Structural Complexity Evolution in Free Software Projects: A Case Study

Antonio Terceiro and Christina Chavez

Computer Science Department – Universidade Federal da Bahia
{terceiro,flach}@dcc.ufba.br

Abstract. The fundamental role of Free Software in contemporary IT Industry is sometimes threatened by the lack of understanding of its development process, which sometimes leads to distrust regarding quality of Free Software projects. While Free Software projects do have quality assurance activities, there is still room for improvement and introduction of new methods and tools. This paper presents a case study of structural complexity evolution in a small project written in C. The approach used can be used by Free Software developers in their own projects to understand and control complexity, and the results provide initial understanding on the study of structural complexity evolution.

1.1 Introduction

Free Software[1] plays a fundamental role in the IT Industry in the contemporary society. Companies, governments and non-profit organizations recognize the potential of Free Software and are at least considering it. But Free Software is developed in a way too much different from the "conventional" software these organizations are used to: teams are distributed throughout the world, most of the time with no contractual obligations; normally, there are no formal requirements specifications, in the sense of those we expect in traditional software development organizations; quality assurance does happen, while not in the way an outsider expects (and perhaps it's not as good as it could be). Quoting Scacchi [Scacchi, 2007], Free Software Development is not Software Engineering done poorly, it's just different.

While there is quality assurance (QA) activities in Free Software projects [Zhao and Elbaum, 2003], there is a lot of room for improvement in this area. One of the areas in which Free Software QA can advance is at the use of project metrics to inform, guide and control development. It's not uncommon to find projects in which the lead developer(s) lost the interest and there are users/contributors willing to continue the development, but the complexity of the code makes it more practical for them to start a new project as a replacement to the original. Sometimes the lead developer(s) realize that the code

[1] also referred to in the research community as "Open Source Software" (OSS), "Free/Open Source Software" (FOSS), "Free/Libre/Open Source Software" (FLOSS) etc.

became so complex that it's more cost-effective for them to rewrite large parts of the software, or even to rewrite it entirely from scratch, than investing time in enhancing existing code.

So, if Free Software developers have tools and methods to understand and tame the complexity of their code, there will be a more healthy Free Software ecosystem. Less complex code may promote maintainability and help Free Software projects to get and retain new contributors.

This paper goals are twofold: first, to experiment an approach for studying the evolution of structural complexity in Free Software projects that can be used by Free Software developers in their own projects; second, to provide initial results on the study of structural complexity evolution in Free Software projects written in C (so we can e.g. compare them with the Java projects studied in [Stewart et al., 2006]). For that, we present a case study in which we analyze the evolution of structural complexity in a small Free Software project during a period of approximately 15 months.

The remainder of this paper is organized as follows: related work is described in section 1.2; section 1.3 briefly describes the tool infrastructure used to extract structural complexity metrics from projects written in the C language; section 1.4 presents the case study; and finally, we provide final remarks and discuss future work in section 1.5.

1.2 Related Work

Stewart and colleagues studied 59 projects written in Java that were available on Sourceforge [Stewart et al., 2006], using as a software complexity measure the metric "CplXLCoh", the product of Coupling ("Cpl") and Lack of Cohesion ("LCoh"). They found four patterns on software complexity evolution among those projects: 1) early decreasers, in which the complexity starts to decrease in the very beginning of the project's public availability, then gets stable for a period, and then starts a slight growing trend; 2) early increasers, where the complexity starts to increase just during the beginning of the project, and after some time gets stable; 3) midterm increasers, which experiences a faster growing of the complexity several months after the start of the project; and 4) midterm decreasers, that continued to decrease complexity during the middle of the observed period before stabilizing.

Capiluppi and Boldyreff [Capiluppi and Boldyreff, 2007] presented an approach to use coupling information to indicate potentially reusable parts of projects, which could be distributed as independent projects and reused by other software in the same application domain or with similar non-functional requirements. Their approach was based on a instability metric, as in the work of Martin [Martin, 2003]. This metric is defined in terms of afferent coupling (number of modules calling the module in question, C_a) and efferent coupling (number of modules that the module in question calls, C_e), as $I = \frac{C_e}{C_a + C_e}$. They show that modules (represented in their study by folders) with low insta-

bility (i.e. stable modules) are good candidates to be turned into independent, external modules.

Wu [Wu, 2006] studied the dynamics of Free Software projects evolution, and argues that it happens in the form of punctuated equilibrium: the projects alternate between periods of localized and incremental changes and periods with deep architectural changes.

1.3 `egypt`: tool support for extracting coupling and cohesion data from C programs

`egypt` is program originally developed by Andreas Gustafsson[2]. It works by reading intermediate files generated by the GNU C Compiler and producing as output a call graph in the format used by the Graphviz graph visualization software [3], so we can visualize the call dependencies between functions in C source code.

We made the following main modifications in `egypt` to use in this study:

- Implementation of variables usage detection, to identify which functions use which variables. This is used for calculating both coupling between modules (in the case where modules use variables from other modules) and lack of cohesion for a given module.
- Addition of an option to group the calls and variable usages by module, so that we can have a module dependency view. The original `egypt` only produced graphs at the function level, which makes it impossible to understand the structure of non-trivial software.
- Refactoring of the `egypt` script into an object-oriented design to be able to plug different extraction and reporting modules.
- Implementation of a metrics output which, instead of producing Graphviz input files, produces a metrics report on the extracted design including coupling and cohesion data.

Our modified version is available as a git repository at `http://github.com/terceiro/egypt`.

Since `egypt` extracts coupling data, it can also be used to carry studies like the one by Capiluppi and Boldyreff [Capiluppi and Boldyreff, 2007] to identify potentially reusable modules in Free Software Projects. This is not our interest in this paper, though.

1.4 Case study: the Ristretto project

Ristretto[4] is a fast and lightweight picture-viewer for the Xfce desktop environment. It's written in C, uses the GTK+ user interface toolkit and is licensed

[2] `http://www.gson.org/egypt/`
[3] `http://www.graphviz.org/`
[4] `http://goodies.xfce.org/projects/applications/ristretto`

under the GNU General Public License, version 2 or later. In this study we **analyze** the Ristretto project **with the goal of** characterizing it **with respect to** size and complexity over time **from the perspective** of developers.

1.4.1 Planning

In this study, our "population" is the series of releases the Ristretto project had since the beginning of its development. This includes 21 consecutive releases, from 0.0.1 to 0.0.21, spanning a period of approximately 15 months.

For each release of the project, the following data was extracted:

- Independent variables:
  - Release day (RD): the number of days that has passed since the first release. The first release itself has $RD = 0$.
- Dependent variables:
  - Physical Source Lines of Code ($SLOC$).
  - The product of average module coupling and average module lack of cohesion ($CplXLCoh$), as in [Stewart et al., 2006], as a measure of complexity. We have used the classic coupling and lack of cohesion metrics by Chidamber and Kemerer [Chidamber and Kemerer, 1994].

We formulated two hypothesis for this study, which are described below.

Hypothesis 1. We want to test whether the project presents a consistent growth, as reported in the literature for both "conventional" Software Engineering [Lehman et al., 1997] and for Free Software projects [Koch, 2007]. Since we are interested only in testing for consistent growth and don't need a precise prediction of project size, we consider enough to test for a linear correlation between the date of release and the size of the project. Our null hypothesis H_0^1 is that there is no linear correlation between time and size of the project, and our alternative hypothesis H_A^1 stands for a positive linear correlation between the variables:

$$H_0^1 : r_{RD,SLOC} = 0 \qquad H_A^1 : r_{RD,SLOC} > 0$$

Hypothesis 2. To test whether the project becomes more complex as the time passes, we want to verify how does our complexity metric evolve. The theory suggests that software projects tend to become more complex through time, unless explicit actions are taken to prevent it [Lehman et al., 1997]. Our null hypothesis H_0^2, then, says there is no linear correlation between time and complexity; our alternative hypothesis H_A^2 is that there is a positive linear correlation between them:

$$H_0^2 : r_{RD,CplXLCoh} = 0 \qquad H_A^2 : r_{RD,CplXLCoh} > 0$$

1.4.2 Data Extraction

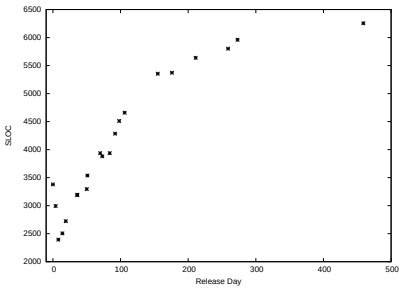
For measuring $SLOC$, we used David A. Wheeler's `sloccount` tool, available at `http://www.dwheeler.com/sloccount/`. For each release, `sloccount` is run and only the total Physical Source Lines of Code count is taken.

For measuring $CplXLCoh$, we used our modified version of `egypt`.

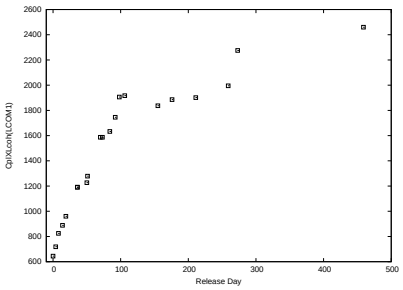
Ristretto's Subversion repository was imported into a `git`[5] repository. We then used a script that iterates through all the releases, gets the release date from the version control data, checks out the source code of the given release, invokes `sloccount` to get the size of the project at that release, builds the project so the GNU C Compiler generates the needed intermediate files, and invokes `egypt` to extract the design information from the GCC intermediate files.

1.4.3 Data analysis

Figure 1(a) presents the evolution of size in the Ristretto project, using the day of release as data in the X axis (while using the version string as label), and $SLOC$ as the Y-axis. The plot shows that Ristretto is consistently growing: it goes from approximately 2500 $SLOC$ in the first release to more than 6000 $SLOC$ in the last observed release. On the other hand, looking at the latest releases makes us wonder if the growth rate isn't decreasing.



(a) Growth of Ristretto project over time



(b) Evolution of the complexity metric in the Ristretto project

Fig. 1.1. Ristretto evolution data

The correlation test, using Pearson's method, gives us $r_{RD,SLOC} = 0.9041602$, with $p < 0.01$. This way we are able to reject the null hypothesis H_0^1 and accept the alternative hypothesis H_A^1: the current data allows us to say that there is a positive linear correlation between the release day and the size, measured in Physical Source Lines of Code.

[5] `http://www.git-scm.org/`

Figure 1(b) shows the data for our complexity metric. The plot shows that although the complexity in increasing, there are specific releases in which either the complexity does not increase significantly or even the complexity decreases in comparison with the previous release. This behavior is discussed in more detail in the section 1.4.4

The correlation test for RD and $CplXLCoh$ gave us $r_{RD,CplXLCoh} = 0.8636375$ with $p < 0.01$, also using Pearson's method. This way we can reject our null hypothesis H_0^2 and accept our alternative hypothesis: there is a linear correlation between release day and complexity.

1.4.4 Interpreting the Results

The growth data allows us to assert that the project is consistently growing since it's first release. This suggests it's being actively developed and is receiving new features as an effect of new user requirements. But growth is not out main interest in this study.

The complexity data reveals interesting issues. In the long term, the complexity grows as the time passes, but the curve we could draw through the data points shows discontinuities (see figure 1(b)):

1. Approximately in the 40th day, in the 6th release of Ristretto, the complexity increase is attenuated.
2. Approximately in the 150th day, in the 16th release, the complexity *decreases* in comparison with the previous release.
3. Approximately in the 450th day, in the 21th and last release, the complexity also seems to increase less in comparison with the previous release than it increased in the previous release in comparison with the release before it.

All these discontinuities coincide with major architectural changes in Ristretto: releases 0.0.6, 0.0.16 and 0.0.21 introduced new modules in comparison with their respective previous releases. Figure 1.2 shows the four different architectures Ristretto had during its life cycle: the leftmost graph shows the architecture of the first release, and them the architecture in releases 0.0.6, 0.0.16 and 0.0.21.

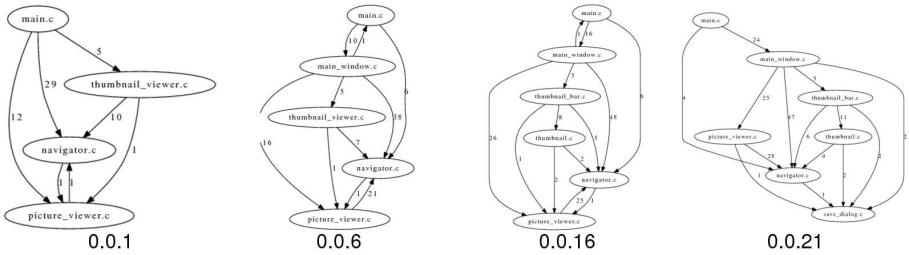


Fig. 1.2. Ristretto's architectural evolution. Graphs by Graphviz from `egypt` output.

It is easy to understand why the introduction of new modules has an impact in complexity increase. As the newly introduced module tends to be less complex than the previously existing ones, and the sum of the complexities is now divided by $n + 1$ instead of n, the new module brings down the average complexity metric.

Ristretto exhibits the behavior described by Wu [Wu, 2006]: it alternates periods of incremental change, in which the complexity increases, with moments of rupture in which the architecture changes. In the case of Ristretto, these architecture changes made it less complex, or at least attenuated the complexity increase trend at that moment.

1.4.5 Limitations of this study

We chose a small project to study on purpose, since we wanted to do a exploratory study and experiment the approach we are developing. A larger project may not exhibit a similar behavior as Ristretto.

Although the version control history data lists 13 different contributors to the Ristretto source folder, we later identified that actually only one developer made changes to the source code. The other contributors' changes were mainly updates to user interface translations, which are separated from the C source code. This way we have no data that allows us to investigate the relationship between the structural complexity and the number of developers who contributed in a given period (for example).

1.5 Conclusions

This paper presented a case study on structural complexity evolution. We analyzed 21 versions of the Ristretto project, and concluded it grows consistently, and its structural complexity increases as time passes. Both size and complexity metrics have a high correlation with the release day. We identified that specific releases where structural complexity decreases or starts to increase more slowly than in the previous release seem to be related to significant architectural changes. These changes, in the case of this small project, were additions of new modules.

We believe that our approach can be used to make larger-scale studies. These include individual studies with projects larger than Ristretto and studies comparing the structural complexity evolution of different projects. By comparing several different projects, perhaps we'll be able to associate different patterns of structural complexity evolution with characteristics of the projects. In special, it would be interesting to compare the results of studying C projects with the results of the Java projects studied by Stewart et al [Stewart et al., 2006]. Ristretto, as this paper has shown, seems to belong to the *early increasers* group described by them. Other type of study that may produce interesting results is studying structural complexity evolution in a more fine-grained scale: instead

of analyzing only the released source code, we can use the history stored in the version control system and analyze every single revision to identify the exact changes that introduced either an increase in complexity or a refactoring that made the software less complex.

The approach used in this paper can also be used by Free Software developers to monitor the structural complexity of their C projects. By using the `egypt` tool to obtain both design graphs and metrics, they can verify whether a specific change increases the overall system complexity or if a refactoring reduced the complexity in comparison with a given previous state of the code. They can also inspect the history of the projects in points in which the complexity decreased to learn important lessons about their own projects.

References

[Capiluppi and Boldyreff, 2007] Capiluppi, A. and Boldyreff, C. (2007). Coupling patterns in the effective reuse of open source software. In *FLOSS '07: Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*, page 9, Washington, DC, USA. IEEE Computer Society.

[Chidamber and Kemerer, 1994] Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

[Koch, 2007] Koch, S. (2007). Software evolution in open source projects—a large-scale investigation. *J. Softw. Maint. Evol.*, 19(6):361–382.

[Lehman et al., 1997] Lehman, M., Ramil, J., Wernick, P., and Perry, D. (1997). Metrics and laws of software evolution-the nineties view. In *Proceedings of the 4th International Symposium on Software Metrics*.

[Martin, 2003] Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[Scacchi, 2007] Scacchi, W. (2007). Free/open source software development: Recent research results and methods. In Zelkowitz, M. V., editor, *Advances in Computers*, volume 69, pages 243–269. 2007 edition.

[Stewart et al., 2006] Stewart, K. J., Darcy, D. P., and Daniel, S. L. (2006). Opportunities and challenges applying functional data analysis to the study of open source software evolution. *Statistical Science*, 21(2):167–178.

[Wu, 2006] Wu, J. (2006). *Open source software evolution and its dynamics*. PhD thesis, University of Waterloo, Waterloo, Ont., Canada, Canada.

[Zhao and Elbaum, 2003] Zhao, L. and Elbaum, S. (2003). Quality assurance under the open source development model. *J. Syst. Softw.*, 66(1):65–75.