Mining Knowledge on Technical Debt Propagation

Tomi 'bgt' Suovuo, Johannes Holvitie, Jouni Smed, and Ville Leppänen

TUCS – Turku Centre for Computer Science,
Software Development Laboratory &
University of Turku,
Department of Information Technology,
Turku, Finland
{bgt, jjholv, jouni.smed, ville.leppanen}@utu.fi

Abstract. Technical debt has gained considerable traction both in the industry and the academia due to its unique ability to distinguish asset management characteristics for problematic software project trade-offs. Management of technical debt relies on separate solutions identifying instances of technical debt, tracking the instances, and delivering information regarding the debt to relevant decision making processes. While there are several of these solutions available, due to the multiformity of software development, they are applicable only in predefined contexts that are often independent from one another. As technical debt management must consider all these aspects in unison, our work pursues connecting the software contexts via unlimited capturing and explanation of technical debt propagation intra- and inter-software-contexts. We mine software repositories (MSR) for data regarding the amount of work as a function of time. Concurrently, we gather information on events that are clearly external to the programmers' own work on these repositories. These data are then combined in an effort to statistically measure the impact of these events in the amount of work. With this data, as future work, we can apply taxonomies, code analysis, and other analyses to pinpoint these effects into different technical debt propagation channels. Abstraction of the channel patterns into rules is pursued so that development tools may automatically maintain technical debt information with them (the authors have introduced the DebtFlag tool for this). Hence, successfully implementing this study would allow further understanding and describing technical debt propagation at both the high level (longitudinal technical debt propagation effects for the project) and the low level (artifact level effects describing the mechanism of technical debt value accumulation).

1 Introduction

Technical debt is a software development concept that is interested in exposing asset management characteristics for project trade-offs [5]. Working with scarce resources to fulfill ever-changing requirements, software projects often need to

emphasize certain development driving aspects over others, such as delivery deadlines over thorough documenting. Further, invalid or lacking knowledge on certain aspects of the development may lead to emphases made that improperly reflect the actual situation. In both cases the informed and uninformed decisions result to trade-offs that accumulate technical debt [13].

It has been argued [16] that a key factor for the adoption of technical debt management into software development is the capability to produce and maintain technical debt information within the project. That is, the project trade-offs must be identified, their distribution and effects defined, and this information must be maintained to reflect the true software project state. Undoubtedly, failures in the information delivery result in unmanaged technical debt, or decisions being made based on outdated information, both of which, implicitly or explicitly, affect the project.

Technical debt research has been proficient in suggesting identification, tracking, and governance solutions to overcome the technical debt information production issues [12]. The problem is that while solutions have been proposed and trialed on various software contexts, no prior research has properly investigated the whole software context space. That is, identifying and classifying where and how technical debt exists and how does it propagate intra- and inter-software-contexts. This higher level structure may be described in some studies as the concept of technical debt interest and its accumulation, but it has not been explicitly examined; being less important to the relevant studies' goals. Arguably, however, in order to make technical debt management applicable, the various solutions must function together, and in this the enabling factor is technical debt propagation.

Today, the software projects that plug into social media services through APIs (Application Programming Interface) are an exemplar field of software context versatility. Updates to these APIs, invoked by their external authors, indicate sources of technical debt accumulation and propagation in their clients', often business critical, software. Mining Software Repositories (MSR) for the clients that are subject to these updates enables studying the software context space to address the cap in technical debt propagation knowledge.

In the 1980s software applications were relatively simple and they were delivered as is. They were relatively bug free and needed no updates. Once an application was released, any existing technical debt was outside the organization's control. As software grew increasingly complex, especially with the emergence of the Internet in the 1990s, bigger applications were released with more issues remaining. The practise eventually turned out having regularly released patches as a norm, as they were also easily distributed through the net. Technical debt was feasible and also realized. Now, in the 2010s we have complex applications that not only utilize third party libraries, but also third party services through APIs. There are regular updates to the libraries and the APIs, as well as to the client applications themselves. These all are sources of technical debt. Further, as previously shown [6], a singular technical debt instance rarely limits to a single software development component but rather spans over multiple (e.g., design,

implementation, and testing), making the emerging debt even more cumbersome to track.

Our intention is to understand the technical debt propagation context by investigating the latest trends: use of external APIs and especially those of social media services. The paper is structured as follows: we begin in Section 2 by reviewing the background. Section 3 builds on this and introduces our technical debt propagation research objectives. We introduce our approach to overcome the objectives and initial results in Section 4. The concluding remarks appear in Section 6.

2 Background

We will introduce here related work regarding technical debt, propagation in the software context, and APIs. Whilst defining core concepts for the article's foundation, empirical work is also visited so as to further understand the state of current research.

2.1 Technical Debt and Its Propagation

The term "technical debt" was initially coined by Ward Cunningham [2]. In his experience report, releasing code was paralleled to going into debt: trade-offs are made in the software project to meet a deadline, and these trade-offs can be considered debt that should be paid off when resources permit. Until the debt is paid off, it will incur interest payments—that is, later work in the project must accommodate the inoptimalities resulting from the trade-offs. This description has remained applicable to these days. Later revisits to the definition have mainly captured dimensions that further explain the role of the debt in the project: McConnell [13] provides a definition for intentional and unintentional technical debt, while Brown et al. [1] give a further description of the debt's effects via reflection to the financial domain and discussion on the resolution probability.

Firstly, McConnell [13] provided a definition for the intentionality behind the debt: intentional debt is a trade-off made whilst fully aware of its consequences, an investment with an expected return. Unintentional debt on the other hand is accumulated due to, for example, lack of knowledge. This type is a cause for concern as it remains unmanaged until discovered. Secondly, Brown et al. [1] gave a further description of the debt's effects via reflection to the financial domain: the earlier trade-offs accumulate interests payments manifesting as increased future costs, and trained decisions should evaluate if paying the interest is more profitable over reducing the loan via refactoring. Differing from the financial domain, here, the debt's interest has a probability that captures if the trade-off will have visible effects on future development: debt within a software artifact that will not be visited has a realization probability of zero.

Management of technical debt requires that we are capable of identifying and tracking the trade-offs, the atomic instances, that form the debt for a project.

Without this information readily available, trained decision regarding the debt's governance cannot be made [16]. The software context, however, makes the identification, and especially, the tracking an arduous task: instances of technical debt can span over multiple development phases and the most affected part is the software implementation [6] which arguably grows exponentially complex in the future through various abstraction layers and techniques. Nevertheless, the tracking should be able to follow a technical debt instance in this context.

From the latest systematic mapping study on technical debt [12] we can see that several solutions for tracking technical debt are available. However, we also observe (see Figure 10 in [12]) that there are areas in the software development context that are not covered by any solution; whilst most of the solutions cover sub-contexts focusing on predefined environments and specific parts of the software life-cycle. Furthermore, from Kruchten et al. [10] and Izurieta et al. [7] we can see that the causes for technical debt are various and they can be described using various characteristics. We consider all these findings indicative of the multiformity of the context of technical debt in software projects. Thus, in addition to searching for solutions in this context, technical debt research should pursue mapping the full context space and an understanding of technical debt's value in it.

Lastly, we note that technical debt tracking is the process of indicating technical debt propagation in the software context. To this end, the authors identify only the work by McGregor et al. [14] to explicitly address this issue. Here, considering mainly the software implementation, they note that technical debt for a new software asset is affected by the technical debt in relied upon assets, the amount of abstraction layers may diminish the amount of technical debt that propagates, and, in another scenario, rather than being directly accumulated from integrated assets, the technical debt has an indirect effect on the asset's users—for example, by making adoption more difficult.

2.2 Software Change Analysis

What is pursued herein is a better understanding of the context of technical debt propagation in software. We argue that *software change* should be considered the fundamental unit for this. Something that Schmid [15] also considered core to technical debt modelling during software evolution. Capturing software changes and distinguishing between technical debt inclined and other changes (that is, changes using information relatable to technical debt properties described by Brown et al. [1] and discussed in Section 2.1, and changes with no such properties) would allow non-restricted observation of technical debt in the full software context. Identifying software change retrospectively for projects corresponds to Mining Software Repositories (MSR).

Kagdi et al. [8] produce a taxonomy on MSR techniques, defining software change as *"the addition, deletion, or modification of any software artifact such that it alters, or requires amendment of, the original assumptions of the subject system."* Here, a source code change is indicated as the fundamental unit for

software evolution, but as the causes [10, 7] and the manifestations [6] for technical debt do not limit to the implementation, we adopt *software change* as the fundamental unit.

In this work, the mining efforts focus on large open-source, social-networking-enabled, repositories in order to maximally cover the diversity of software change. Tsay et al. [18] note that in GitHub handling of pull-requests is affected by social factors: highly discussed requests enjoy a lower acceptance rate, while submitters relations to—especially the manager of— the accepting project increases acceptance; this is supported also by [3]. Kalliamvakou et al. [9] survey GitHub as a MSR target. They conclude that the repository gives solid data on basic project properties, such as program language use, but synthesizing more abstract conclusions requires careful assessment. The main cause for concern here is GitHub's utilization as infrastructure for personal projects. This form of usage vastly deviates from others. To counter this bias, Kalliamvakou et al. [9] suggest considering only projects with more than two authors and demonstrated activity in both commit and pull requests.

3 Seeking Technical Debt Knowledge

In the following we address our ongoing technical debt propagation research on two distinct levels: the inter-dependency effects at the software artifact level and the longitudinal effects at the project level.

3.1 Inter-Dependency Effects within Software Artifacts

As discussed in Section 2, a multitude of solutions exist for both identifying and tracking technical debt. However, most of the solutions are intended for pre-defined software development contexts; for example, limiting their use to a specific sub-set of implementation techniques and herein, during continued software development, to certain mechanisms for technical debt propagation.

However, the ability to produce exhaustive technical debt information requires that all possibilities for technical debt propagation are acknowledged. We postulate, based on the properties of technical debt identified by Brown et al. [1] and to the average cover of single technical debt instances queried by Holvitie et al. [6], that the propagation "stream" for technical debt is capable of leaving the current host technique and merging into others. This is indicative of several sub-areas within technical debt research.

Foremost research area for technical debt propagation in software artifacts, is (1) to show that technical debt propagates between software components that can exist in external and independent projects and be implemented using different technologies. The interest and even the whole initial debt can be created in an external, but linked project that is worked by another team. The works referred here do not dispute this information, and may even implicitly assume this, but it is important to recognize this phenomenon explicitly and have quantitative research conducted on it to indisputably point it out.

Second research area, partially reliant on the first, is (2) to accumulate a documentation that describes the possible ways in which technical debt can propagate. Preferably, this would be a taxonomy capturing the unique propagation channels for technical debt. Finally, in order to enable information delivery for technical debt management purposes, (3) the channel descriptions must be enriched with information regarding technical debt value accumulation for all unique accounts of propagation. This would enable, possibly automated, technical debt information maintenance as the taxonomy is capable of tracking and valuating technical debt through out the software project.

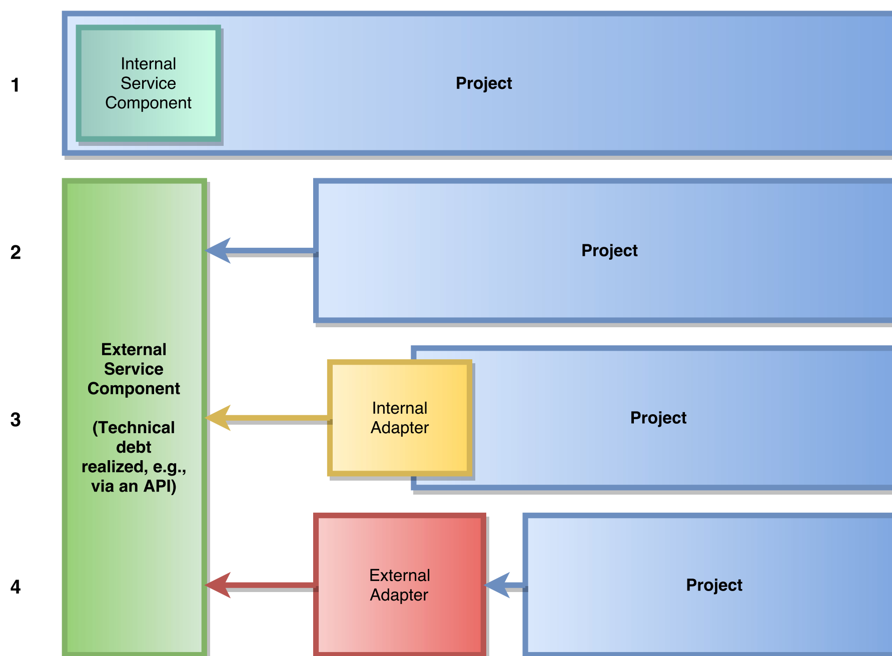

Fig. 1: Coarse classification for different *chains of projects* (COP)

3.2 The Chain of Projects

One way to identify the propagation of technical debt is to make longitudinal studies of increased debt in different phases of a project and connect them with the root causes. Technical debt can be identified as matters, such as discovered vulnerabilities, updates, and feature discontinuation in systems related to the project. Also, adding a new feature in a utilized external service API may cause technical debt when the project customer wants the new feature implemented in the project. We can identify different propagation paths by following how such

an event causes extra work in the *chain of projects* (COP) that are all linked with each other.

If an API is not interfaced directly but through a third party library, it may be that the customer is not happy to wait until the library is updated with the new feature. This will cause the project debt to be paid by implementing this new feature quickly with an internal solution. This will become a new kind of a debt, from the opposite end of the COP, when the referred library is finally updated. Here, the internal solution becomes legacy and requires refactorization into a solution that utilizes the library again, for example, in accordance to the coding conventions followed by the programming team.

There are cross waves moving back and forth in the COP from the root cause, through the library, to the end of chain application. These can be tracked by following the amount of increased work in each area.

Figure 1 demonstrates a sample classification for COPs. Here, case **1** demonstrates a monolith project that has internally implemented services with no outside dependencies. This is a classical, and probably the most studied, scenario for technical debt management, where the debt is only internally caused, felt, and managed. Cases **2** through **4** depict more modern scenarios, where the projects depend on external service providers. In case **2**, the project has a direct dependency to the service and adapts explicitly and directly as invoked by the service. A slightly dampened version, but still fully managed by the project organization is presented in case **3**, where the project, possibly alongside with the organization's other projects, uses an internally produced adapter to access the service. Hence, the project itself does not directly feel changes in the external service, but adaptation to them is still managed internally. Finally, in case **4** the project uses an external adapter to access the service. The external adapter generally serves a broader range of projects and hence is not customized for the needs of specific projects. On the other hand, external adapters tend to retain compatibility as long as possible which dampens change speeds invoked by the external service.

The classification in Figure 1 is especially important from the viewpoint of distinguishing between the "noisy" and the technical debt inclined software changes, as the monolith projects of similar size can be used as the baseline when studying how the external service invokes and propagates technical debt. Further, as per the previous description, it can be expected that the invoked technical debt will propagate quicker in the directly dependent cases than in the indirect cases **2** to **4**.

4 Exploiting Open-Source Projects

Exploiting open source code repositories enables us to make longitudinal surveys of the history. The GitHub code repository service [1] appears as a treasure trove for this kind of research. We can take a project from GitHub, and we can find for it, neatly logged, each change and its date with great detail.

[1] See `https://github.com/`

GitHub gives an open access to several different projects. However, there is also an option of hosting private projects for premium users as mentioned in Section 2.2. With only the public access to the repositories, the sample is likely to be biased. This means that traditionally non-disclosed for-profit projects cannot be found in GitHub like this, which entails that a lot of professional work is not covered by this study. However, it can be argued that functionality is delivered via the same technologies in closed-source projects.

Furthermore, regarding mapping the *software change* (as discussed in Section 2), the GitHub API gives an easy access to byte-wise size of source files and line-wise size of code change per commit. Through this we have the scale of the whole project in bytes, but the scale of changes in lines of code. Optimally both variables would be measured identically, but we can only rely on these two measures being sufficiently comparable. The only other option would be to go through the source files and count the line breaks outside the GitHub API support.

As elaborated in Section 3, we want to observe the propagation of technical debt on both at the software project and the software artifact levels, and with as little constrain as possible so as to capture the propagation context as complete as possible. Herein, we face the problem of how to identify technical debt in a highly diverse setting, and this is the reason why we emphasize the novelty of researching open-source social-networking-enabled projects.

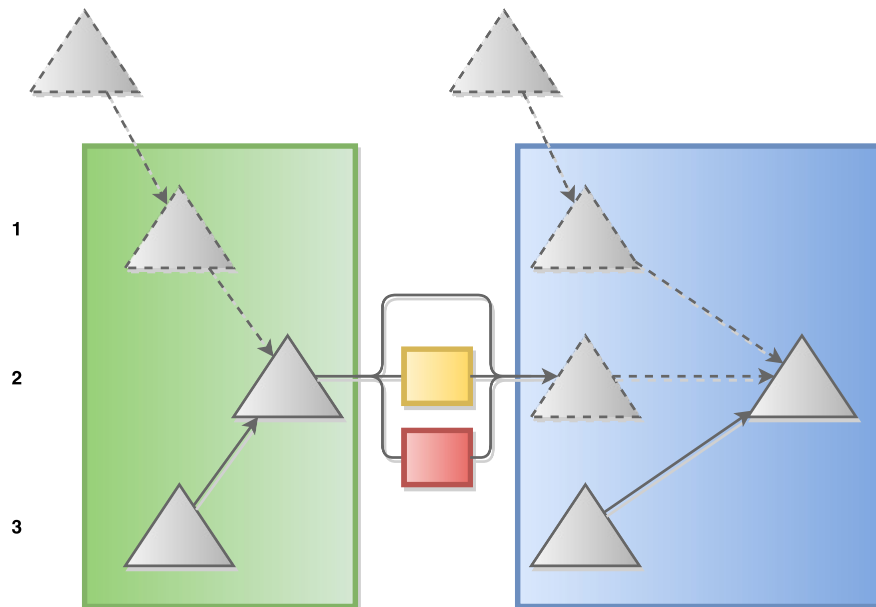

Fig. 2: Coarse classification for technical debt accumulation in projects with dependencies to external services

Figure 2 captures the different technical debt accumulation classes for projects with dependencies to external services. Case **3** depicts the most common situation in which the project accumulates technical debt that realizes at a certain point in time. In case **1** factors external to the component and its development invoke technical debt, and it may realize and invoke management needs at a point in time. In case **2** technical debt has realized (its interest probability is one, or a decision to remove the debt has been made) and it affects the project. In this scenario, the debt will propagate onwards, directly or through intermediaries, and accumulate in dependants. Accumulation channels are addressed in Figure 3.

The classification in Figure 2 is important for distinguishing technical debt inclined software change, as we must be able to distinguish between invoked change (case **2**) and internally accumulated debt (case **1** and **3**). This is because the monolith projects (see Figure 1) are able to internally accumulate technical debt, and we must form the baseline whilst aware of this.

In addition to source code, open-source projects provide access to documentation and other descriptors. Of these, the social media enabled ones form a set of projects that share a joint technical debt inducer: the social media APIs. These APIs provide business critical functionality for the projects, and every time they change, it causes several changes for their clients. Due to the massive adoption of social media services, their APIs (e.g., the Facebook Graph API [2] and the Google OpenID API [3]) integrate into and affect a vast amount of projects. This diverse collection of technologies, which all connect to the APIs that now cause changes for them, unveils a unique opportunity for technical debt research. As the changes propagate through various different technologies, they demonstrate a variety of technical debt propagation paths. Whilst our survey on to the social media involved open-source does not capture the full propagation space, particularly, propagation to business processes, it does yield a formidable library for the propagation of technical debt in delivered software and its supporting structures. Considering that usually this corresponds to the projects' delivered value, research should have a special interest to it.

Figure 3 demonstrates two channels, from a plethora of foreseeable options, through which technical debt can propagate and accumulate in new components. The upper channel captures a more problematic propagation method, in which no explicit dependency exists. In this, accumulated technical debt in the form of incomplete documentation causes a misunderstanding in a conceptualization phase of software development and leads to a complex component design. The lower channel demonstrates an explicit channel, where an interface change is felt in the dependent project as component disconnection. For example, a referred class is renamed in the service due to which the client can not access it in the original fashion. This leads to an erroneous implementation state in the dependent and undoubtedly invokes reparation efforts. In our MSR of open-source projects, over going both the human-produced messages and the automatically

[2] See `http://graph.facebook.com/`
[3] See `https://profiles.google.com/`

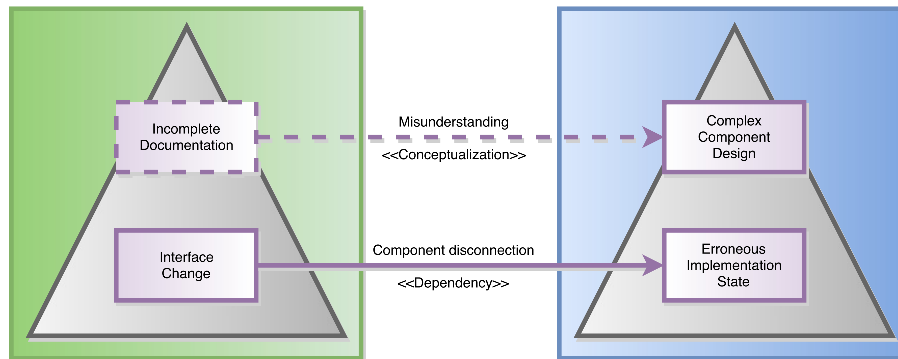

Fig. 3: Two examples of technical debt propagation channels

identified changes should reveal instances that fit both channels shown in Figure 3, but due to its implicit nature, identification of cases in the upper channel will be difficult.

4.1 Study Approach

We use the GitHub API through *PyGithub/PyGithub* library [4]. Our crawler is a Python program [5] designed to crawl through all commits of a given project and report, for each commit, the date it was committed, the amount of changes (as the amount of added and removed rows), and the changed files. As such, our crawler is in itself an end part of a COP.

For an initial test of concept we chose Google's closing of OpenID 2.0 service on April 20th 2015 [4] as a source of technical debt. We made a manual search in GitHub and discovered two Java projects which had closed issues mentioning Google closing the service. One was the Passport-based User Authentication system for *sails.js* applications—GitHub repository *tjwebb/sails-auth*. The other was a Grails website that provides information about festivals—GitHub repository *domurtag/festivals*. For a control project we selected another Java project that was similarly a user authentication system for *sails.js* as *sails-auth*, but did not appear to be involved with Google services—GitHub repository *waterlock/waterlock*

4.2 Initial Results

Our analysis produced the graphs shown in Figure 4.The blue colour is used for *sails-auth*, red for *festivals* and cyan for *waterlock*. The X-axis marks the time. The dots denote the amount of changes in a commit. The bars denote commits

[4] see `https://github.com/PyGithub/PyGithub` and in similar fashion for the other mentioned repositories as well
[5] GitHub repository *tomibgt/GitHubResearchDataMiner*

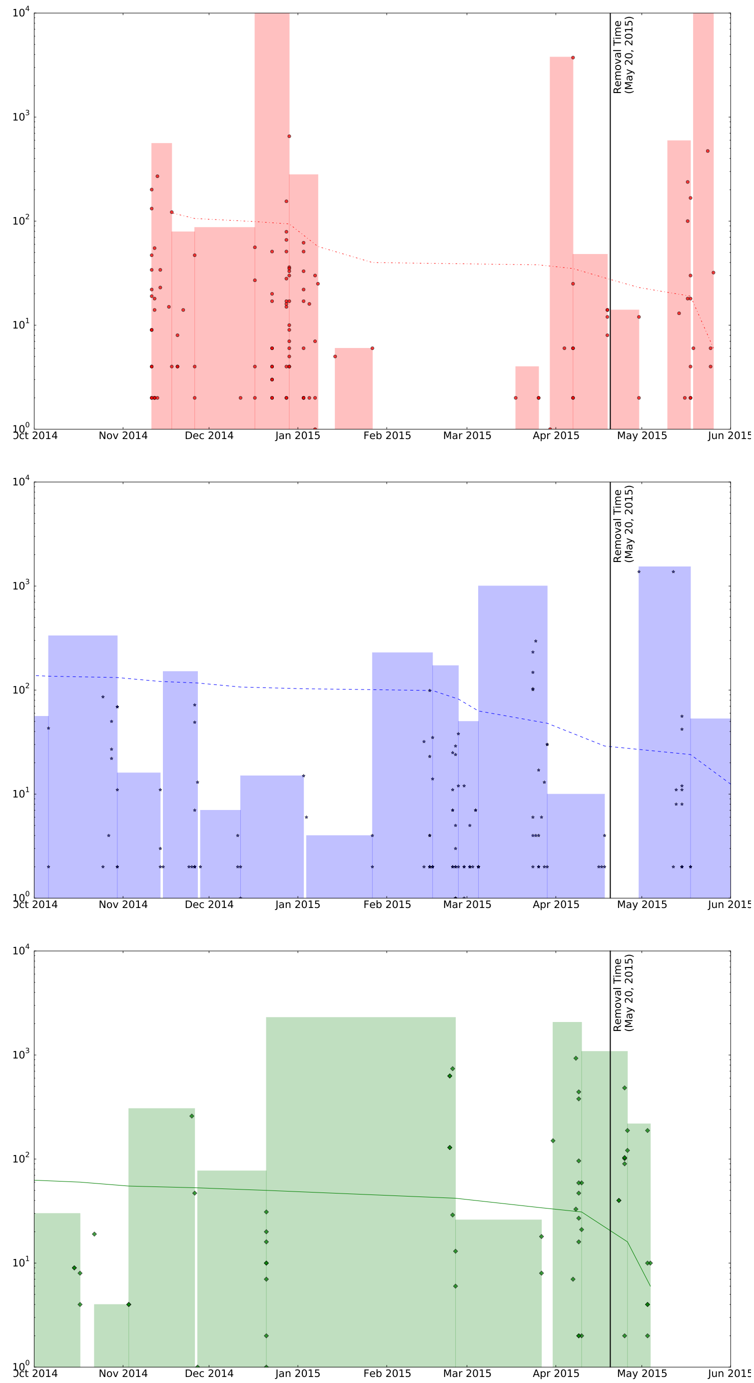

Fig. 4: Commit amount analysis for the three selected GitHub repositories

for a time period at least a week long. The lines denote commit frequency for previous time interval of at least a week. Finally, on the graph is marked the date-of-interest, April 20th 2015.

The lines show a general decline, which would appear to indicate that as a project progresses, less and less changes are made for it. Note that the Y-axis is logarithmic, which makes the lines curve down, instead of appearing linear.

It would appear to be supporting our hypothesis, where, after the marked date, *sails-auth* and *festivals* show decrease in the decline, unlike the control project *waterlock*. With only three projects and without more precise investigation we can not, of course, claim this to be strong evidence, but it is enough to encourage us in continuing with this approach.

Table 1: Commits for the *festival* repository file `show.gsp` around *Removal Time*

Time	Add	Remove	Delta
5/18/2015	0	2	-2
5/18/2015	7	12	-5
4/19/2015	**3**	**1**	**2**
4/19/2015	**14**	**0**	**14**
12/29/2014	11	6	5
12/29/2014	2	2	0
12/29/2014	2	1	1
12/28/2014	7	3	4
12/28/2014	8	14	-6

Table 2: Technique-wise recorded changes around *Removal Time*

Type	Add	Remove	Delta
`js`	86	2	84
`gsp`	35	3	32
`jpg`	.	.	.

With moderate work, the analyser can be modified to point out the files where there has been increasing changes in the commits correlating to the investigated events. (See Tables 1 and 2.) Looking into the changes made into these files should help us to analyse further the effort put by the programmers to pay the specific technical debt. Also, it should be possible to follow the wave of changes throughout the COP and analyse the propagation of the debt and the involved work and communication.

5 Applicability and Limitations

The aforedescribed approach is limited by certain factors which we would like to address here. Firstly, we described this method as a possibility to explore the complete software context space, but the study design suggests using service calls to, especially social media, APIs and libraries as the method. It can be assumed, as previously discussed, that this approach does not capture all possible varieties of *software change* (see 2.2). This is a foreseeable data limitation even though it can be argued that the volume of captured *changes* would produce a representative set for analysis; accumulating enough assurance to allow abstraction to non-captured context areas.

Second, there are limitations potentially affecting the identification of technical debt instances. We discussed the technical debt properties which can be used to associate a *software change* with managing technical debt. While this set of properties currently accounts the state-of-the-art from technical debt research, if not exhaustive, the properties may lead to missing particular sub-classes of technical debt. Approach discussed in the following paragraph, can be considered a partial remedy to this.

Finally, foreseeable limitations may also affect the tracking of technical debt instances. As a premise for tracking, [6] showed the instances' ability to span over multiple components. Modelling of the *chain of projects* was introduced as the method to allow capturing this behaviour. The current classification presented in Figure 1 considers one dimension for the COPs—presumed to be the most dominant. This classification can be a limiting factor, especially in large hybrid COP projects, but we argue that this can be countered by iteratively exploring more dimensions for the COPs until all technical debt inclined changes have been successfully associated to the technical debt instances.

Overcoming the limitations and achieving the study's objectives, there is a number of applications for the results (discussed in Section 3.1). Firstly, demonstrating technical debt's ability to propagate, almost boundlessly, between software projects and artifacts should fuel the apparent paradigm shift in software life-cycle management where the inter-connectivity of software project entities carries increased value. Second, documenting the ways in which technical debt can propagate should provide an interface for integrating knowledge from other research domains to enhance technical debt management by for example applying financial models for technical debt strategization. Lastly, associating the documentation's technical debt propagation channels with information regarding their value accumulation allows automated tooling approaches to be introduced, but also makes technical debt an integral and explicit component of the software project's value production and its assessment.

6 Conclusions and Future Work

With similar studies in the future, using different event markers, it is possible to map the propagation of technical debt by observing the amount of increased work caused by different causes of technical debt. It is possible to observe who pays the technical debt and how it is propagated from the original cause (e.g., a change in a fundamental library used by many projects) through facade libraries and components to the final applications.

In an effort to efficiently analyse the propagation of technical debt through propagation channels, a taxonomy of projects in GitHub should be created to help characterize and predict the characteristics of the projects. To this end, and to achieve the goals stated herein, we have analyzed over twenty-eight thousand projects from GitHub and have successfully identified a number of projects with references to suitable external services. According to Lambe [11], even taxonomies founded on criteria that do not stand all scrutiny, can allow for reliable

predictions and descriptions of characteristics of new members of the taxonomy based on very little information. A well created taxonomy combined with our expected mining results should help us identify different propagation channels within the projects without even analysing them at the code level. Should we find two or more clusters of different kinds of change behaviour within a single taxonomy class, it could suggest that the propagation channels between these clusters differ from each other.

There can, of course, be other causes to variance within a class. For example, it would be beneficial to have the information of the process maturity level for each project team. This kind of information would be significant in understanding the project's sensitivity to external changes and the general preparedness and carefulness in the design. [17]

Such work would provide us with a better understanding of the economy of technical debt, which again would help us give good estimates on the actual costs of applying, for example, social media APIs in an application system and compare it with the projected benefits and income. It would help in answering the question: would applying certain features increase the revenue from the service.

Acknowledgment

J. Holvitie is supported by the Nokia Foundation Scholarship and the Finnish Foundation for Technology Promotion, the Ulla Tuominen Foundation, and the Finnish Science Foundation for Economics and Technology grants.

References

1. Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., et al.: Managing technical debt in software-reliant systems. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research. pp. 47–52. ACM (2010)
2. Cunningham, W.: The WyCash portfolio management system. In: Proceedings Addendum for Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). pp. 29–30. No. 22 (1992)
3. Dabbish, L., Stuart, C., Tsay, J., Herbsleb, J.: Social coding in github: transparency and collaboration in an open software repository. In: Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work. pp. 1277–1286. ACM (2012)
4. Google Developers: Migrating to google sign-in (2015), https://developers.google.com/identity/sign-in/auth-migration
5. Guo, Y., Seaman, C., Gomes, R., Cavalcanti, A., Tonin, G., Da Silva, F., Santos, A., Siebra, C.: Tracking technical debt - an exploratory case study. In: 27th IEEE International Conference on Software Maintenance (ICSM). pp. 528–531. IEEE (2011)
6. Holvitie, J., Leppänen, V., Hyrynsalmi, S.: Technical debt and the effect of agile software development practices on it-an industry practitioner survey. In: Sixth International Workshop on Managing Technical Debt (MTD). pp. 35–42. IEEE (2014)

7. Izurieta, C., Vetrò, A., Zazworka, N., Cai, Y., Seaman, C., Shull, F.: Organizing the technical debt landscape. In: Third International Workshop on Managing Technical Debt (MTD). pp. 23–26. IEEE (2012)
8. Kagdi, H., Collard, M.L., Maletic, J.I.: A survey and taxonomy of approaches for mining software repositories in the context of software evolution. Journal of Software Maintenance and Evolution: Research and Practice 19(2), 77–131 (2007)
9. Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D.: The promises and perils of mining github. In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 92–101. ACM (2014)
10. Kruchten, P., Nord, R.L., Ozkaya, I.: Technical debt: From metaphor to theory and practice. IEEE Software 29(6) (2012)
11. Lambe, P.: Organising knowledge: taxonomies, knowledge and organisational effectiveness. Chandos Publishing (2007)
12. Li, Z., Avgeriou, P., Liang, P.: A systematic mapping study on technical debt and its management. Journal of Systems and Software 101, 193–220 (2015)
13. McConnell, S.: Technical debt. 10x Software Development Blog,(Nov 2007). Construx Conversations. URL= http://blogs. construx. com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx (2007)
14. McGregor, J., Monteith, J., Zhang, J.: Technical debt aggregation in ecosystems. In: Third International Workshop on Managing Technical Debt (MTD). pp. 27–30. IEEE (2012)
15. Schmid, K.: A formal approach to technical debt decision making. In: Proceedings of the 9th International ACM SIGSOFT Conference on Quality of Software Architectures. pp. 153–162. ACM (2013)
16. Seaman, C., Guo, Y., Izurieta, C., Cai, Y., Zazworka, N., Shull, F., Vetrò, A.: Using technical debt data in decision making: Potential decision approaches. In: Third International Workshop on Managing Technical Debt (MTD). pp. 45–48. IEEE (2012)
17. Suenson, E.: How Computer Programmers Work – Understanding Software Development in Practise. Ph.D. thesis, Turku Centre for Computer Science (2015)
18. Tsay, J., Dabbish, L., Herbsleb, J.: Influence of social and technical factors for evaluating contribution in github. In: Proceedings of the 36th International Conference on Software Engineering. pp. 356–366. ACM (2014)