Restraining technical debt when developing large-scale Ajax applications

Yoav Rubin, Shmuel Kallner, Nili Guy, Gal Shachor
IBM Research - Haifa
Haifa University Campus
Haifa, Israel
{yoav, kallner, ifergan , shachor}@il.ibm.com

Abstract - **Addressing technical debt during the software development process relies heavily on a refactoring phase, in which automatic code transformations are used as a crucial mechanism to reduce a system's technical debt. However, automatic refactoring is not an option when developing Ajax applications. Therefore, an approach that restrains the accumulation of a system's technical debt is needed. In this paper, we present and evaluate such an approach and its reification as a framework. We conclude that our proposed framework enables restraining technical debt in a large-scale Ajax application without the need for automatic code refactoring tools.**

Keywords: software engineering; dynamic languages; code reuse; technical debt; Ajax

I. INTRODUCTION

Software development is an engineering discipline, and as such, it is composed of an ongoing process of decision making on the one hand and acting upon these decisions on the other. A key aspect in a project's decision-making process is handling technical debt [1]—the toll that suboptimal decisions or actions impose on the future welfare of that project. Technical debt is resolved using technical means and resources.

The impact of suboptimal decisions on a project resembles the impact of financial debt. In some cases, incurring small debts can result in large future rewards. Yet, debt usually comes with interest, which if not paid on time can inflict severe consequences, including a complete halt of the related activity [1].

Technical debt is considered one of the causes of hatching a catastrophe [2] and may affect the eventual success of a software project.

One of the most common technical debt payback strategies, which endeavors to decrease, manage, and control technical debt [3], is code refactoring [4]. This is a code modification process, that can be done either manually or using automatic tools. The essence of this process is to apply behavior-preserving transformations to the code in a way that the resulting code provides better reusability, compatibility among different components, and simplicity of the iterative software design process [5].

A. *Refactoring dynamic languages codebase*

Ever since the refactoring browser [6] was introduced, targeting the Smalltalk-80 [7] programming language, several attempts were made to create refactoring tools for dynamic languages [8, 9]. These tools aimed at performing automatic refactoring transformations on code written in a dynamic language such as Ruby [10] or JavaScript [11]. Each such tool tried to overcome the lack of type information, which is essential for correct refactoring transformations [5], by using other sources of information. In the refactoring of Smalltalk codebase, the automatic tool used a combination of test-cases, results of dynamic analysis, and method wrappers [6]. Another technique is static pointer analysis, which was the vehicle that drove automatic refactoring in JavaScript codebases [9]. Another strategy was to rely not only on the analysis of a project's codebase, but rather on additional information provided by the developers, as was done in a Ruby codebase refactoring mechanism [8].

However, automatic refactoring that is based on the techniques described above does not always result in behavior-preserving transformations. Basically, these tools breach the complete correctness requirement that is assumed by developers when using automatic tools. This partial correctness is unavoidable. It can be attributed to the fact that these tools rely on the existence of non-compulsory information, such as a test suite with complete coverage [6], or assume the absence of dynamic behavior [9].

The semi-automatic approach that relies on user input also has its downsides, as it may lead to user errors and suffers from occasional false-negative effects [8].

B. *Constraints of Ajax development*

The frontend development of web applications is a special case of using a dynamic programming language.

In this domain, the software development is usually done using a collection of technologies termed Asynchronous JavaScript and XML—Ajax [12]. Ajax builds a complete stack of technologies, from document structuring in HTML [13] through its internal representation using Document Object Model (DOM). APIs [14] and visual aspects are modified using Cascading Style Sheets (CSS) [15]. Communication is usually done using the XmlHttpRequest API [16], while interaction among all of the above technologies (and many more) is done using the JavaScript programming language [17].

All modern web browsers implement the stack of Ajax technologies, though the implementations are not identical. Therefore, in addition to understanding each of these technologies, Ajax developers face the cross-browser compatibility problem [18]. Each technology must be executed within different browsers that might have slightly different semantic interpretations of syntactic elements or might simply have implementation bugs [19].

To reduce the efforts involved with using several technologies and to address the problem of several implementations for each technology, two main approaches are used when developing Ajax-based software [20]:

- The first strategy is to write code in a non-Ajax technology that compiles into Ajax code. One example for this approach is using GWT [21], which is based on Java technologies. Another example is using the CoffeeScript [22] language, which provides syntactic sugaring on top of JavaScript.
- The second strategy is to use a web development library that buffers the different incompatibilities among browsers. Examples for such libraries are YUI [23] and Dojo [24].

Combining these two approaches is possible by using a development platform that is not based on Ajax code but rather compiles down to JavaScript code, which runs on top of a JavaScript library, e.g., ClojureScript, [25] which is compiled to run on top of Closure [26].

The main drawback of the first approach results from the fact that another level of indirection has been added, possibly making it difficult to trace problems in runtime back to the appropriate location in the source code. The main drawback of the second approach results from the fact that libraries also define their own coding idioms, which are different than those of pure JavaScript. Specifically, libraries tend to provide their users mechanisms of object-oriented programming (OOP). This is done in each library by providing unique definitions of a metaobject and of metaobject protocols [27]. This can be thought of as an additional, ad-hoc programming language layer that is specific for the defining library.

The inconsistency among the coding idioms of the various JavaScript libraries results in the inability to create automatic refactoring tools that are library-agnostic, as each such library requires its own code analysis and refactoring mechanisms.

Due to the dynamic nature of JavaScript alongside the differences among the coding idioms of different Ajax libraries, automatic refactoring tools that target Ajax code base do not exist.

The absence of these tools results in a situation in Ajax codebase in which resolving technical debt by using refactoring is done manually. Thus, this process is demanding, error-prone, and difficult to perform, especially when large transformations are needed.

Based on the previously described constraints, along with the innate nature of dynamic languages, using our financial analogy, we can describe technical debt in an Ajax application as a loan shark debt. This is due to the cost of falling back on payments—using dynamic languages means that many errors are detectable only at runtime. This type of debt also leads to the almost impossibility of paying it off once it starts to accumulate (no automatic refactoring tools exist). Naturally, preventing such debt and restraining it once it starts to accumulate should be a high priority.

The purpose of this work is to describe how a small development team was able to use our framework to deliver a large-scale web application in a relatively short time, while facing the issue of technical debt in an Ajax application. Our approach does not rely on automatic tools, but rather proactively uses abstractions and patterns [28] and especially adheres to the idea of lists as the skeleton of software components [29]. Our approach resulted in a software project that incorporated mechanisms to prevent and restrain technical debt so as to enable the successful delivery of a high-quality product.

The remainder of the paper is structured as follows: Section 2 describes related work. Section 3 introduces the project; Section 4 discusses the abstraction and the way that it was reified. Section 5 presents the evaluations performed with regard to the use of the abstraction. We present the results of our evaluations in Section 6 and we explain them in Section 7. Finally, Section 8 concludes the paper and outlines possible future directions.

II. Related work

The idea of addressing technical debt as part of the development process, though initially presented decades ago [1], has just begun to resurface and has gained significant interest in the software development community in recent years [31]. As such, not much academic research has been published on this issue to date. Moreover, most of the existing work revolves around the management of technical debt, with an "after the fact" approach, namely by employing various code refactoring methods [4]. This is accompanied with a decision-making process to optimize debt reduction while facing the costs of the code refactoring [32].

Amongst the work that was performed to date on technical debt, we are not aware of any work that is focused on approaches to restrain such debt in a "factor instead of refactor" fashion. An iterative approach, which can be thought of as a compromise between a "post-debt" and "pre-debt" approach, was discussed by Nanette Brown, et al. [33]. They suggested methods to assess the resulting technical debt in an iterative architectural project planning process by using dependency analysis. Such measurements can help in making the right architectural decisions and thus decrease the accumulating debt.

Handling technical debt in a large-scale web application project was discussed by Israel Gat and John D. Heintz [34]. Their paper presents how the Cutter's technical debt assessment tool, which employs both static and dynamic code analysis methodologies, was used to define a technical debt reduction project—one that included a complete rewrite of the frontend component in JavaScript. Reassessment of the new frontend implementation showed that the amount of code duplication remained significantly high (40%).

III. USE CASE

A team needed to develop a web frontend component of a case management [30] product using Ajax technologies, particularly the Dojo toolkit [24]. The project was assigned to a team composed of a lead developer experienced with web application development and a few developers lacking this expertise. The project itself had a tight schedule and needed to be released as part of a larger product with strict deadlines. It had to be developed using an agile methodology, as most of the user interface and user experience requirements were to be defined in an iterative fashion. From day one of the project, the team could clearly see that due to the time constraints, development friction resulting from technical debt could cause the entire project to fail and would have a severe impact on the entire product. Technical debt cannot be overlooked and must be avoided. A solution that prevents the future accumulation of technical debt had to be devised before any other aspect of the component could be developed.

IV. SOLUTION

A. *Code and abstraction reuse*

We designed a development strategy in light of the experience gained by the team's technical leader in previous web application development projects. Our strategy was to base the software components on a single abstract idea, whose essence is that an application's frontend is composed of various manageable lists of repetitive items, each consisting of another element. Within each list, the items are identical in their list management behavior (adding, changing location, removing), yet they may vary in presentation as well as in the elements that each list item contains.

To allow maximal reusability of this abstraction, we needed to develop an implementation that was as flexible as possible. As such, we developed an implementation that could handle all the list management related aspects and the entire lifecycle of the nested elements, all while remaining presentation-agnostic.

A hidden design agenda of the manageable list abstraction was to force its users to provide code that adapts the abstraction's core functionality alongside the presentation rules, within each use. This would result in constructing a mental model of the abstraction's capabilities from the beginning. Our intention was to verse the developers in using the abstraction for all types of needs, thus enabling them to compose much of an application's frontend from building blocks that are extensions of this idea.

B. *The Wrapper/WrapperContainer framework*

We turned the reification and implementation of the managed list abstraction into a framework composed of three classes. Two classes correspond to a list—one for a general list and one that supports a drag and drop operation among the list items. The third class corresponds to a list item. We implemented the following responsibilities into the framework:

- List management
- Event handler with callback hooks
- Lifecycle management of the list, items, and the nested elements

The list item abstraction was implemented in a class called Wrapper, as it acts as a general wrap for any kind of element. The list abstraction was implemented in the class WrapperContainer, as it acts as a container for wrappers, and DndWrapperContainer, which stands for a WrapperContainer that supports drag and drop operations.

The entire framework was implemented in six hundred and forty lines of code (LOC), all in JavaScript and using the Dojo toolkit APIs. The hooking up of the callbacks as well as the possibility to manage the lifecycle of the wrapped element was based on the dynamic nature of JavaScript alongside its idiomatic usage of runtime time inspection.

V. EVALUATION

A. *Methodology*

To understand the impact of using the abstraction and framework we described above on the project, and especially to determine if it stood up to its target of technical debt restraint, we designed and performed two different evaluations. The first is based on lines of code analysis and the second on a review by a group of experts. This combination of methodologies was picked so it would provide a clear view as to whether the framework was used appropriately, and if so, the extent of its usage.

B. *Analysis of the project's codebase*

We completed the first evaluation by performing a static analysis of the project's code to measure the portion of reuse that can be attributed to the framework in an attempt to reveal the cost effectiveness of investing in designing, implementing, and using it. Our results pointed out the extent of the framework's use, and hence its significance in the overall codebase.

To perform this analysis, we divided the codebase into three distinct components:

- Framework: the code that was used to develop the Wrapper/WrapperContainer framework
- Extensions: the code that was used to develop the widgets that extend the Wrapper/WrapperContainer framework (a widget is a class, or other software component, which also has a visual representation)
- Other: all the project's code that is not part of the framework or extending it

In our analysis, we concentrated only on the portion of the Ajax codebase that was written in JavaScript, as HTML code is almost always tailored for a specific use. Also, most of the CSS code was part of a library that was used throughout the organization—one that was not developed as part of the project. Another part of the project's codebase that we ignored in this measurement was a small JavaScript library that was developed in another project and was used "as-is".

C. *Review by experts*

The second evaluation was done by having five software engineers versed in the domain of large-scale web application development perform reviews of the project.

These engineers were qualified as experts based on the following "expert's threshold" criterion:

- More than 10 years of professional experience as a software engineer
- Of which, at least 5 years working as a front-end engineer
- Of which, at least 3 years working as part of a team that develops large scale Ajax-based web application

We educated the evaluating engineers about the Wrapper/WrapperContainer framework and asked them to use the application and inform us of any place in the application where they see fit for using our framework. Their answers were later compared to their actual use of the framework.

The analysis of the overlap between the reviewers' answers to their actual use was performed to gain an understanding on the use coverage of the framework, i.e., whether the team had used it as much as possible, thus efficiently restraining the project's technical debt. This is especially important in lieu of the hidden agenda behind the design of the abstraction. Moreover, from the reviewers' answers, we could see whether the abstraction indeed fits the domain.

On top of that, as a side-effect of this measurement, we can detect whether technical debt still exists in the system due to not using the framework where it could have been used.

VI. RESULTS

A. *Analysis of the project's codebase*

We present the results of our first evaluation in Table I, showing that the Wrapper/WrapperContainer framework was extended twenty times in the project, as each file corresponds to a class. The forty files marked as extensions are basically twenty pairs, with each such pair composed of a class that extends Wrapper and a class that extends either WrapperContainer or DndWrapperContainer, depending on its need to provide a drag and drop behavior to the user.

TABLE I. PROJECT'S CODE BY COMPONENT

	Framework	Extensions	Other	Total
Number of LOC	640	9054	13725	23419
Percentage of LOC of entire project	2.73%	38.66%	58.61%	100%
Number of files	3	40	70	113
Percentage of files of entire project	2.65%	35.39%	61.96%	100%

In light of the large number of reusing classes, especially when considering the fact that more then a third of the project classes are extensions of the framework, we can easily accept that the designed abstraction does play a central role in the project.

Also of note is the fact that the portion of the code that extends the framework attributes for more then 38% of the application's code. When we look at the framework alongside its extenders, we see that the list abstraction covers more then 40% of the application code.

From these numbers, not using this abstraction and solving each of the twenty usage scenarios differently would clearly have enforced a large allocation of resources—such as adding more time by delaying the project deadlines or adding more developers. Needless to say, these solutions were unacceptable.

Moreover, in cases in which technical debt is created as part of a specific extension of the framework, it would be secluded from other parts of the application. This results in reduced code cohesiveness and minimal effect of each scenario on the overall technical debt of the system.

B. *Review by experts*

The results of the second evaluation are presented in Table II, which summarizes several review sessions that were held with three experts. It is important to state that we believe that the reviewers' high expertise and deep knowledge in the domain of web applications development more than compensates for the small number of reviews.

Table II shows how many locations in the application each reviewer thought were applicable for using the framework (the Found column). Such locations were marked either as a location where the development team had indeed used the framework (the In use column) or marked as a location where the team did not use the framework (the Not in use column).

Table II clearly shows that the abstraction that was reified by the framework is indeed a natural fit for the project. The table also hints that it can be used in other frontend projects, as the five reviewers found a high number of places to use it within the discussed application. This is due to the reviewers' familiarity with the usage of the managed list abstraction in such applications.
Moreover, although not presented in this table, all of the twenty places where the team used the framework were detected by the experts (when we superpose their reviews).

TABLE II. REVIEWS SUMMARY

	Found	In use	Not in use
Reviewer-1	18	18	0
Reviewer-2	17	17	0
Reviewer-3	16	16	0
Reviewer-4	19	19	0
Reviewer-5	19	18	1

One usage location that was pointed out by Reviewer-5 was marked as Not in use. The framework was not used there since it required a modification to the framework that would change its semantics, a task that the team preferred not to do at the time that that specific location in the application was implemented.

VII. DISCUSSION

In addition to the results presented in the previous section, it is important to state that the project was delivered on time, with high quality, and was praised by its stakeholders and clients. Thus, in light of the aforementioned statements and based on the presented results, we would like to address the idea of using the managed list abstraction alongside its logical reification by the Wrapper/WrapperContainer framework from several perspectives.

A. *Software design perspective*

From a software design perspective, the Wrapper/WrapperContainer framework, as reification of the managed list abstraction, was a natural fit to act as a main component in a large-scale Ajax project. This component had proved itself as flexible enough to be used in numerous contexts while preserving and reusing its core functionality.

The decision to provide a non-holistic component that incorporates a complete logic implementation with callback hooks and that lacks visual representation resulted in a highly usable and flexible component as the apparent choice in the trade-off between adaptiveness and rigorousness.

B. *Developer's cognitive load perspective*

We discovered that the use of a single abstraction as the main workhorse of the frontend code had a twofold benefit:

- A lower learning curve: The project's novice developers had to learn only one main abstraction and quickly become proficient in using it and its reification. They were able to do so after an almost insignificant portion of time with respect to the entire project's duration. The reviewing experts understood it after a thirty minute educational session.
- The proficiency of the developers in using the abstraction and the framework: Interpreting that the large number of uses of the framework within the project was a result of the assimilation of the abstraction and the framework into the developers' mental arsenal is sensible. Thus, the developers used it in all appropriate situations. Even during times when the deadline pressure increased, the developers still thought of using the abstraction and the described framework as the path of least resistance.

These two gains resulted in a lower mental burden on the developers, which allowed them to free mental resources to make better decisions and find better solutions in the overall development process.

C. *Debt accumulation perspective*

As a result of the limitations imposed by the project's technological domain, the development process of "first code then refactor" was thought of as inadequate. The lack of automatic refactoring tools forced developers to think ahead about their solutions and code to come up with a development flow that did not assume the existence of automatic refactoring tools. This approach was nicknamed "factor instead of refactor". The absence of such an efficient debt payment mechanism resulted in the emergence of a paradigm that minimizes the accumulation of technical debt. This paradigm achieved its goal by focusing on a highly useful abstraction with flexible implementation. This kind of focus had an effect on the project's technical debt similar to the effect of a highly rewarding investment. Basically, since the framework was reused twenty times throughout the project, we can say that the "factor instead of refactor" approach was nineteen times more efficient then the "code then refactor" approach.

We can conclude that using the managed list abstraction and the Wrapper/WrapperContainer framework as a mechanism to control and restrain technical debt in a large-scale Ajax project has proven itself beyond any expectation of the development team. Its contribution to the successful delivery of the project, on time, and with high quality, is highly significant.

VIII. CONCLUSIONS AND FUTURE WORK

Large-scale web applications are becoming abundant for various reasons. An Ajax-based frontend is a crucial component in cloud-based applications as well as in non-native mobile applications. Moreover, users are expecting to have the ability to access their once desktop-only applications via web interfaces. However, the domain of web application development is relatively young, and due to its special constraints, traditional software development processes and tools are rarely sufficient.

Mapping knowledge and ideas that are applicable for static languages to be used in dynamic languages is in dire need. Tools that rely on information that is extracted from a programming language type system (such as automatic refactoring tools) have a key role in the development process of modern software. Since such information is not found in Ajax-based applications, these tools are not available for software developers, and thus standard development processes become less effective up to the point that a project's success can be jeopardized.

Methodologies and paradigms that handle problems that occur in large software projects need to be adapted to web application projects. One such problem, addressed in this work, is how to restrain technical debt. In this paper, we presented one solution—the abstraction of a managed list and its implementing framework. However, other abstractions may fit other types of projects. These abstractions target not only JavaScript components but other technologies as well, such as CSS or HTML.

Finding ways to track technical debt that originated from components implemented using different technologies and multiple programming languages, as part of a single software project, is also necessary. Moreover, we also must address the innate technical debt that is found due to the cross browsers compatibility problem. As such, finding ways to mitigate it into a debt-tracking system that does not yet exist is also a worthy research direction.

ACKNOWLEDGMENT

The authors would like to thank Jayasimha S Kanakatte, Kalaivanan Saravanan, Ravi Ray, Seema Meena, and Susheel Ahuja for all the time they invested on the project. We would also like to thank Maya Barnea, Yossi Mesika, and Andrei Kirshin for the fruitful discussions.

REFERENCES

[1] W. Cunnigham, "The WyCash Portfolio Management System" in Addendum to the proceedings on Object Oriented Programming Systems, Languages, and Applications, pp. 29-30, 1992.

[2] F. P. Brooks Jr, "The Mythical Man-Month" (anniversary ed.). Addison-Wesley Longman Publishing. 1995, pp. 66-69.

[3] A. Nugroho, J. Visser, and T. Kuipers, "An Empirical Model of Technical Debt and Interest". In Proc. of the 2nd International Wrokshop on Managing Technical Debt (MTD 2011), 2011.

[4] M. Fowler. "Refactoring: Imporving the Design of Existing Code". Addison-Wesley Longman Publishing. 1999.

[5] W. F. Opdyke, "Refactoring Object Oriented Frameworks". PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[6] D. Roberts, J. Brant, and R. Johnson, "A Refactoring Tool for Smalltalk". Theory and Practice of Object Systems. Vol 3, Issue 4, 1997. pp. 253-263.

[7] A. Goldberg and D. Robson, "Smalltalk-80: The Language and its Implementation". Addison-Wesley Longman Publishing. 1983.

[8] T. Corbat, L. Felber, M. Stocker, and P. Sommerlad, "Ruby Refactoring Plug-in for Eclipse. In proceedings of Object Oriented Programming, Systems, Languages, and Applications. 2007. pp. 779-780."

[9] A. Feldthaus, T. Millstein, A. Moller, M. Schafer, and F. Tip, "Tool-supported Refactoring for JavaScript".In Proceedings of the ACM International conference on object oriented programming systems languages and applications. 2011. pp 119-138

[10] D. Flanagan and Y. Matsumoto, "The Ruby Programming Language, first Edition". O'Reilly Media. 2008.

[11] ECMA. ECMAScript Language Specification, 5th edition, 2009. ECMA-262 - accessed Aug. 13th, 2012.

[12] J.J Garret, "Ajax: a New Approach to Web Applications, http://adaptivepath.com/ideas/ajax-new-approach-web-applications. - accessed Aug. 13th, 2012.

[13] The World Wide Web Consortium (W3C), "HTML 4.01 Specification", http://www.w3.org/TR/REC-html40/ - accessed Aug. 13th, 2012

[14] The World Wide Web Consortium (W3C), "Document Object Model (DOM) Level 2 Core Specification", http://www.w3.org/DOM/ - accessed Aug. 13th, 2012

[15] The World Wide Web Consortium (W3C), "Cascading Style Sheets ",http://www.w3.org/Style/CSS/ - accessed Aug. 13th, 2012.

[16] The World Wide Web Consortium (W3C), "XMLHttpRequest", http://www.w3.org/TR/XMLHttpRequest/ - accessed Aug. 13th, 2012

[17] A. T. Holdener III. "Ajax: The Definitive Guide". O'Reilly Media. 2008.

[18] A. Mesbah and M. R. Prasad, "Automated Corss-Borwser Compatibility Testing". Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011).

[19] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz, "Web Browser as an Application Platform: The Lively Kernel Experience". Sun Microsystems Laborateries Technical Report TR-2008-175, January 2008.

[20] T. Mikkonen and A. Taivalsaari. "The Mashable Challenge: Briding the Gap Between Web Development and Software Engineering". In Proceedings of the FSE/SDP workshop on Future of software engineering research (FoSER '10). 2010.

[21] Google, Inc., "Google Web Toolkit Overview". http://code.google.com/webtoolkit/overview.html - accessed Aug. 13th, 2012.

[22] http://coffeescript.org/ - accessed Aug. 13th, 2012.

[23] Yahoo! Developer Network, "YUI Library". http://developer.yahoo.com/yui/ - accessed Aug. 13th, 2012.

[24] The Dojo Foundation, http://dojotoolkit.org/ - accessed Aug. 13th, 2012.

[25] S. Sierra, "Introducing ClojureScript". http://clojure.com/blog/2011/07/22/introducing-clojurescript.html - accessed Aug. 13th, 2012

[26] Google, Inc. "Closure Tools". http://code.google.com/closure/ - accessed Aug. 13th, 2012.

[27] G. Kiczales, J. des Rivieres, and D. G. Bobrow. "The Art of the Metaobject Protocol". Cambridge, MA: The MIT Press, 1991.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Software".Addison-Wesley, 1994.

[29] J. McCarthy, "Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I". Communications of the ACM, vol. 3 Issue 4, pp. 184-195, April 1960.

[30] M. Zisman, "Representation, Specification, and Automation of Office Procedures". PhD thesis. Wharton Business School, University of Pennsylvania, 1977.

[31] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R.L. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka. Managing technical debt in software-reliant systems. In Proceedings of the FSE/SDP workshop on Future of software engineering research (FoSER '10). 2010.

[32] N. Zazworka, C. Seaman, and F. Shull. "Prioritizing Design Debt Investment Opportunities". In Proc. of the 2nd International Wrokshop on Managing Technical Debt (MTD 2011), 2011.

[33] N. Brown, R.L. Nord, and I. Ozkaya,M. Pais. "Analysis and Management of Architectural Dependencies in Iterative Release Planning". In Proceedings of the Ninth Working IEEE/IFIP Conference on Software Architecture (WICSA). 2011.

[34] I Gat, J. D. Heintz. "From Assessment to Reduction: How Cutter Consortium Helps Rein in Millions of Dollars in Technical Debt". In Proceedings of the FSE/SDP workshop on Future of software engineering research (FoSER '10). 2010