Assessing Technical Debt in Automated Tests with CodeScene

Adam Tornhill
Empear AB
Malmö, Sweden
adam.tornhill@empear.com

***Abstract*—Test automation promises several advantages such as shorter lead times, higher code quality, and an executable documentation of the system's behavior. However, test automation won't deliver on those promises unless the quality of the automated test code itself is maintained, and to manually inspect the evolution of thousands of tests that change on a daily basis is impractical at best.**

This paper investigates how CodeScene -- a tool for predictive analyses and visualizations – could be used to identify technical debt in automated test code. CodeScene combines repository mining, static code analysis, and machine learning to prioritize potential code improvements based on the most likely return on investment.

Keywords—technical debt; test automation; code quality; repository mining; vendor tools

I. Introduction

Many of today's software projects are social endeavors where hundreds of developers collaborate on millions lines of code. The scale of such systems make 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]. This increases risks of schedule overruns and expensive software maintenance. Test automation has grown in popularity as a technique to address those challenges [3].

However, test automation involves producing more software; automated tests are code themselves, and as such they are subject to the same maintenance challenges as any other software code [4]. Unfortunately many organizations fail to maintain the same quality in their test code as they do in their application code [5][17]. Consequently, the maintenance of automated tests is viewed as problematic by practitioners in the industry [2].

To counter this negative trend and succeed with automated tests, this paper suggests a metric-based approach where test code is pro-actively refactored as part of the ongoing work. In practice, such potential improvements have to be prioritized based on the most likely return on investment; improving existing code is always a trade-off to adding new features. The prioritization technique presented in this paper builds on behavioral code analysis where we analyze patterns in how developers work with code to come up with relevant and actionable recommendations supported by machine learning algorithms.

This paper demonstrates the technique by applying CodeScene on a well-known open source project to identify refactoring candidates in the automated tests. We also make the analysis data available as a public interactive visualization for future research [6]. The CodeScene tool is further explained in the screencast at the following URL:

https://www.youtube.com/watch?v=SWFwPkgLcpo

II. Background and Related Work

A. Static Code Analysis and Repository Mining

Static code analysis techniques may help an organization detect code smells in their automated tests such as large modules, high cyclomatic complexity, and duplicated code that limits maintenance productivity [7][8]. However, static code analysis treats all code as equally important and may consequently generate thousands of warnings on a large legacy codebase. This makes the technique impractical as the pay-off on improving the code is uncertain.

CodeScene resolves this issue by viewing static analysis techniques through the lens of behavioral data that shows where in the code the developers actually work. This behavioral data is mined from software repositories stored in a version-control system, where the tool extracts evolutionary properties like change frequencies, author contributions, and co-changing entities. Those properties are available in all modern version-control tools, and this behavioral data makes it possible for a skilled human in the loop to prioritize – or ignore – possible code smells.

B. Group Psychology

The human side of the code is invisible in the software itself, which is unfortunate since social factors have a strong impact on the project outcome [9]. Social phenomena such as diffusion of responsibility and normalization of deviance may lead to a situation where bad code is allowed to continue to deteriorate without any corrective actions [1].

This observation applies to automated test code too, which might put test code at risk for the Ringlemann effect where developers become less effective as the organization grows in size [10].

CodeScene acknowledges the power of such social factors by considering and weighting technical metrics against social metrics such as the number of developers that work on a test, their coordination needs, and their experience with the codebase. The social data is mined from the version-control history too.

C. Machine Learning

As stated above, CodeScene calculates a multitude of metrics from two data sources, the source code and its version-control history. Not all metrics are relevant in every situation. For example, on a small project technical issues tend to take precedence, while social factors such as coordination overhead become increasingly important for larger projects with more developers. Thus, the metrics need to be prioritized depending on the situation and also, ideally, unified to a single rank for each piece of code.

CodeScene resolves this by applying a machine learning algorithm that operates on those lower level metrics to detect deeper patterns. The output of that algorithm ranks each source code file according to its impact on the overall maintenance efforts in the codebase.[1]

A similar combination of code and change metrics with machine learning techniques have previously been used for fault localization [15]. CodeScene's algorithm differs as it is tailored to prioritize code under active development that is also hard to understand and, hence, to maintain. The contrast is that such code indicate weakness in the design that slows down development work, but doesn't necessarily contain bugs [16].

III. Method

To demonstrate the CodeScene tool, this paper presents an analysis of the open source project ASP.NET MVC from Microsoft [11]. ASP.NET MVC is a framework for building dynamic web sites on the .NET platform. At this time of writing, the codebase consists of 300,000 lines of code that has been developed by 125 authors. With comments and blank lines stripped away, the source code is made up of:

- 68,000 lines of application code.
- 155,000 lines of code for automated tests.

This ratio, with more than twice as much test code as application code, suggests that the test code is a stronger driver of maintenance costs than the application code itself. Hence, improvements to the most important parts of the test code are likely to have a real productivity effect.

This analysis looks at three years of development activity starting on 2013-12-12 (date of the first commit) and including all commits until 2016-12-06.

CodeScene automates all steps in the analysis, including result visualizations, and we only have to point CodeScene to ASP.NET MVC's GitHub repository. In this paper we will limit the result inspection to the CodeScene analysis *Refactoring Candidates* that prioritizes code based on its impact on the overall maintenance efforts. We also limit the investigation to the automated test code.

IV. Results

There were 2,734 commits analyzed by CodeScene for the three initial years of development on ASP.NET MVC. CodeScene's algorithm identified a clear refactoring candidate in the test code as shown in Figure 1.

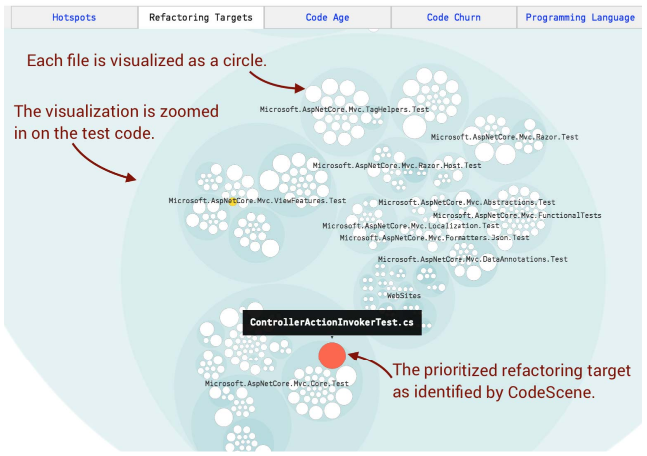

Hotspots
Refactoring Targets
Code Age
Code Churn
Programming Language
Each file is visualized as a circle.
The visualization is zoomed in on the test code.
Microsoft.AspNetCore.Mvc.TagHelpers.Test
Microsoft.AspNetCore.Mvc.Razor.Test
Microsoft.AspNetCore.Mvc.Razor.Host.Test
Microsoft.AspNetCore.Mvc.ViewFeatures.Test
Microsoft.AspNetCore.Mvc.Abstractions.Test
Microsoft.AspNetCore.Mvc.FunctionalTests
Microsoft.AspNetCore.Mvc.Localization.Test
Microsoft.AspNetCore.Mvc.Formatters.Json.Test
Microsoft.AspNetCore.Mvc.DataAnnotations.Test
WebSites
ControllerActionInvokerTest.cs
Microsoft.AspNetCore.Mvc.Core.Test
The prioritized refactoring target as identified by CodeScene.

Figure 1: A visualization of CodeScene's refactoring candidates results on ASP.NET MVC.

The results of the refactoring candidates analysis are visualized and presented in an interactive enclosure diagram that follows the hierarchical structure of the source code. Each file is visualized as a circle, and the more code in that file, the larger its diameter. The refactoring candidates are presented using color markup where the color red indicates the highest priority.

A. Analyze the Refactoring Candidate

In this case study, the refactoring candidate with the highest priority is the automated test *ControllerActionInvokerTest.cs*. It's a C# file with 2,501 lines of code that has attracted 107 commits during the analysis period. The CodeScene data also shows a high degree of coordination needs since that file has been modified by 22 authors.

The next step is to determine whether the refactoring candidate, *ControllerActionInvokerTest.cs*, presents an actual problem. And if it does, we need to know if it is code that continues to degrade in code quality or if the team is aware of a potential problem and might therefor already have started to improve the design. This question is answered by CodeScene's *Complexity Trend* analysis as shown in Figure 2.

1The specific machine learning algorithm and training techniques are proprietary and won't be explained in more depth.

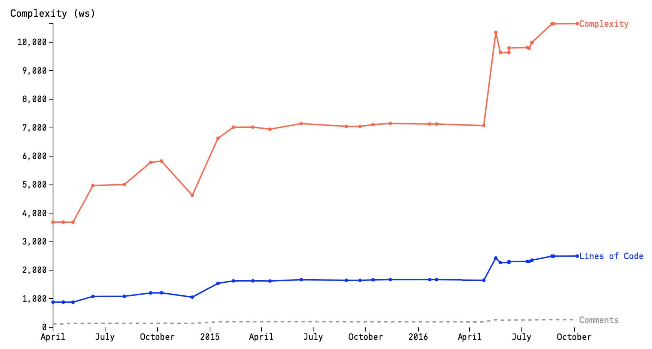


Figure 2: The evolution of our refactoring candidate in terms of code complexity.

The complexity trend analysis fetches the historic revisions of the refactoring candidate and calculates three different trends:

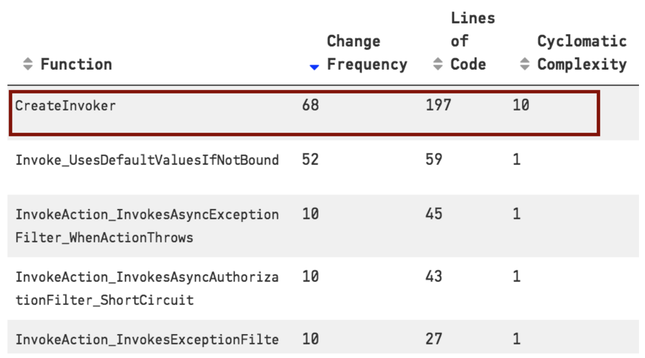
1. Complexity growth: This metric is based on indentation-based complexity in order to present a language neutral metric that works for all text based programming languages [12]. The advantage—in comparison to traditional metrics like McCabe's cyclomatic complexity--is that the same metric can be used across files implemented in different languages without the need of dedicated parsers.
2. Code growth: This metric calculates the total number of lines of code over time with comments and blanks stripped away.
3. Comments growth: Finally, CodeScene calculates the number of lines with source code comments.

Figure 2 shows that *ControllerActionInvokerTest.cs* grew dramatically back in April 2016 where it accumulated 800 lines of code and a significant complexity increase. Based on this data, a skilled human in the loop could inspect the code and decide if it needs to be reworked. However, code inspections are time consuming and the larger the source code files, the harder. To be actionable in practice, the metrics have to be much more specific.

CodeScene resolves this by introducing its *X-Ray analysis*. An X-Ray analysis runs the same calculations, but at the method level. That is, CodeScene parses the source code and maps individual methods to process metrics mined from the version-control data. The X-Ray analysis presents three metrics that help a developer choose upon a low-level starting point for a code improvement:

1. Change Frequency: The number of commits that modified each method serve as a proxy for the interest rate on any potential technical debt.
2. Lines of Code: Counts the number of effective lines of code with blanks and comments stripped away.
3. McCabe Cyclomatic Complexity: Together with the lines of code count, this metric might give a human additional insights into the kind of code that is found in the tests [13].

Figure 3 shows the X-Ray results on the refactoring candidate *ControllerActionInvokerTest.cs*.

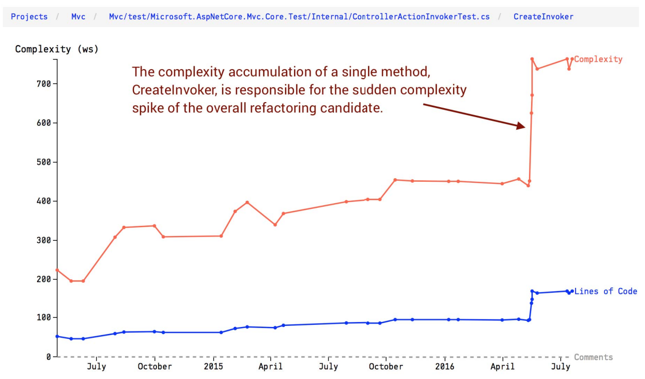


Function	Change Frequency	Lines of Code	Cyclomatic Complexity
CreateInvoker	68	197	10
Invoke_UsesDefaultValuesIfNotBound	52	59	1
InvokeAction_InvokesAsyncException Filter_WhenActionThrows	10	45	1
InvokeAction_InvokesAsyncAuthorizationFilter_ShortCircuit	10	43	1
InvokeAction_InvokesExceptionFilte	10	27	1

Figure 3: X-Ray prioritizes methods as refactoring candidates inside large files.

Based on the findings in Figure 3, the initial target for a refactoring should start in the *CreateInvoker* method inside *ControllerActionInvokerTest.cs* because it's complex code that has to be worked on often, which makes it likely that an improvement here will pay-off in terms of lower future maintenance costs through increased risk of change.

CodeScene can also calculate a complexity trend for individual methods. Figure 4 shows that the low-level refactoring candidate, the *CreateInvoker* method, has accumulated most of its excess complexity recently.



The complexity accumulation of a single method, CreateInvoker, is responsible for the sudden complexity spike of the overall refactoring candidate.

Figure 4: CodeScene calculates complexity trends for individual methods.

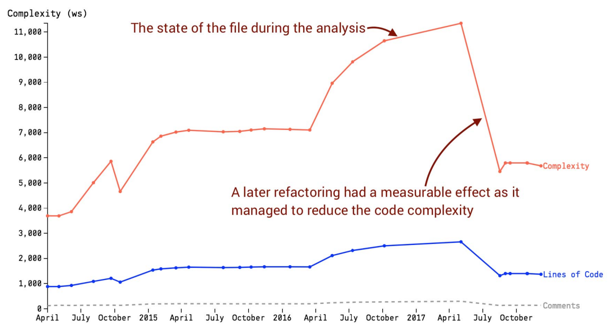
B. Evaluation and Future Evolution

To evaluate these findings, we performed a code review of the suggested refactoring candidate. This revealed several maintenance problems in the *CreateInvoker* method, and the implementation of the test in general. The most common smells in the code are Mock related, such as Mock Happy and Mocking

Everything [1]. Since CodeScene's algorithm is based on the fact that the developers need to work often with this code, these test code smells indicate a maintenance bottleneck.

The continued evolution of the codebase after this initial analysis lends further support to the results; the development team behind ASP.NET MVC had to refactor the code to increase the code re-use [14].

That work, done after the analysis results presented in this paper, can be viewed in CodeScene to assess the effects of the refactoring, which is observed in a new analysis with more recent development work. Figure 5 shows the results with a significant drop in code complexity after the refactoring described in the complexity reducing commit [14].



F

Figure 5: A complexity trend analysis shows the effect of a refactoring as reduced code complexity.

This future evolution of the file—together with the findings from the code review—suggests that CodeScene's refactoring candidate analysis identified a real maintenance bottleneck.

V. SUMMARY AND CONCLUDING REMARKS

Using CodeScene, we started with more than 300,000 lines of code that we narrowed down to a prioritized refactoring candidate with just 2,501 lines of code, and with X-Ray we arrived at a specific starting point consisting of a mere 197 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 test code based on how the organization actually works with the system.

Since virtually all large projects use version-control, the necessary behavioral data already exists. Based on experience, in a project with at least 10 contributors, 3-4 weeks of version-control history is enough for a CodeScene analysis.

These analyses have the potential to improve the effectiveness of test automation projects by letting the stakeholders share a view of what the code quality looks like, as well as how the development efforts change over time. This aspect 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.

Acknowledgment

This work has been financially supported by the ITEA3 initiative TESTOMAT Project (www.testomatproject.eu) through Vinnova – Sweden's innovation agency.

References

[1] V. Garousi, & B. Küçük, “Smells in software test code: A survey of knowledge in industry and academia”, Journal of Systems and Software (Volume 138), 2018.

[2] D.M. Rafi, K.R. Moses, & K. Petersen, “Benefits and Limitations of Automated Software Testing: Systematic Literature Review and Practitioner Survey”, Automation of Software Test (AST), 2012 7th International Workshop on, 2012

[3] E. Dustin, J. Rashka, & J. Paul, Automated Software Testing: Introduction, Management, and Performance, Addison-Wesley Professional, 1999

[4] A. Tornhill, “Your Code As A Crime Scene: Use Forensic Techniques to Arrest Defects, Bottlenecks, and Bad Design in Your Programs”, The Pragmatic Bookshelf, 2015

[5] S. Berner, R. Weber, & R.K. Keller, “Observations and Lessons Learned from Automated Testing”, Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on, 2005

[6] CodeScene analysis of ASP.NET MVC, retrieved on January 2nd 2018: https://codescene.io/projects/1690/jobs/4245/results

[7] G.K. Gill, & C.F. Kemerer, “Cyclomatic complexity density and software maintenance productivity”, IEEE Transactions on Software Engineering (Volume: 17, Issue: 12), 1991

[8] B. Baker. “On Finding Duplication and Near-Duplication in Large Software Systems”, Proceedings of the Second Working Conference on Reverse Engineering, 1995

[9] M. John, F. Maurer, & B. Tessem, “Human and social factors of software engineering: workshop summary”, ACM SIGSOFT Software Engineering Notes (Volume 30, Issue 4), 2005

[10] M. Ringelmann, M. “Research on animate sources of power: The work of man”, Annales de l'Institut National Agronomique, 2nd series (Volume 12), 1913

[11] ASP.NET MVC, Git repository, retrieved on January 2nd 2018: https://github.com/aspnet/Mvc

[12] A. Hindle, M.W. Godfrey, & R.C. Holt. “Reading Beside the Lines: Indentation as a Proxy for Complexity Metric”, ICPC 2008. The 16th IEEE International Conference on, 2008.

[13] T.J. McCabe, “A Complexity Measure”, IEEE Transactions on Software Engineering: 308–320, 1976.

[14] Commit showing the refactoring of ControllerActionInvokerTest.cs, retrieved on January 2: https://github.com/aspnet/Mvc/commit/e7bd6cfc0699af0407f2ce4cb4b2dfee2dbc7f3f#diff-9632862de855fc15d0e48ec4cec16f28

[15] J. Sohn, & S. Yoo, “FLUCCS: Using Code and Change Metrics to Improve Fault Localization”, ISSTA 2017 Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2017.

[16] D. Mathew and K. Foegen, "An analysis of information needs to detect test smells," Proceedings of the IEEE/ACMInternational Conference on Automated Software Engineering, 2016.

[17] K. Wiklund, S. Eldh, D. Sundmark and K. Lundqvist, "Technical Debt in Test Automation," 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, 2012.