An exploratory study on JavaScript applications from a Technical Debt point of view

SOEN 691E - Software Re-engineering

Everton da Silva Maldonado & Shahriar Rostami
Concordia University
Engineering and Computer Science Department

Winter 2015

1 Project Description

JavaScript is object-oriented to its core, with powerful and flexible programming capabilities. This language also support functional and imperative programming styles. It is ubiquitous, it is fast and getting faster as compared to other web programming languages. Developers can craft it manually or they can target it by compiling from another language [1] [2] . It has been few years since JavaScript [3] is competing with other server side languages like (PHP, Ruby and etc.)

Source code analysis in object oriented and generally statically typed languages has been the interest of researchers for decades [1] [2]. The purpose of static analysis can be differ from scenario to scenario. We propose a technique to find technical debts (TD) in JavaScript using source code analysis to find bad design practices [3] [4]. In the rest of this document we will use technical debts and code smells interchangeably based on the context.

The term technical debt is used to express faults and non-optimal solutions made conscientiously in a software project in order to achieve a short-term goal like meeting urgent deadlines or fixing last minutes bugs. The metaphor plays an important role in the communication between developers and managers [5], as the financial model fits well to explain delicate scenarios, that otherwise, could lead to a heavy technical conversations that, most of the time, is proven to be unproductive for both sides.

Prior work showed that technical debt is unavoidable [6] and that there are different types of technical debt, e.x. defect debt, design debt, testing debt, documentation debt, etc [7] . One of the most impacting types is design debt [8][9] (a.k.a Bad Smells). A lot of effort are made to understand and manage TD in different domains as databases schemas [10] or even in the development community [11]. However, to the best of our knowledge,

[1] TypeScript: http://www.typescriptlang.org/Tutorial
[2] CoffeeScript: http://coffeescript.org/
[3] NodeJS: http://nodejs.org

very scarce research has been done in JavaScript applications from the technical debt perspective. Thus we collaborate with the technical debt landscape analyzing its behavior in JavaScript applications.

JavaScript is one of the most utilized languages and is present in countless applications. GitHub points that JavaScript is the most utilized language hosted on its web site. However, there is several reasons that can made JS applications hard to maintain and understand like the lack of deep understanding from developers, the language weakly typing and no compilation. Therefore, its increasing utilization in addition of its inherent complexity makes technical debt management even more important.

In this work we aim to answer following research questions:

- **RQ1** - How can we compare the way JavaScript developers (dynamic world) introduce bad smells as compare with Java (statically typed languages – compile errors in case of violation)?

  Motivation: There is a difference between statically typed languages and dynamic languages. in statically typed language developers tend to remove technical debt to prevent the compiler of throwing some errors and warning. But in JavaScript which is interpretation language we can study if compiler restrictions are helpful to prevent technical debts (defects) or not by comparing one statically typed language like Java to one weakly-typed and dynamic language like JavaScript.

  Approach: If we can reach to the point to be able to match two code smells in Java and JavaScript we can differentiate how these two language can contribute to code smells.

- **RQ2** - How much technical debt can be found in JS applications?

  Motivation: In order to understand how technical debt behaves in JS applications we need to be able to identify it first. We want to know exactly how much technical debt can be found, if any, and quantify its occurrences.

  Approach: First we extract the source code from its repository and use source code based analysis tools [3],[8] (or the JSDeodorant) to parse and analyse the code. Second we store and quantify the results. Finally we mine the issue tracker in order to verify how the incurrence of technical debt relates if the number of bugs.

- **RQ3** - Know techniques like unit testing can have a positive impact reducing technical debt?

  Motivation: By comparing project with test units to those that ship without unit tests we can deduce the fact that those come with unit test is way much robust than the other. However this is the assumption we have to empirically study.

  Approach: First we extract the source code from five projects that has unit tests and from five projects that dont have unit tests from its repositories and use source code based analysis tools [3],[4] (or the JSDeodorant) to parse and analyse the code.

[4] JSLint: http://jslint.com

Second we store and quantify the results. Finally we mine the issue tracker in order to verify how the incurrence of technical debt relates if the number of bugs. We expect to verify if more unit testing in java script strongly correlates with less technical debt.

- **RQ4** - Once introduced how much of them is removed ? - one assumption is that the high number of releases could be related to the TDs.

  Motivation: The motivation behind this question is to understand how often developers introduce technical debt in the application and also how often they remove then, this can also clarify which phase of the development cycle is more likely to technical debt be introduced.

  Approach: We will take one project that has several releases and download and analyse them. We will analyze the latest version that a technical debt can be found and then we proceed to newer versions in order to determine if the technical debt still remains or not. In order to achieve that we will use the source code analysis tools [3],[5] (or JSDeodorant) to parse and analyse the source code.

2 Related papers

Detecting bad smells in source code using change history information [1]

They propose a technique to detect source code smells based on change history information extracted from versioning systems, and, specifically, by analyzing co-changes occurring between source code artifacts. We can do the same by analyzing different JavaScript versions to find more information about code smells.

Exploring the impact of inter-smell relations on software maintainability: An empirical study[2]

They mentioned measuring the defectiveness of an individua code smells had been investigated by other researchers, however in this paper they study how and in what extent interactions between code smells can affect the maintainability of the project. We can study how these code smells or generally, technical debts can affect the project maintainability.

JSNOSE: Detecting JavaScript Code Smells[3]

Author of this paper already found 13 object oriented code smells in JavaScript using a combination of static and dynamic analysis. To critique their approach one can just say how they put a threshold for those metric to consider one piece of code as code smell. Moreover requirement of having execution environment for detecting those code smells is not straightforward to be used by developers as other tools that equipped in IDEs. We can just use static analysis to find these technical debts and by having a plugin in IDE we can just notify them about the issue that they are going to introduce.

Normalizing object-oriented class styles in JavaScript[4]

They use string matching solution to find different class-like declaration in JavaScript and they tried to make them consistent in terms of implementation style. Having different

[5]JSLint: http://jslint.com

ways of declaring classes in JavaScript and each of them has its cons and pros, we can consider few of them as technical debts (i.e. lacking of reusability or polluting namespace and etc.)

Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt[5]

The author of this work states several definitions of each kind of technical debt and proposes a discussion based on the research evolution of the subject and industry experience so far. In conclusion they states open questions about the subject like When does software quality become a technical debt issue? and propose a classification of the technical debt categories that eliminates some of the known technical debts such as defect technical debt.

A case study on effectively identifying technical debt[6]

The authors conduct an experiment to compare the efficiency of automated tools in comparison with human elicitation regarding finding technical debt. They find that there is few overlap between the two approaches thus is better to combine them than replace one for another. In addition to that they states that automated tools are more efficient finding defect debt whereas humans can realize more abstract categories of technical debt.

Chapter 2 - Measuring and Monitoring Technical Debt[7]

The authors qualifies the technical debt metaphor, discussing the use of it in software development instead of in the financial context. They states that managing technical debt is complex due a high amount of uncertainty that is inherent of software development and they propose a mechanism (the use of techniques and tools) to manage existing technical debt. In their conclusion they states open questions in the research area as well they thoughts to future work.

Investigating the impact of design debt on software quality[8]

In this work the authors conducted a study to measure the impact of technical debt in software quality. Their approach was detecting a well know design TD, God Classes, and then verify how likely they are to change comparing to non God Classes classes. They find that God Classes are more likely to change and therefore has an higher impact in software quality.

Investigating the impact of code smells debt on quality code evaluation[9]

In this work the authors investigate design technical debt represented by code smells. They use metrics techniques to find three different design technical debt, God Classes, Data Classes and Duplicated Code. They propose an approach to classify which one of them should be addressed first, based on a risk scale.

Managing Technical Debt in Database Schemas of Critical Software[10]

The authors propose an analogy using technical debt to identify and manage foreign key problems found in databases schemas. In Their work, they define that lack of necessary foreign keys is technical debt in databases. They also says that it can be caused by developers who does not enforce the same rules found in the code to the database or during a migration of databases technology.

What is social debt in software engineering?[11]

The author of this paper propose a new analogy called social debt, that stands to the the unpredictable cost that suboptimal development communities can incur into a software project. They compare and map the characteristics if this analogy with technical debt and therefore propose open research questions as future work.

References

[1] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 268–278.

[2] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Software Engineering (ICSE), 2013 35th International Conference on*, May 2013, pp. 682–691.

[3] A. Fard and A. Mesbah, "Jsnose: Detecting javascript code smells," in *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, Sept 2013, pp. 116–125.

[4] W. Gama, M. Alalfi, J. Cordy, and T. Dean, "Normalizing object-oriented class styles in javascript," in *Web Systems Evolution (WSE), 2012 14th IEEE International Symposium on*, Sept 2012, pp. 79–83.

[5] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi, "Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 5, pp. 51–54, 2013.

[6] N. Zazworka, R. O. Spínola, A. Vetro, F. Shull, and C. Seaman, "A case study on effectively identifying technical debt," in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2013, pp. 42–47.

[7] C. Seaman and Y. Guo, "Chapter 2 - measuring and monitoring technical debt," ser. Advances in Computers, M. V. Zelkowitz, Ed. Elsevier, 2011, vol. 82, pp. 25 – 46. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780123855121000025

[8] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 17–23.

[9] F. Fontana, V. Ferme, and S. Spinelli, "Investigating the impact of code smells debt on quality code evaluation," in *Managing Technical Debt (MTD), 2012 Third International Workshop on*, June 2012, pp. 15–22.

[10] J. Weber, A. Cleve, L. Meurice, and F. Bermudez Ruiz, "Managing technical debt in database schemas of critical software," in *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*, Sept 2014, pp. 43–46.

[11] D. Tamburri, P. Kruchten, P. Lago, and H. Van Vliet, "What is social debt in software engineering?" in *Cooperative and Human Aspects of Software Engineering (CHASE), 2013 6th International Workshop on*, May 2013, pp. 93–96.