Prioritize Technical Debt in Large-Scale Systems using CodeScene

ADAM TORNHILL, Empear AB, Malmö, Sweden, adam.tornhill@empear.com

Large-scale systems often contain considerable amounts of code that is overly complicated, hard to understand, and hence expensive to change. An organization cannot address and refactor all of that code at once, nor should they. Ideally, actionable refactoring targets should be prioritized based on the technical debt interest rate to balance the trade-offs between improvements, risk, and new features. This paper examines how CodeScene, a tool for predictive analyses and visualizations, can be used to prioritize technical debt in a large-scale codebase like the Linux Kernel based on the most likely return on code improvements.

KEYWORDS

technical debt, legacy code, large-scale systems, repository mining, CodeScene, Vendor Tools

1 INTRODUCTION

A large-scale software project is a social endeavor where hundreds of developers collaborate on millions of lines of code. The scale of such systems makes them virtually impossible to reason about for a single individual, in particular as code is a moving target that changes and evolves on a daily basis [1]. Such systems often contain considerable amounts of technical debt that drives maintenance costs, hampers innovation, and increases the risk of schedule overruns.

It's also important to note that just because some code is complicated, doesn't mean it's technical debt. It's only technical debt if we need to pay interest on it [2]. Since interest is a function of time, we need to consider the temporal dimension of the codebase to avoid spending valuable time improving parts of the code that won't have an impact. Thus, this paper suggests a prioritization technique based on how developers work with the code using the CodeScene tool.[1]

2 BACKGROUND AND RELATED WORK

Static code analysis techniques may help an organization detect potential maintenance issues such as high cyclomatic complexity and duplicated code [4]. However, static code analysis treats all code as equally important and may consequently be impractical as the pay-off on improving the code is uncertain. CodeScene resolves this issue by viewing code properties through the lens of behavioral data mined from version-control systems. CodeScene acknowledges that the importance of individual metrics varies with the situation. For example, on a small project technical issues tend to take precedence, while social factors such as coordination overhead are more important for larger projects. Hence, CodeScene applies a proprietary machine learning algorithm that ranks each source code file based on both technical and social factors.

3 METHOD AND RESULTS

We demonstrate the CodeScene tool on the open source Linux Kernel [5]. At the time of the case study, the codebase consists of 15,600,000 lines of code, 15,738 authors, 557,038 commits, and 11 years of development activity starting on 2005-94-16 and including all activity until 2016-11-30 [3].
CodeScene's algorithm identified several refactoring candidates as shown in Fig. 1. The refactoring candidates with the highest priority are located in the driver for an Intel graphics card, the i915 module. We focus the follow-up analysis on the file **intel_display.c** since it's the largest module with 12,500 lines of code that has attracted 3,040 commits during the analysis period and 535 commits just over the past year. The data also shows a high level of coordination needs since that file has been modified by 30 authors.

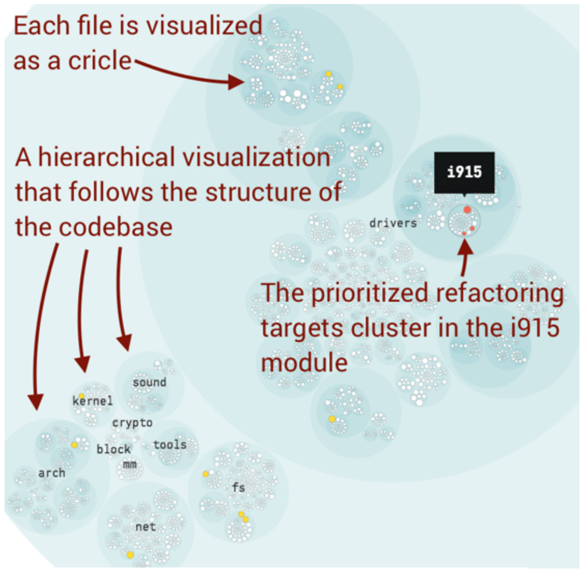

Each file is visualized as a cricle
A hierarchical visualization that follows the structure of the codebase
i915
drivers
The prioritized refactoring targets cluster in the i915 module
sound
kernel
crypto
block
mm
tools
arch
fs
net

Fig. 1. Refactoring candidates in Linux.

The next step is to determine whether the refactoring candidate continues to degrade in code quality. This question is answered by a *Complexity Trend* analysis which calculates the indentation-based complexity of **intel_display.c** [6]. The

[1] CodeScene screencast: https://www.youtube.com/watch?v=SWFwPkgLcpo

results in Fig. 2 show a strong upwards trend in both size and complexity.

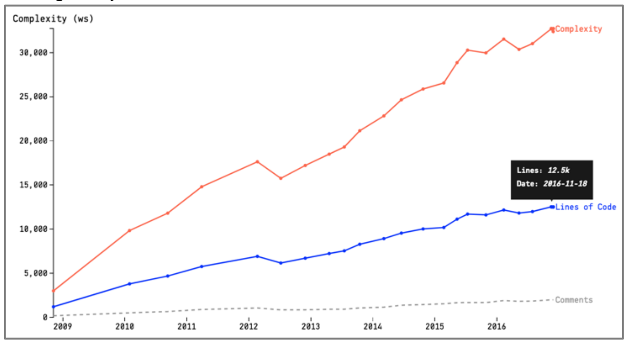
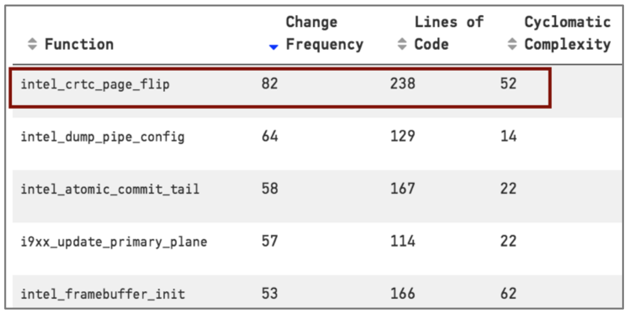


Fig. 2. The evolution of our refactoring candidate.

Based on this data, a skilled human observer could inspect the code and decide if it needs to be reworked. However, code inspections are time consuming and to be actionable in practice, the metrics have to be more specific. CodeScene resolves this by introducing its *X-Ray analysis*. An X-Ray analysis runs the same calculations, but at the individual method level as shown in Fig. 3.

To evaluate these findings, we performed a code review of the suggested refactoring candidate. This revealed several maintenance problems in the **intel_crtc_page_flip** function, and we concluded that the code would benefit from a series of refactorings like Extract Method, Introduce Explaining Variable, and Decompose Conditional [7].

The continued evolution of the **intel_display.c** after this analysis further supports our conclusion that



Function	Change Frequency	Lines of Code	Cyclomatic Complexity
intel_crtc_page_flip	82	238	52
intel_dump_pipe_config	64	129	14
intel_atomic_commit_tail	58	167	22
i9xx_update_primary_plane	57	114	22
intel_framebuffer_init	53	166	62

Fig. 3. X-Ray prioritizes functions inside large files.

intel_crtc_page_flip was a real maintenance cost; in July 2017, the code was removed with the motivation that it was the reason for a "bunch of special cases" [9].

4 SUMMARY AND CONCLUDING REMARKS

Using CodeScene, we started with more than 15 million lines of code that we narrowed down to a prioritized refactoring candidate with just 12,000 lines of code. With CodeScene's X-Ray we arrived at a specific starting point consisting of a mere 238 lines of code. More importantly, the data is now on a level where a developer can act upon the results and do a focused refactoring of the code based on how the organization actually works with the system.

Tooling like this also has the potential to highlight the impact of technical debt by letting stakeholders share a view of what the code quality looks like, as well as how the development efforts change over time. This makes it easier for non-technical managers and technical personnel to communicate as they both share the same view of the system. Such communication is often key to deciding when to focus more on features versus knowing the time to take a step back and invest in code improvements [8].

ACKNOWLEDGMENTS

This work was partially financially supported by the ITEA3 initiative TESTOMAT through Vinnova.

REFERENCES

[1] A. Tornhill. 2018. Software Design X-Rays: Fix Technical Debt with Behavioral Code Analysis. The Pragmatic Bookshelf,Raleigh, NC.
[2] W. Cunningham. 1992. The WyCash Portfolio Management System, Addendum to Proc. Object-Oriented Programming Systems Languages and Applications (OOPSLA 92). 29-30
[3] CodeScene analysis of Linux, retrieved on January 10th 2018: https://codescene.io/projects/1737/jobs/4353/results/code/hotspots/system-map
[4] G.K. Gill, & C.F. Kemerer. 1991. Cyclomatic complexity density and software maintenance productivity. IEEE Transactions on Software Engineering , Vol. 17, Issue: 12
[5] Linux Git repository, retrieved on January 10th 2018: https://github.com/torvalds/linux
[6] A. Hindle, M.W. Godfrey, & R.C. Holt. 2008. Reading Beside the Lines: Indentation as a Proxy for Complexity Metric. ICPC 2008. The 16th IEEE International Conference on
[7] D. Roberts, W. Opdyke, K. Beck, M. Fowler, J. Brant. 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, Boston, MA.
[8] E. Lim, N. Taksande, C. Seaman. 2012. A Balancing Act: What Software Practitioners Have to Say about Technical Debt. IEEE Software, Vol. 29, Issue: 6
[9] Pull request to Linux, retrieved on March 7th 2018: https://patchwork.freedesktop.org/series/2757