Software Implementation Knowledge Management with Technical Debt and Network Analysis

Johannes Holvitie
TUCS - Turku Centre for Computer Science &
Department of Information Technology, University of Turku
Turku, Finland
jjholv@utu.fi

***Abstract*—Modern, fast-phased, iterative and incremental software development constantly struggles with limited resources and a plethora of frequently changing requirements. This environment often requires the development projects to intentionally — for example through implementing quick-and-dirty — or unintentionally — for example through misinterpretation of requirements — deviate from the optimal product state. While most of the deviation is caught through practices like customer reviews, the remainder stays hidden in the product. The undocumented remainder is difficult to remove, it expands uncontrollably and it negatively affects development as deviations are unexpectedly encountered and overcome. The term technical debt describes this process of accumulating hidden work. Management of technical debt can be expected to be a major factor in software development efficiency and sustainability and as such it should be an integral part of the software implementation's knowledge management. In addition to being difficult to capture, the continuous evolution of the implementation makes maintenance of gained information a challenge. This paper discusses applying technical debt management for software implementations including the entry points for knowledge discovery, network analysis for overcoming the maintenance challenges as well as the pursued outcomes.**

***Keywords*—technical debt management, network analysis, program visualization, refactorization.**

I. Introduction and Problem Statement

Modern software development methods deal with increasing complexity and frequently changing requirements by decreasing increment size and shortening iteration times. These, often Agile or Lean, methods try to imply control over the product in requiring that all releases meet the definition of done. For the client this definition encompasses the product's perceivable capability to fulfill set requirements. The developer's definition is a super-set of this in also capturing how the requirements are implemented. This dualism of definitions allows iterations to deliver increments which meet the former but not the latter, that is they are perceived done while actually being incomplete.

This is a very common phenomenon in software development resulting from developer actions that are either intentional, like relaxing quality requirements when pressed for time, or unintentional, like making uneducated design decisions. Since the opposing party, the client, does not perceive these inconsistencies it can not pursue their completion. Rather, the developing organization is implicitly assumed to manage them. But, when faced with the choice of allocating resources to implement perceivable requirements or fixing unperceivable inconsistencies, the choice of the former can be expected — especially when no information supporting the latter is readily available. These decisions and the discussed phenomenon respectively accumulate new and increase existing technical debt.

Technical debt considers the deviation between the current and optimal product state and it is problematic due to a number of reasons. Firstly, as its emergence is the end result of a rather obfuscated process, it rarely gets documented. Low visibility makes management difficult and it hides problem severity. Technical debt has the ability to accumulate super-linearly as software solutions are built by depending onto earlier, unsuccessfully implemented, components. And even if technical debt gets indicated the evolving implementation promptly degrades this information.

Due to the multiple issues exposed by technical debt, software development projects require mechanisms that are able to efficiently capture, track and govern it. The mechanisms need to accomplish these while retaining development characteristics like agility. The mechanisms should also accommodate the fact that some instances of technical debt exist due to informed decisions.

This paper discusses research to overcome these challenges and it is structured as follows. Section II discusses motivation for this research in more detail while Section III derives the main hypothesis and work objectives. In Section IV work related to technical debt and its management is presented with accompanying terminology. Section V proceeds to layout the research plan to accomplish set objectives while Section VI discusses the expected outcomes. Section VII concludes this paper.

II. Motivation

There exist mechanisms, like bug reports and unit tests, that are able to capture the deviation between the current and optimal software implementation states, but none of these consider the dynamic aspects of software development. The Agile Manifesto's [1] 10^{th} principle states that "*software development should pursue simplicity - the art of maximizing amount of work not done*". In respect to this, it is not ideal to immediately fix every encountered non-conformality, since this may amount to a lot of unnecessary work, but at the same time total ignorance can not be exercised as some of the non-conformalities may end up causing much more work than the initial fix would have required.

Technical debt is about acknowledging these dynamic aspects in order to optimally manage the singular deviations. Technical debt management pursues the formation of technical

debt instances [2], units of deviation that share the same context and are thus governable by a single person or team. This enables documenting the impact probability for each instance, capturing the chances that it causes additional work during continued development. Further, the instances allow estimating accumulated impact for them as they become more coupled to the system. The estimates can then be fed back to project decision making in order to make technical debt — or optimized deviation — management a concrete part of it.

As technical debt management does not concern with different types of non-conformalities it is possible to superimpose it on top of existing mechanisms. Additionally, this allows considering contexts composed from elements at different implementation abstraction levels, in different project artifacts or purely subjective observations as manageable instances.

While technical debt exposes these dynamic characteristics, it does not expose a model or a mechanism for maintaining the captured information. Thus, the greatest challenge in introducing technical debt management to software development lies in deriving feasible maintenance approaches. In technical debt management research (see Section IV), one discussed approach to this considers manual updating a possibility. The research presented in this paper argues that software development, especially for the implementation artifact, is carried out based on a number of underlying models for which network analysis can reveal static update models. A hybrid approach of initial manual input and continued updates through a static model could retain a high level of accuracy while supporting automated maintenance.

III. Hypothesis and Objectives

Research work discussed in this paper facilitates exploring a general hypothesis through a more specific one. As technical debt management pursues optimized governance of non-conformalities in software development, a general hypothesis for this is stated as *technical debt management is a factor in software development efficiency and sustainability*. As the non-conformalities are exposed, made into explicit instances, and input into the decision making processes, the uncertainty about required additional work decreases. This is expected to result in more streamlined development and thus increased development efficiency. Sustainability is increased based on the same grounds: as non-conformalities and their impact on to development are made explicit, the developing organization is provided with means to perceive and control the problem before it becomes unbearable.

The more specific hypothesis, or task, studied here facilitates the general one. It concerns the maintenance challenges in the software implementation context (see Section II) in stating that *software implementation technical debt can be captured and maintained*. Captured via tools that are present when notions about technical debt are made and maintained via a model that takes the subjective notions and incrementally updates them. These increments require a static reference, a model, that dictates granular updates for the structured notions based on observed evolutionary steps in the software implementation.

Three consecutive work objectives facilitate studying the specific hypothesis. The first step concerns *building a tool capable of capturing technical debt instances*, the notions discussed earlier. The second step *builds the static model* by applying network analysis to programming theory, the user inputs and captured technical debt propagation characteristics. The third step combines the tool with the model, inputs the model with structured notions about technical debt and observed evolutionary steps in the implementation *to derive information updates*. The specific hypothesis is studied by determining if the information captured and maintained in the third objective is successful in serving technical debt management approaches. Quantifying enhancements to technical debt management can be seen to further efforts to overcome the general hypothesis.

IV. Related Work and Terminology

In meeting the work objectives discussed in Section III, a number of research contexts are consulted. Building the tool for the first objective, the documentation structure needs to accommodate requirements set by technical debt management. The update model derived in the second objective bases on identifying and separating technical debt instances from software implementations and applying network analysis to capture shared and unique attributes. In the following sections essential research and terminology are introduced for these areas.

A. Technical Debt

Technical debt is a term that was first coined by Ward Cunningham in his technical report to OOPSLA'92 [3]. Updated definitions have followed this for example in Seaman et al. [4] as well as in Brown et al. [5]. A consensus between these definitions is that technical debt is based on a principal on top of which interest is paid — similarly to its financial counterpart. The principal captures the original, deliberately or indeliberately, incurred non-conformality. The size of the principal is equal to the work required to provide an optimal solution for it. The interest is relative to the amount of adaptation the surrounding system has committed to when the principal has become more coupled to it. Seaman et al. [2] expand the definition of interest by stating that interest is an impact value coupled with impact probability. This probability communicates about the chances of development work being continued in the interest's implementation area. For example, if no further development is carried out in this area, the probability of needing to pay the interest is zero.

Technical Debt Management Framework (TDMF) introduced in Guo et al. [6] is a software development method independent approach for integrating technical debt management into project decision making. The two first steps of this three part approach build a Technical Debt List (TDL) which captures the technical debt instances for a software project. The instances are captured as Technical Debt Items (TDI) following a documentation structure dictated by the TDMF. Most notably, this structure requires that for each TDI, explicit impact size and interest probability are estimated. As the TMDF enables integration of and information interchange between other management approaches, the tool pursued for the first work objective will adhere to it.

McGregor et al. [7] were the first to discuss about technical debt's propagation in software implementations. In their work

they hypothesize about two concurrent mechanisms. The first states that "*technical debt for a newly created asset is the sum of the technical debt incurred by the decisions during development of the asset and some amount based on the quality of the assets integrated into its implementation*". During introduction of this mechanism, they communicate about technical debt's ability to diminish as a result to increases in interface layers. The second mechanism describes that "*the technical debt of an asset is not directly incurred by integrating an asset in object code form, but there is an indirect effect on the user of the asset*".

B. Capturing Information from Software Implementations

Software implementations are structures in which components rely onto others in order to fulfill their functionality. Capturing these structures as sparse matrices either dynamically, from the program execution trace, or statically, from data mining the source code, allows the application of network analysis approaches to further analyse and understand them.

Link structure algorithms produce either global or query-specific importance rankings for matrix elements. The PageRank algorithm by Page and Brin [8] has received most of the attention regarding producing global rankings for implementation components. In amongst others [9]–[11] have introduced their own adaptations of the algorithm for use in analysing software implementations. They have acknowledged the possibilities in using this approach for example in valuing components for software impact analysis as well as relating implementation elements for feature location. An important notion from these works is that even the slightest changes in building the matrix result in large fluctuations in the received rankings.

Program visualization is, in some regards, a more humane approach to link structure analysis. Interesting observations can be quickly made even for a complex context if it is possible to produce a visualization for it and especially to highlight structures of interest within it. Caserta and Zendra present a program visualization classification in [12]. In this they state that *graph* as an approach is a forerunner for architecture level visualizations that focus on highlighting relationships. Noack and Lewerentzes discuss a requirements space for visualizing software implementations with graphs [13]. They formalize it as the three degrees of clustering, hierarchicalness and distortion. For approaches interested in exploring directly visible structural relations, all previous degrees should be low.

V. Approach and Challenges

The author's study focuses on capturing and maintaining technical debt information for the software implementation in order to facilitate integration of technical debt management for it. Section III described the three work objectives that were derived to overcome this: creating a tool capable of capturing technical debt instances, a static model to automate maintenance for them and integration of the model with the tool to produce a fully functioning management suite. At the current state, a solution exists for the first objective while research is underway to facilitate the second one. This chapter walks through the three objectives describing the chosen approach, existing work and foreseeable challenges for each.

A. First Objective - Technical Debt Management Tool

The first objective required a tool that was readily available when notions about technical debt were made in the implementation. To overcome this the author co-designed a two-partite tool called *DebtFlag* which is presented in [14]. The first part of this tool (see Fig. 1) is a plug-in for the Eclipse integrated development environment (IDE). The plug-in allows developers to create TDIs by interacting directly with edited implementation elements. The TDIs are bound to revisions and thus are synchronized through the version control system. In addition to ensuring that intrinsic information about the TDI's impact and propagation characteristics is recorded, the plug-in also provides a representation for them. The plug-in introduces an update mechanism that uses the static update model from Section V-B to calculate where given TDIs propagate. Affected implementation elements are highlighted in colors according to given rules. The highlights in both the editor view as well as the content-assist (see Fig. 2) make it virtually impossible to carry on implementation without knowledge about technical debt's presence. This mechanism is one of the first concrete tool solutions to controlling unwanted technical debt propagation.

The second part of this tool is an overview web-application (see Fig. 2). The plug-ins are to communicate TDIs to a

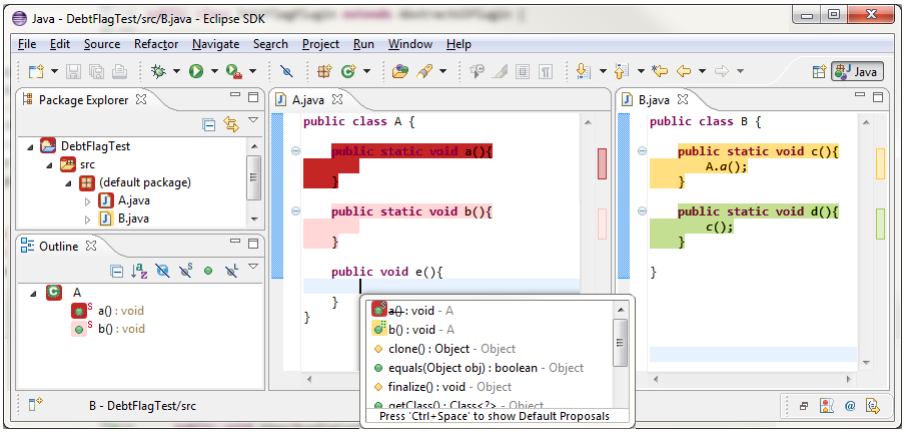

Fig. 1. The *DebtFlag* plug-in managing code highlighting and content-assist cues in the Eclipse IDE

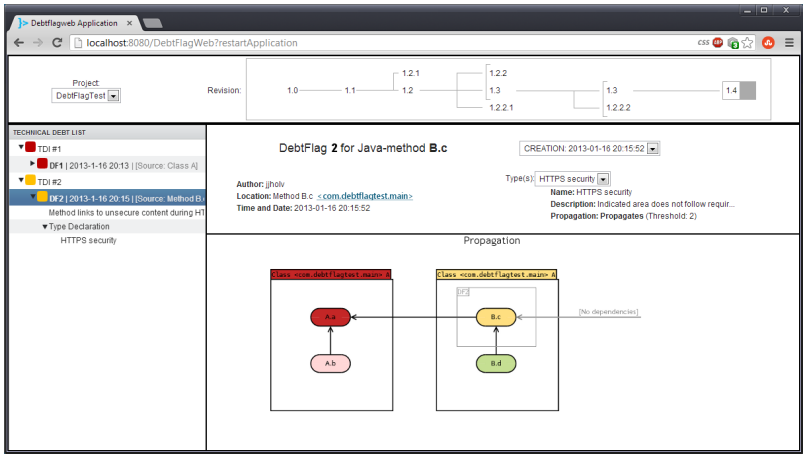

Fig. 2. The *DebtFlag* tool's web interface

database from which the web-application builds project specific TDLs. Basic functionalities, like modifying details for TDIs and observing their history as a function of revision commits, are available through the web-application. The TDLs can be used as is for inter-organization communication but they are also intended to provide an interface for integrating other technical debt management approaches to the projects.

The most formidable challenges regarding the tools use affect application of the captured information. Firstly, maximized efficiency and intuitiveness for the tool's usage tries to encourage developers to capture all technical debt they perceive. Unfortunately, this also presents a way to measure performance for the authoring persons. This would inevitably discourage further use of the tool and negatively affect the approach's viability. The presenting paper [14] discusses certain levels of information hiding as a remedy for this. The other challenge is closely related to this: if authors for TDI entries are hidden and producing them is easy, there is a danger that for TDIs for which the correct action would have been to directly repair them, instead a TDI entry is made. Expediting repairs for small entries through valuing them higher in the static model is discussed as a possible answer to this.

B. Second Objective - Static Update Model

The second work objective concentrates on building a static update model for the *DebtFlag* tool introduced in the previous section. The tool queries the model with indicated locations and expects a (fuzzy) set of elements as output — indicating technical debt expansion for them. This objective is a derivative from the technical debt integration requirement by Seaman et al. [4] which states that software project decision processes need to be accompanied with usable and current information regarding the project's technical debt. Introducing automated maintenance for manual observations allows efficient extension of their applicability thus retaining more information for decision making.

The author has partaken in a number of studies to facilitate exposing the static models. In the first such study we examined the role of dependency propagation in the accumulation of technical debt by conducting a manual retrospective analysis for a large refactorization project [15]. Based on captured relations at the class level, the following observations were made. First, the number of incoming dependencies to an implementation element correlated with the number of propagation paths for technical debt. Second, dependency propagation was the main driver for technical debt accumulation. This was evident from that for a majority of chronological modifications a direct dependency relation was observed.

Third, technical debt diminished due to propagation. This was a result from measuring propagation depths to be smaller than what component dependencies would have allowed for. Fourth, the component's role was a good indicator of the propagation's shape. This was apparent from observing that for technical debt affecting data models the reparations expanded in the system in a shape that was wide but shallow, while for reparations targeting direct functionalities the shape was more focused but deeper reaching. In our yet unpublished journal extension *Examining Technical Debt Accumulation in Software Implementations*, we studied the four observations for another independent data set at a lower, class-member, level. Finding that data supported the observations here as well was perceived as an indicator for technical debt's universal propagation capabilities. As such, the initial static model is required to accommodate at least the four observed characteristics.

To facilitate studying technical debt and its propagation characteristics for large implementations and non-conformality counts, a program visualization approach was designed. We demonstrate this in a yet unpublished study *Illustrating Software Modifiability - Capturing Cohesion and Coupling in a Force-Optimized Graph*. The approach has three consecutive steps to forming the visualization. The first step traverses a source implementation and captures all program elements that are capable of forming direct dependencies in it. The second step forms a graph by presenting the implementation elements as nodes and capturing the direction and frequency of dependency invocation between them as the graph's directed and weighted edges. The third step lays out the graph through force-minimization. In this, the directed weighted edges represent forces and finding a global energy minima for such a system emphasizes those structures that contribute towards it.

Fig. 3 demonstrates applying the visualization approach to the source code of the Eclipse IDE's Debug component. Here,

nodes represent Java interface members and edges capture their invocations. The gray graph is the component's part from the Eclipse's implementation force-minimally laid out. The distinct hub in the upper part is the component's more independent, cohesive and less coupled, event system. The highlighted part in the bottom is the component's bug #148965. The red and green lines indicate dependencies that are outbound and inbound respectively to elements declared for the bug. The highlighting mechanism allows us to quickly inspect how non-conformalities propagate. In this case, the green lines indicate that the root cause for this bug maybe coming from outside the Debug component's implementation.

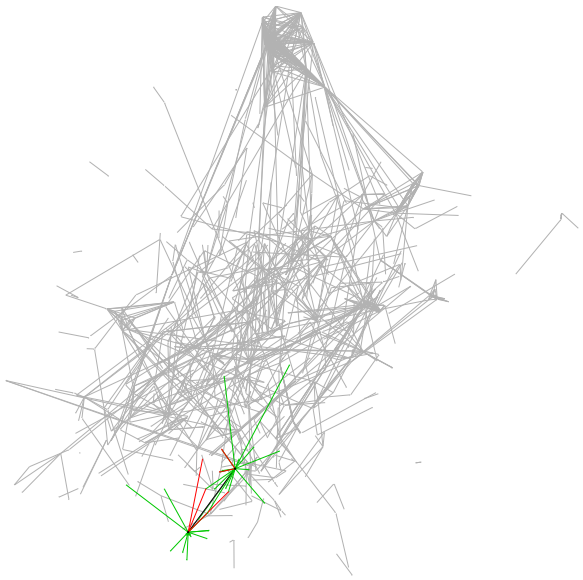

Fig. 3. Presents Eclipse's bug no. 148965 highlighted against its host Debug project graph

Variations of the static update model are currently being trialed to effectively account for the four observations that were described earlier. The aforementioned visualization approach allows to efficiently handle large implementation data sets and to distinguish special characteristics from them. The network analysis approaches discussed in Section IV-B are used on the produced graphs to rank their elements and produce correlation coefficients. Distinguished special characteristics are then extracted as components for the static model using basic functional regression and related machine learning approaches.

There are two perceivable challenges to this work objective. The first is the applicability of the model. A model that covers multiple programming languages will not take into account intrinsic details for each one. This is a trade-off between applicability and accuracy. Additional research needs to balance this as it is important to produce approximations with a minimum of false-positives since these are directly used to produce the technical debt representation. Secondly, the propagation model training sets as well as its inputs may contain speculation. For instance the bug in Fig. 3. The location initially indicated for the bug can be speculative and the true location is only known when it has been fixed and verified as such. This must be taken into account.

C. Third Objective - Maintaining Captured Information

The third and last work objective will take the static update models from the second objective and inputs them to the *DebtFlag* tool from the first objective. As stated, the *DebtFlag* tool exposes an abstract update descriptor and is thus able to accommodate all static update models that inhere to it. Field testing of different models is thus conducted through the *DebtFlag* tool which puts further emphasis on its implementation and user experience quality.

As the third objective corresponds to examining the specific hypothesis introduced in Section III, it will be overcome in two stages. The first stage will take the most promising models, inputs them to the *DebtFlag* and approaches a small number of organizations from varying development contexts. This stage will provide both the tool and the used models with enhancements. Having accommodated them, the second stage will be conducted as a quasi-experiment where the tool is introduced, with the chosen models, to controlled development organizations. For these organizations, their implicit and explicit technical debt management approaches need to be identified so as to be able to observe how the tool affects them.

This objective decompiles into proving the specific hypothesis, which stated that software implementation technical debt can be captured and maintained. Measuring this as project work-in-progress capability increments as well as improvements in estimate confidence comes very close to examining the general hypothesis for technical debt management. The main challenges here are those shared by all controlled experiments. What is expected to be especially challenging is the derivation of valid measuring points and relation of gained results with control groups.

VI. Contribution

The contributions of the presented research come from accomplishing the three work objectives discussed in the previous chapter. As it currently stands, the author has completed the first work objective and is working towards completing the second one. Completion of both the first and second objective is required to pursue the third objective, which constitutes majority of the presented researches' contribution. However, several advances can already be seen and they are discussed in the following.

The *DebtFlag* tool from the first objective is the first in its kind to explicitly pursue technical debt management as part of software development. Close integration with development tools allows the tool to provide an implementation level representation for captured technical debt, which makes technical debt unaware development impossible while also enabling micro-management for singular technical debt instances. This should result in notable efficiency improvements when taking into account technical debt's super-linear accumulation speed.

Capability to explicitly manage implementation components prone to technical debt has lead to further studies in which the *DebtFlag* tool is applied to overcome issues in legacy software development. The ability to produce a "legacy interface" for the project allows to efficiently restrict incoming dependencies to these parts while developers work towards adapting the legacy parts to the rest of the project.

Research to expose properties of technical debt propagation in the second objective allows capturing them in static update models. Capability to automatically update manually made observations during continued development allows extending their applicability. Such a mechanism is required to pursue optimizing defect governance. Additionally, exposing and reporting on found propagation characteristics hopefully leads to further research on the area, possibly emergent to complementary update models.

Finally, integration of the static models with the *DebtFlag* tool in the third objective is expected to produce a fully functioning technical debt management suite. Capturing notions in a way that adheres to the TDMF documentation requirements should produce a medium through which developers can easily communicate about technical debt and its resource requirements to the management. Underlying update model ensures that all information is current and thus applicable in decision making. Adherence to the TDMF's documentation policies also allows this development method independent suite to act as an interface to integrate further technical debt management policies into these projects.

VII. Conclusion

Technical debt captures the uncertainty for a software project and communicates about the effects it causes. On the highest level these can be seen to include declining development efficiency and sustainability. Uncertainty in development calls for reserving more resources in order to overcome possibly encountered issues. This leads to a less streamlined and less efficient process. Having fewer resources available reduces development sustainability. The project becomes more rigid and less capable of accommodating quickly changing requirements. That is, technical debt hides the project's true state and leads to decisions being made disconnected from it: unaware of the project's actual capabilities to overcome requirements and unforeseen risks while ignoring the optimal moments for reductive maintenance.

Research proposed in this paper facilitates introducing technical debt management for software implementations. As they usually constitute majority of the projects' accumulated value, the effects of technical debt and its management are felt the strongest here. Ability to capture and maintain information about software implementations' technical debt does not only allow the introduction of further management approaches but also the introduction of this information to existing approaches so as to make them sensitive to technical debt as well.

Three consecutive work objectives were discussed to achieve this. The first objective called for a tool that allowed subjective notions about technical debt to be captured in a structured manner. The *DebtFlag* tool was introduced as a solution to this. In addition to supporting the TDMF, the *DebtFlag* introduced a novel micro-management approach for technical debt. The second objective finds static models to be used in updating the captured notions. Research towards this is currently underway, having already distinguished a number of general propagation characteristics for technical debt while continued research tries to identify and model the unique characteristics of specific implementation techniques. The third and final objective will then combine the two former ones in order to produce a fully functional technical debt management suite to overcome the matters discussed in the previous paragraph.

The author expects the pursued approach to result in a number of enhancements for software implementation technical debt management with practical applications. Even small advancements should be considered as the iterative and incremental properties of current software development methods multiply the effect in the host project. At the same time, these methods correspond to the research's greatest challenge as technical debt management needs to integrate to them without suppressing their unique characteristics.

References

[1] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries *et al.*, "Manifesto for agile software development," 2001.

[2] C. Seaman and Y. Guo, "Measuring and monitoring technical debt," *Advances in Computers*, vol. 82, pp. 25–46, 2011.

[3] W. Cunningham, "The WyCash portfolio management system," in *Addendum to the proceedings on Object-oriented programming systems, languages, and applications (OOPSLA)*, vol. 18, no. 22, 1992, pp. 29–30.

[4] C. Seaman, Y. Guo, C. Izurieta, Y. Cai, N. Zazworka, F. Shull, and A. Vetrò, "Using technical debt data in decision making: Potential decision approaches," in *Managing Technical Debt (MTD), 2012 Third International Workshop on*. IEEE, 2012, pp. 45–48.

[5] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya *et al.*, "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 2010, pp. 47–52.

[6] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Da Silva, A. Santos, and C. Siebra, "Tracking technical debt - an exploratory case study," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 2011, pp. 528–531.

[7] J. McGregor, J. Monteith, and J. Zhang, "Technical debt aggregation in ecosystems," in *Managing Technical Debt (MTD), 2012 Third International Workshop on*. IEEE, 2012, pp. 27–30.

[8] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web." 1999.

[9] B. Turhan, G. Kocak, and A. Bener, "Software defect prediction using call graph based ranking (cgbr) framework," in *Software Engineering and Advanced Applications, 2008. SEAA'08. 34th Euromicro Conference*. IEEE, 2008, pp. 191–198.

[10] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking significance of software components based on use relations," *Software Engineering, IEEE Transactions on*, vol. 31, no. 3, pp. 213–225, 2005.

[11] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 2012, pp. 419–429.

[12] P. Caserta and O. Zendra, "Visualization of the static aspects of software: a survey," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 17, no. 7, pp. 913–933, 2011.

[13] A. Noack and C. Lewerentz, "A space of layout styles for hierarchical graph models of software systems," in *Proceedings of the 2005 ACM symposium on Software visualization*. ACM, 2005, pp. 155–164.

[14] J. Holvitie and V. Leppänen, "DebtFlag: Technical Debt Management with a Development Environment Integrated Tool," in *Managing Technical Debt (MTD), 2013 Fourth International Workshop on*. IEEE, 2013.

[15] J. Holvitie, M.-J. Laakso, T. Rajala, E. Kaila, and V. Leppänen, "The role of dependency propagation in the accumulation of technical debt for software implementations," in *13th Symposium on Programming Languages and Software Tools*, k. Kiss, Ed. University of Szeged, 2013, p. 6175.