node and Clojure are in the compilers domain, while the other projects in which experience did not influence structural complexity variation significantly are in other domains. It is possible that experience only plays a significant role in specific applications domains (compilers in the case of the present study) and not in others.

- It is also possible that some characteristics exhibited by Node and Clojure that was not captured in this study might cause their results to be different from the ones in the other projects.

Table V presents a summary of the obtained regression models, with the factors that had a statistically significant influence over the structural complexity variation for both types of change.

We can see that $CF$ is a recurring factor, being present in 7 out of 10 cases. The fact that a commit needs to change a large number of files may be a sign that functionality is not well factored in independent modules: whenever a developer needs to add a new functionality or correct a defect, she needs to change several separate modules. When changing lots of different modules is also associated with larger increases in structural complexity, this is an indicator of a poor modularization that is harmful to the understandability of the codebase.

$\Delta LOC$ is also a significant factor in 5 out of 10 cases. In the case of changes that decrease complexity, it is understandable why removing code makes the software less complex. In the case of changes that increase structural complexity, the fact that an increase in size necessarily implies an increase in structural complexity may also be a sign of a suboptimal modularization.

Structural complexity is an average across all modules, and so it may increase with size in two ways. First, maybe new functionality was added to existing modules while making them more complex; second, maybe functionality was added as new modules that are more complex than the average. In both cases, this may be a sign of poor modularization.

On other hand, we did not find a significant influence of $\Delta LOC$ over structural complexity variation in 5 out of 10 cases. While system size variation must be looked after, there are cases in which systems complexity does not vary according to how size does. That is the case for the Node project. As we saw before in figure 2, Node structural complexity evolution curve presents no similarity whatsoever with the corresponding size evolution curve.

Software developers and project managers can use the results discussed above by inspecting the past commits that changed the structural complexity and had a large number of changed files or a large size variation to figure out the reason why those commits make the software more complex. They can then make informed decisions on where to direct their perfective maintenance efforts. They can also use these results to monitor future changes. If the team does not have enough resources to explicitly review every commit, they can give priority to commits that add more than a specified number of lines, or that touch more than a specified number of files, since those commits may be more likely to cause larger variations in the structural complexity of the codebase.

Another important fact is that most of the models do not have a large coefficient of determination (adjusted $R^2$). This coefficient can be interpreted as the amount of variation in the dependent variables ($\Delta SC_1$ and $\Delta SC_d$) that is predicted by the independent variables, or how likely the dependent variable might be predicted by the resulting regression model. 0 means no prediction at all, and 1 means that the independent variables predict the dependent variable perfectly.

For example, in the case of changes that decrease complexity in Zeromq2, 88% of the variation is predicted by just $\Delta LOC$ and $CF$: in that project, decreasing the complexity necessarily means removing code, what can also be a sign of a suboptimal modularization.

In the other cases, we did not obtain such a good a model: coefficients of determination range from 0.04 to 0.31, what leaves room for plenty of future work in order to obtain improved models that can predict the structural complexity variation in those projects with higher accuracy.

**VI. Threats to validity**

Although it was carefully designed, the present study is not free from threats to its validity. We checked our study against the list of threats to validity by Wohlin et al [32] and the issues we found are reported in this section.

Since we only looked at free software projects, we cannot assume that the results hold for the general case of software development projects. Strictly speaking, the results also cannot be generalized to other free software projects, since that classification does not imply the use of any specific development methodology, platform, tool, or technique.

An inherent limitation of the metric we used for developer experience with the codebase, which would increase as the developer works in the project and decay in periods of inactivity, or when other developers change the code they wrote.

We also did not look at the nature of the change introduced by commits: the fact that a commit represents a bug fix, the implementation of a new feature or a refactoring may make a big difference. For example, the nature of structural complexity that is introduced by the implementation of a new

<table>
<thead>
<tr>
<th>Project</th>
<th>$\Delta SC &gt; 0$</th>
<th>Adj $R^2$</th>
<th>$\Delta SC &lt; 0$</th>
<th>Adj $R^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clojure</td>
<td>$\Delta LOC$, $CF$</td>
<td>0.20</td>
<td>Experience</td>
<td>0.04</td>
</tr>
<tr>
<td>Node</td>
<td>Experience, $CF$</td>
<td>0.10</td>
<td>$CF$</td>
<td>0.10</td>
</tr>
<tr>
<td>Redis</td>
<td>$CF$</td>
<td>0.31</td>
<td>$CF$</td>
<td>0.18</td>
</tr>
<tr>
<td>Voldemort</td>
<td>$\Delta LOC$, $CF$</td>
<td>0.22</td>
<td>$\Delta LOC$</td>
<td>0.07</td>
</tr>
<tr>
<td>Zeromq2</td>
<td>$\Delta LOC$</td>
<td>0.21</td>
<td>$\Delta LOC$, $CF$</td>
<td>0.88</td>
</tr>
</tbody>
</table>
feature may be different from the structural complexity that is introduced by a refactoring. A refactoring could increase the structural complexity because it is adding framework-like capabilities to the code base, what would in turn allow future changes to be made with a smaller increase in structural complexity.

Commits that did not change structural complexity – i.e. those commits for which \( \Delta SC = 0 \) – were not included in our analysis. It is possible for a commit that does not change the structural complexity to actually influence future commits that do alter the structural complexity. For example, a commit that introduces a serious defect might lead developers to realize that they need a large refactoring to properly avoid the same type of defect in the future. Including these commits in the analysis performed in this study would not help, though: since they all exhibit the same value for the dependent variable (\( \Delta SC = 0! \)), they would provide us no useful statistical results. We would need a different research design to take advantage of those additional data points.

Another limitation is the assumption that each commit represents a self-contained change. If there were logical changes that were performed as a sequence of separate commits, this information was lost and each commit was analyzed as being an independent change.

VII. Conclusions

This paper reports on a study to identify factors that influence the structural complexity evolution. We investigated the influence of developer experience within the project, size variation and change diffusion in the variation of structural complexity, using commit data from 5 well-established free/open source projects in different application domains and written in different programming languages.

The method applied and results found can be used as guidance to support decisions on how to assign developers for maintenance work. For example, managers that find an association between lower developer experience within a project and structural complexity increase might want to allocate their most experienced developers to work on risky changes. If no such association is found in the history of the project, maybe there is no point in allocating a specific developer to perform the change.

Managers can also monitor changes and decide whether a specific change needs to be explicitly reviewed by a second developer when it crosses a threshold of number of lines of code or number of files changed, if those are relevant factors for a given project.

This study taught us important lessons, which should be taken into consideration by practitioners looking for applying these results in their software development projects.

All of the studied factors have influence over structural complexity evolution in at least 2 regression models. This means that all of developer experience within the project, size variation and change diffusion might be taken into consideration by project managers when monitoring software metrics, or to decide on how to prioritize resources for code review.

Change diffusion is most influential factor among the ones we studied. In 7 out of 10 regression models, changes that touched more files were the ones that added/removed more structural complexity and removed. Software size was the second most influential factor, being significant in 5 out of 10 regression models. Developer experience only had a significant influence in 2 out of 10 regression models, but in one of them neither size variation nor change diffusion were significant.

System growth is not necessarily associated with structural complexity increase. Contrary to the common sense, it is possible for a project to grow without having a corresponding increase in complexity.

Different projects are influenced by different factors. A direct implication for project managers is that to apply these results on software development projects, previous project history must be studied to identify which factors influence specific projects.

Factors that influence the increase of structural complexity are different from the ones that influence the decrease. Depending on their goals, project managers might consider a different set of factors: if the main goal is to control or avoid the increase in structural complexity, a different set of factors must be considered in comparison with when the goal is to actively work in the reduction of structural complexity by means of activities such as refactorings and re-engineering in general.

There are other influencing factors that we did not investigate in this study. Our regression models did not achieve a high coefficient of determination, what means that there are other factors that influence structural complexity variation that were not considered in the present study. Future work might involve performing qualitative studies and look in detail at specific changes, trying to identify other factors that might be included in a quantitative analysis such as the one presented in this work.

We also plan to investigate other factors that may influence the increase (or reduction) of structural complexity in software projects. This includes both developer characteristics, such as familiarity with the modules being changed, and factors associated with the nature of changes performed, such as the type of maintenance (corrective, perfective etc) that is taking place at each commit.

Another venue for future work is analyzing change in structural complexity in a more detailed context: when we analyze a given change in the context of a large code base, it is perceived to make a smaller difference than if it was analyzed only in the context of e.g. the modified subsystem.

Yet another idea for future work is to perform a qualitative study on the different amounts of structural complexity variation. In figure 2 we can see that there are several points in time where projects have very sharp increases or decreases in structural complexity, and sometimes these large variations don’t come together with a comparable variation in size. A closer observation of those changes might provide a deeper insight about factors that influence structural complexity. Such a study
would also help to clarify counter-intuitive results such as the ones for the Voldemort project (a negative influence of size variation in structural complexity reduction and $R^2 = 0.88$) presented in section IV.

ACKNOWLEDGMENTS

The authors would like to thank colleagues from the Software Practices Lab at the University of British Columbia for the insightful comments. Special thanks to Gail Murphy, Nicholas Sawadsky and Robin Salkeld who provided precious feedback on a draft version of this paper. This work has been partially supported by INES (National Institute of Science and Technology for Software Engineering), funded by CNPq (Brazilian National Council of Research and Development), grant 573964/2008-4.

REFERENCES


