Maintenance and Evolution: GrimoireLab Graal

W. Meijer, D. Visscher, E. de Haan, M. Schröder, L. Visscher, A. Capiluppi, I. Botez
University of Groningen, Faculty of Science and Engineering
Groningen, The Netherlands
{w.meijer.5,d.j.visscher,e.w.de.haan,m.l.schroder,l.visscher.2,i.botez}@student.rug.nl,
a.capiluppi@rug.nl

ABSTRACT

E-type open-source software inevitably grows in size and complexity over time, and without performing anti-regressive tasks this type of software has a limited lifespan. In this project, a case study of the effect of such anti-regressive tasks is conducted using GrimoireLab Graal as a subject. This process is guided by quality metrics and developer insights. The outcome of this work is a life-cycle of maintenance activities, ultimately resulting in a refactored version of GrimoireLab Graal. After applying anti-regressive actions, commonly used software quality metrics decreased (lower is better). Additionally, after performing an experiment to test the evolution readiness of the software, the complexity of the original software increased significantly, whilst no side effects were measured in the revised software.

CCS CONCEPTS

• **Software and its engineering** → **Software evolution**; **Maintaining software**.

KEYWORDS

software maintenance, software evolution, quality metrics, refactoring

ACM Reference Format:
W. Meijer, D. Visscher, E. de Haan, M. Schröder, L. Visscher, A. Capiluppi, I. Botez. 2022. Maintenance and Evolution: GrimoireLab Graal. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3524842.3528521

1 INTRODUCTION

With constant pressures to improve the performance and utility of our software, its complexity seems to inevitably grow. There are different approaches to guiding the evolution of a piece of software, varying in scope from the technical to the organizational. In this project, a case study is performed on the effect of maintenance activities on software quality. The goal of this is to gain insights into how a data-driven approach can be used in maintaining a piece of software, allowing us to make decisions based upon more than just user feedback and developer intuition. This project was developed during the MSR Hackathon 2022[1].

Anti-regressive tasks are methods by which we can reduce complexity in an effort to combat software decay [4]. This decay commonly occurs during the evolution of software when features it was not initially designed for are added. To resolve such issues, code can be re-architected and refactored.

To detect software decay, we can be guided by more than just developer intuition. Quality metrics and statistics extracted from developer tooling – while not proving the existence of a problem – can be valuable indicators [7]. This can lead us to look in places we might initially overlook.

2 SYSTEM CONTEXT

GrimoireLab was first introduced by Dueñas et al. [2] as a tool for data extraction related to software development processes. It eases the data collection step by allowing any user to access various data sources through one common interface and visualize their results. In total, it consists of 12 different core components[2] each performing a different task such as data retrieval, storage, visualization, etc. One of these components is Graal, which was first introduced by Cosentino et al. [1], extends the data retrieval component of GrimoireLab by integrating various third-party repository analysis tools (such as `Pylint` or `CLOC`) into the system. Both GrimoireLab and Graal are open-source systems, and although the original versions of Graal originally supported a limited number of third-party tools, over the years its support has grown noticeably. As the nature of this software system is similar to that of an E-type system [4], problems related to this type of system can be expected to arise.

Quality Assessment

To evaluate the expectation of evolution-related problems, quality metrics were collected using GrimoireLab itself, which supports extraction of Cyclomatic Complexity (CCN) [5], Number of Methods (NOM), and Lines of Code (LOC). When observing these metrics over time for all repositories related to GrimoireLab, it becomes apparent that the average and maximum of these metrics only increases over time. An example of this can be found in Figure 1. This seems to indicate that evolution, as indicated by Lehman [4] is following the second law: that the complexity of E-type systems will continue to increase if no anti-regressive efforts are made. Although they have been used in the past, modern literature cannot agree on the causal effect these metrics have [7]. However, when evaluating these metrics with commonly used thresholds ($CCN = 10$) it can be observed that none of the core components of GrimoireLab

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MSR 2022, May 23–24, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9303-4/22/05…$15.00
https://doi.org/10.1145/3524842.3528521

[1] Project Group "The Groninger Bugbusters". https://github.com/Groninger-Bugbusters
[2] Cereslib, ELK, Graal, Kidash, King Arthur, Manuscripts, Perceval, Sir Mordred, Sortinghat, Sigils, Hatstall, & Kibiter

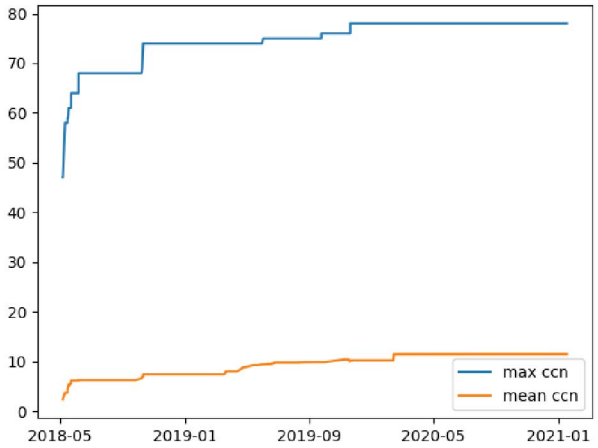

Figure 1: Change in Graal's cyclomatic complexity over time.

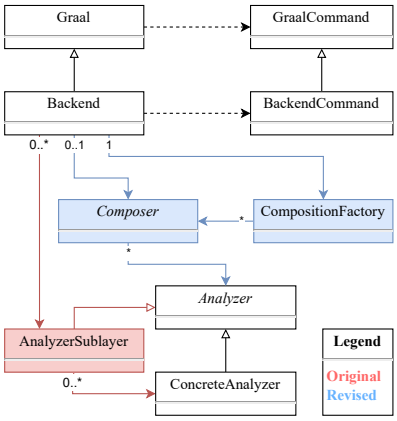

Figure 2: Component diagram of original and revised Graal architecture.

passes this test using their average CCN. In one particular case, our observations led us to a problem that already had a related open issue in the Graal repository, in which the increasing complexity was pointed out[3].

In order to measure test coverage for the different GrimoireLab components, we implemented an extension to Perceval[4]. This extension retrieves historical code coverage data for repositories that use Coveralls. Using this extension we could gather additional data, and load it into Grimoirelab, which might indicate problem-areas. It allowed us to look at the evolution of test coverage over time, and compare it with other metrics. However, we did not use this extension for any further decision making.

3 ORIGINAL ARCHITECTURE

Following the recovery process described by Medvidovic and Jakobac [6] the original architecture was recovered. During this recovery process, eight different component groups were found, within three domain groups: *System Objects*, *Back-ends*, and *Analyzers*. *System Objects* is a domain group containing all of the common Graal components, used for the general orchestration of Graal. *Back-ends* address a specific analysis type, such as code quality or code vulnerabilities, that can be analyzed; third-party analysis tools can be linked to one or more of these back-ends. The final group, *Analyzers*, consists of all third-party analyzer implementations supported by Graal. Because the different back-ends have a very similar structure, these could be abstracted as shown in figure 2.

Evolution Readiness

After closely observing the architecture of Graal back-ends, it becomes apparent that the original architectural vision of Graal was incapable of supporting the increased complexity resulting from growth in feature support. As originally every back-end was expected to have one – and only one – analyzer attached to it (e.g. the `CoCom` back-end was expected to only have `Lizard` attached to it), additional tool support was implemented in an evolution unfriendly manner, using the *AnalyzerSublayer* as an intermediary solution.

Problems related to this architecture are more apparent in more mature back-ends, as the number of tools supported in these is larger. In all of these back-ends, coupling between different components was high: at different stages of the data collection process, the system state was evaluated to determine what concrete subcomponent to use. Resulting in long `if`-trees in multiple files. Using the analysis category (an input parameter), a back-end would select a concrete analysis sub-layer. That sub-layer would then use this analysis category to select one or multiple concrete analyzer implementations. A consequence of this is high coupling and low coherence between analyzers, sub-layers, and back-ends.

As Graal is an E-type system that is expected to support more tools in the future, it is expected that this pattern will repeat itself in future versions of the software. As time goes on, solving this problem becomes more and more cumbersome, giving cause for a refactor at this point in time.

4 REVISED ARCHITECTURE

To improve the evolution-readiness of Graal, a number of changes are made to its underlying architecture[5]. Multiple alternatives with different amounts of complexity and impact, designed to tackle the identified problems, were considered to resolve these problems. Simpler solutions addressed decoupling by naively removing sub-layers or by transforming concrete analyzers into back-ends themselves. Other solutions had a larger impact on the original architecture as these introduced additional code that manages relations between different back-ends and analyzers. The final approach – selected from the set of options after consulting with a prominent developer – used a *factory-strategy-composition* solution, utilizing a number of patterns described by Gamma et al. [3] (see figure 2).

[3]Issue #89: https://github.com/chaoss/grimoirelab-graal/issues/89
[4]Repository: https://github.com/Groninger-Bugbusters/grimoirelab-perceval-coveralls
[5]See the Github Repository: https://github.com/Groninger-Bugbusters/grimoirelab-graal/tree/refac/signed-off

5 REVISION IMPACT

The impact of the refactor was tested two-fold: by measuring the change in metrics overall, and by measuring how these metrics change during software evolution. The results were measured using Graal in combination with the *Lizard Repository Analysis* category. After refactoring all six back-ends of the software, we observed that, although the maximum CCN, LOC, and NOM did not change (as the files responsible for this were not altered), the average of these values over the entire project declined (see table 1).

Metric	Original	Revised	Δ
avg. CCN	12.8125	9.2308	-3.5817
avg. LOC	92.0800	63.5125	-28.5675
avg. NOM	6.6042	4.9744	-1.6298

Table 1: Impact of the refactor on software metrics.

To test the effect of software evolution, an additional analyzer was added using both the original version of Graal and the refactored version, and the change in CCN was measured. This was done by copying one of the existing analyzer implementations (`Bandit`, which is part of the `CoVuln` back-end) and making according changes in related files. The `CoVuln` back-end only supports the `Bandit` analyzer, and thus this case study increases the supported analyzers to a total of two.

Performing this experiment on the original version affected the software in three manners: a new analyzer was added, the `CoVuln` back-end class was altered, and the `VulnAnalyzer` sub-layer class was altered. When observing the change in the complexity of the software, it could be seen that the CCN of the `CoVuln` and `VulnAnalyzer` increased from 11 to 15. When performing this experiment on the refactored software, the software was affected in two manners: a new analyzer was added, and a `Composer` file was added. Because only new files were added, the `CoVuln` remained untouched, leaving its CCN at 8. This indicates that the complexity of back-end components no longer increases when support for new third-party tools is added to the software.

6 CONCLUSION

This project addressed the effect of anti-regressive behavior on three commonly used software quality metrics by analyzing Graal software [1]. Graal has already shown indicators of unexpected evolution, giving cause to believe it is an E-type system. The code of Graal was refactored by switching its layered architecture for a *factory-strategy-composition* architecture. It was observed that the average CCN, LOC, and NOM of the software declined after the refactor. Additionally, by performing a small case study it was observed that, in the original version of Graal, the CCN of back-end components increased significantly when adding support for a new third-party tool, whilst in the refactored version no files had to be altered. This indicates that adding support for a new feature does not alter the overall complexity of the software.

In summary, the main contributions of this work are:

- The gathering of data on the quality of GrimoireLab Graal.
- The creation of a module for Perceval that allowed us to gain insights on how test coverage changed over time.
- An examination of the original architecture of Graal, guided to problem-areas using software quality metrics and developer insights.
- The selection and implementation of one of our proposed solutions, consulting with a prominent developer on the Graal project.
- The validation of the solution. Showing that quality metrics not only improved after the refactor was performed, but also yielded less additional complexity when more functionality is added afterwards.

7 FUTURE WORK

This work presented a case study in which software decay was identified using software metrics, backed up with additional developer insights, and anti-regressive tasks were performed to mitigate this decay. Currently, identification of decaying software and solutions to mitigate this decay rely very much on just developer expertise, as quality metrics only provide indications of decay.

Future work could therefore present a generalized methodology to identify and mitigate software decay. Such a methodology could levy the combination of both metrics and expertise, to improve software in ways that we could not when relying on just purely on either one.

ACKNOWLEDGMENTS

We would like to thank the MSR 2022 Hackathon organizers, the MSR program committee, and GrimoireLab developers for their support during this project. Specifically, we would like to thank Valerio Cosentino for his valuable input.

REFERENCES

[1] Valerio Cosentino, Santiago Duenas, Ahmed Zerouali, Gregorio Robles, and Jesús M González-Barahona. 2018. [Engineering Paper] Graal: The Quest for Source Code Knowledge. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 123–128.

[2] Santiago Dueñas, Valerio Cosentino, Jesus M Gonzalez-Barahona, Alvaro del Castillo San Felix, Daniel Izquierdo-Cortazar, Luis Cañas-Díaz, and Alberto Pérez García-Plaza. 2021. GrimoireLab: A toolset for software development analytics. *PeerJ Computer Science* 7 (2021), e601.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1993. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*. Springer, 406–431.

[4] Manny M Lehman. 1996. Laws of software evolution revisited. In *European Workshop on Software Process Technology*. Springer, 108–124.

[5] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[6] Nenad Medvidovic and Vladimir Jakobac. 2006. Using software evolution to focus architectural recovery. *Automated Software Engineering* 13, 2 (2006), 225–256.

[7] Alberto S Nuñez-Varela, Héctor G Pérez-Gonzalez, Francisco E Martínez-Perez, and Carlos Soubervielle-Montalvo. 2017. Source code metrics: A systematic mapping study. *Journal of Systems and Software* 128 (2017), 164–197.

,