Co-Existence of the 'Technical Debt' and 'Software Legacy' Concepts

Johannes Holvitie*, Sherlock A. Licorish†, Antonio Martini‡, and Ville Leppänen*

* Turku Centre for Computer Science, Software Development Laboratory & University of Turku, Dept. of Information Technology, Turku, Finland {jjholv, ville.leppanen}@utu.fi

† University of Otago, Department of Information Science, Dunedin, Otago, New Zealand sherlock.licorish@otago.ac.nz

‡ Chalmers University of Technology, Computer Science and Engineering, Göteborg, Sweden, antonio.martini@chalmers.se

***Abstract*—'Technical debt' and 'software legacy' are concepts that both discuss a state of software that is sub-optimal, time constrained, and explain how this state can decrease an organization's development efficiency. However, there is significant confusion in the way the software engineering community perceive these concepts. In this paper we perform an initial examination of technical debt and software legacy concepts, and examine their somewhat challenging co-existence. In motivating our work, we discuss previous survey results which show that practitioners believe that technical debt largely emerges from software legacy. We then identify sources of confusion in comparing popular definitions of both concepts. Finally, we map the use of the 'technical debt' and 'software legacy' concepts in existing research. We conclude that structured co-existence of these terms can be pursued with mutually beneficial gains.**

I. Introduction and Background

All software products are static. That is, after they have been developed, prior to being extended or maintained at a later stage, they remain in the exact same state (formalized for technical debt by Schmid in [1]). The environment around software development, however, is dynamic. Technologies, people, organizational structures and processes all change over software development project duration. These variables, and many others, can be seen to have a link back to the static software product. For instance, consider the continued updates to a technology that is used to implement a software product: here the software product, for which development has stopped, does not abide by the latest changes made to the technology, and it becomes detached from the updates. Hence, when software development is continued for this software product, we note that it has accumulated technical debt in a "delayed fashion", as current assumptions (i.e. strategic and/or accidental technical debt) do not apply to it.

From a management perspective, there are considerable differences between immediate and delayed accumulation of technical debt. Immediate accumulation is affected mainly by matters that reside within the producing organization and its project. We may look into altering strategies and implementing new processes to affect the management of immediate, intended technical debt. Management of immediate, involuntary (or unintended) debt is often more indirect. For example, implementation and design quality issues often arise from practitioners having communication issues or being unaware of all applicable best practices. In these scenarios, exercises to enhance social togetherness and focused training, respectively, can be utilized to reduce technical debt instances.

Different strategies are required when dealing with legacy software. Legacy software has a variety of definitions, but it generally captures software artifacts which cannot be subjected to the same maintenance and management efforts as newly created artifacts [10]. In practice, these are often implementation artifacts which are old, undocumented and/or untested, and for which the original developer is no longer available. This could be due to: (1) a new team has taken over in the organization, or (2) the implementation has been acquired from somewhere else. If we consider technical debt accumulated in a "delayed fashion" to capture sub-optimalities that emerge due to the environment progressing around a static software product, we could argue that legacy software is quite close conceptually to technical debt.

In fact, we previously conducted a practitioner survey to shed light into the accumulation and composition of technical debt that software organizations face today. This survey was administered as a web-based questionnaire in Brazil, Finland, and New Zealand. We collected 184 responses from a diverse set of respondents using both agile and traditional development methods in which the practitioners assumed several roles ranging from developers to managers and client representatives. We have discussed outcomes from the Finnish responses in more detail before [11], while a forthcoming article reviews the multi-national results. As part of the results, the multi-national data-set captured 69 descriptions of concrete technical debt instances from the respondents.

For the captured technical debt instance descriptions, Figure 1 visualizes the distribution of the perceived origins of technical debt. While these results are somewhat preliminary, for over 75% of the instances, the indicated origins of technical debt are in software legacy. This indicates that practitioners perceive there to be a strong connection between 'technical debt' and 'legacy software'. This outcome and the general confusion around the technical debt and software legacy concepts motivates us to examine these two domains together, to identify cross-compatible solutions and enhancements for

both.

Examining these two domains together, technical debt could be seen to benefit from integration of legacy software management procedures, as this field has a very established status. However, several issues should be taken into account when legacy software is integrated with technical debt management. Firstly, is legacy software only a component of technical debt accumulated in "delayed fashion"? Arguably not, as the overall current state of the software product, to which legacy can be a part, has an effect on technical debt accumulation and management [12]. Hence, the domains seem to be distinctively different, which motivates us to explore these concepts.

Second, legacy software is generally used as a negative term for "derelict code", while technical debt implies pursuing asset management functions for sub-optimalities in varying software artifacts. Reviewing Figure 1, one identifies a potential danger in legacy being re-branded under the more favorable technical debt concept. Here, the asset management possibilities are left unexplored if legacy is not diligently converted into technical debt instances enabling full management. Noting the unobtrusive nature of legacy software, this is not an easy task to do, and having technical debt instances with varying levels of accuracy is bound to deteriorate technical debt management efforts overall.

Revisiting our position above, 'software legacy' and 'technical debt' are closely related and as such the software engineering community should pursue narrowing the gap between these fields. We make the first step towards bridging this gap in this work. In the following section (Section II) we provide a short evaluation of the two concepts, examining similarities in their definitions. We next conduct a short mapping study of the concepts' joint use in contemporary research literature in Section III. Finally, in Section IV we conclude this study by providing insights into avenues for how the co-existence of the two concepts could be organized, and outline future directions.

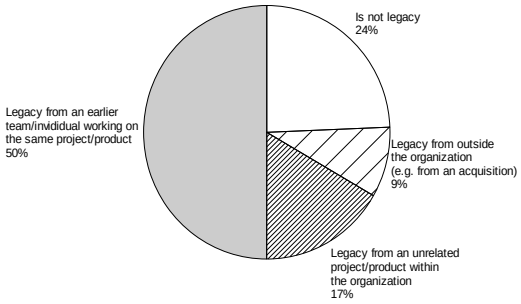

Fig. 1. Origins of technical debt instances (N=78 as multiple origins were indicated for some of the 69 instances)

II. Closeness of the Technical Debt and Legacy Concepts

To facilitate examining the closeness in definitions of the 'technical debt' and 'legacy' concepts, commonly accepted definitions for both are provided in Table I. Due to space limitations, detailed descriptions and examples have been excluded from the definitions.

Reviewing Table I, we note that in Cunningham's definition [2] technical debt (TD) is created always when new code releases are made. The definition leaves very little room for "debt-free code". If the consensus from the legacy definitions communicates that the "legacy" status is something the code can only gain over time, being pristine at the time of implementation, then Cunningham's definition clearly deviates from this. However, Cunningham describes the effects of TD as emergent from "not-quite-right code" which again is able

TABLE I. Definitions for the 'Technical Debt' and the 'Software Legacy' Concepts

	Cunningham [2]	McConnell [3]	Dagstuhl 16162 [4]
Technical Debt	*. . . Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise.*	*The first kind of technical debt . . . is incurred unintentionally. . . . [It] is the non-strategic result of doing a poor job. . . . this kind of debt can be incurred unknowingly, for example, your company might acquire a company that has accumulated significant technical debt that you don't identify until after the acquisition. . . .* *The second kind . . . is incurred intentionally. This commonly occurs when an organization makes a conscious decision to optimize for the present rather than for the future. . . .*	*In software-intensive systems, technical debt is a design or implementation construct that is expedient in the short term, but sets up a technical context that can make a future change more costly or impossible. Technical debt is a contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability.*
	Bennett [5]	Dayani-Fard [6] and Liu & al. [7] in [8]	Sommerville [9]
Software Legacy	*"large software systems that we don't know how to cope with but that are vital to our organization." . . .* *Moreover, many legacy systems are performing crucial work for their organization. Hence the decision on how to manage them is crucial: . . . the very future of the business may be at stake. Legacy systems may represent years of accumulated experience and knowledge.*	*Legacy software systems . . . were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms. The proliferation of such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve.* [6] *. . . many legacy systems remain supportive to core business functions and are indispensable to the business.* [7]	*Legacy systems are socio-technical computer-based systems that have been developed in the past, often using older or obsolete technology. These systems include not only hardware and software but also legacy processes and procedures—old ways of doing things that are difficult to change because they rely on legacy software.* *Legacy systems are often business-critical systems. they are maintained because it is too risky to replace them.*

to accommodate the effects of code that has transformed into “less-right code" overtime (i.e. legacy).

McConnell’s definition [3] is more lenient towards “debt-free code" existing. Technical debt accumulation in both the unintentional and the intentional scenarios is immediately related to the software elements creation, and hence, disallows degradation over time, however other (i.e. initially “debt-free”) scenarios are not excluded. Also, the unintentional incurrence includes acquisition-based accumulation of technical debt, wherein, the acquisition decision can be a “non-strategic result of doing a poor job”. This can include legacy acquisition.

Finally, the recent Dagstuhl 16162 definition [4] explicitly states that the software design or implementation construct has been expedient in the short-term. This implies that software elements, from the beginning of their development, must have a state that, interdependent with other factors, can induce sub-optimalities that contribute as technical debt. This, again, excludes legacy which is initially “debt-free”. The description promotes technical debt to be a “contingent liability" with an impact on “internal system qualities". This can accommodate effects commonly perceived for legacy software [8].

Moving to the lower half of Table I, we note that Bennett’s definition of ‘legacy’ [5] focuses on the operation phase of the software life-cycle [IEEE 12207-2008], and as such does not explain legacy’s emergence. However, the “how to cope with" effect description for legacy is very similar to Cunningham’s “stand-still". The similarity in effect descriptions continues in Dayani-Fard’s definition [6]. Here, legacy effects encompass problems that relate to proliferation of a system that has been continuously developed over decades. With very little effort, this can be classified under the “contingent liability affecting evolvability" from the Dagstuhl 16162 definition [4] for ‘technical debt’.

Finally, Sommerville’s definition [9] expands ‘legacy’ to include socio-technical matters. That is, *legacy systems* include processes and procedures which rely on the legacy software. We note that a very similar association has also been found for technical debt in the form of ‘social debt’ [13].

Reviewing the previous comparisons, we note that depending on the interpretation of the definitions, the concepts partially accommodate one another. The effects described for both the ‘technical debt’ and ‘software legacy’ concepts are similar. Arguably, this can be a source of confusion. However, when it comes to emergence, only McConnell’s definition for technical debt can accommodate legacy (in a very limited fashion). The two concept’s differ here.

Technical debt seems to be associated with a level of immediateness and clarity that implies the debt to be promptly manageable. That is, technical debt management requires information that indicates a particular software element to be sub-optimal. For legacy, there is information indicating a software solution (several software elements) to be sub-optimal in the current context. That is, this context and solution combination produces a sub-optimal outcome (this may again induce more technical debt). Hence, the connection between legacy and technical debt seems to be one of given contexts (aged vs. contemporary) and scopes (solution vs. elements).

TABLE II. Publications Matching Both the ‘Technical Debt’ and ‘Legacy’ Search Terms

Database	Matches
ACM	2
dblp	1
EBSCO	3
Elsevier (Science Direct)	33
IEEE (Xplore)	2
Springer (SpringerLink)	55
Thomson Reuters (WoS)	1
Wiley (WOL)	7
total	*104*

III. Review of Research on Technical Debt and Legacy Concepts

To understand the scale of joint use of the ‘technical debt’ and ‘software legacy’ concepts in research (noted among industry practitioners), we performed a preliminary mapping study over popular publication databases. We used the search phrase `"technical debt" AND "legacy"`. The software context was not explicitly interrogated, as both terms can be seen to already limit the query outcomes. Table II shows the number of matches received for the range of popular databases.

For the matched publications, we reviewed their contents and constructed an initial classification scheme based on how they handled the ‘technical debt’ and ‘legacy’ concepts together. This classification is provided in Figure 2. Over half (N=55) of the matched publications discussed the ‘technical debt’ and ‘legacy’ concepts in separate sections of the publications, and thus, were deemed *Non-related*. We could not access a seventh (N=16) of the publications, which we plan to further explore in a further systematic review. Two of the matched publications were also duplicates.

From the 104 matched publications, 31 discussed ‘technical debt’ and ‘legacy’ concepts together. Our initial classification for these matches shows that over a third of this amount (N=12) discussed the concepts as comparable with each other. For example, in Kennedy et al. [14]: *“This could also be considered as paying off your technical debt. Try to get rid of any legacy code and consider whether there are other ways you can format your CSS in order to get the most out of a minimization algorithm or better rendering...".* A similar portion of the studies (N=9) discussed legacy as an explicit cause for technical debt emergence. For example, in Knodel et al. [15]: *“...in particular for legacy systems, which have a history of design decisions made in the past ...inadequate for today’s requirements, and causing technical debt".*

The remainder of the matches conceptualized technical debt and legacy along two threads. The first half (N=5) described legacy as a modifier of technical debt. Generally, this meant that existence of legacy software in the development organization emphasized the negative effect caused by technical debt. For example, in Shull et al. [16]: *“If an organization has a history of damaging cost overruns during maintenance on a large legacy system, then they would be most interested in controlling design debt by refactoring code that is brittle,*

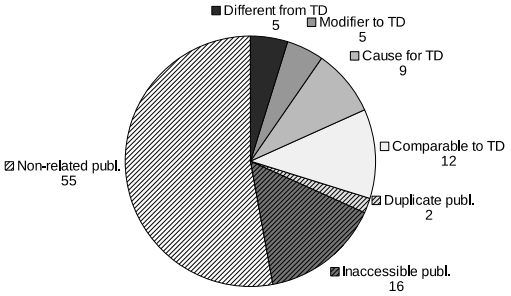

Fig. 2. Matched publications classified based on their relation to the 'legacy' concept ($N = 104$)

overly complex, or hard to maintain". Finally, the second half (N=5) of publications emphasized an explicit difference between 'technical debt' and 'legacy'. For example, in Williams [17]: *"Phases 1 and 2 address PCI DSS 3.1 [conversion from a legacy system], while phases 3 and 4 address the technical debt associated with neglecting this IT system"*.

Reviewing the matched studies above we note that there are a variety of contexts and ways with which technical debt and legacy are joined in research. While most of these contexts demonstrate an awareness of the concepts' co-existence (i.e. legacy may be both a modifier and a cause for technical debt), there are also combinations which seem to exclude either concept (i.e. if legacy is different from technical debt, then these concepts should not be compared).

IV. Structured Co-Existence of Technical Debt and Legacy, and Future Directions

Reviewing the survey results in Section I, the comparison highlighting the closeness in definitions for 'technical debt' and 'legacy' concepts in Section II, and the current state of compound use of the concepts in research mapped in Section III, we may conclude that there is considerable interplay between the technical debt and legacy domains. To this end, we believe that these concepts may be substituted, which could be possible source of confusion. The research mapping further highlighted this, where we found several different—partially exclusive—considerations of technical debt and legacy.

The current state of obfuscation between the concepts is a potential challenge. Notably, technical debt research is trying to come up with ways to identify, track, and manage concise technical debt instances, which would allow for technical debt to be managed in a systematic manner. We discussed in Section I that there is a danger of legacy being re-branded as technical debt, which may be seen to have less negative connotations. As legacy implies a severely reduced level of clarity around a particular set of software elements, re-branding such elements as technical debt without a systematic procedure can severely undermine technical debt management efforts.

Moving forward, we argue that the co-existence of the two concepts requires structuring in order to alleviate the possible confusion between their use, to allow for their concurrent use, and to facilitate mutually beneficial research for these fields. According to our short review, the similarity in the effects of technical debt and legacy seem to be the main challenge and strength for research conceptualization. This could be an excellent starting point for facilitating the structured co-existence of these concepts. The challenges presented by the given similarity in the conceptualization of these concepts may be reduced by providing clear definitions for each. We thus look to pursue this opportunity, first by extending the mapping work that is started here. Ideally, explicit definitions could lead to a process whereby legacy could be transferred into technical debt through a systematic approach.

In fact, the conceptual similarity could be exploited immediately. If the effects caused by legacy are comparable to those induced by technical debt, then this is a clear opportunity for technical debt management to adapt suitable procedures from the legacy domain. Software legacy has been an active research field for much longer than technical debt, and could thus yield mature solutions for this issue. This would allow technical debt researchers to direct more resources to the challenges that are unique to this field.

References

[1] K. Schmid, "A formal approach to technical debt decision making," in *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*. ACM, 2013, pp. 153–162.

[2] W. Cunningham, "The wycash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1992.

[3] S. McConnell, "Technical debt," 2007, 10x Software Development Blog. Construx Conversations. URL= http://www.construx.com/10x_Software_Development/Technical_Debt/.

[4] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)," *Dagstuhl Reports*, vol. 6, no. 4, pp. 110–138, 2016. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6693

[5] K. Bennett, "Legacy systems: Coping with success," *IEEE software*, vol. 12, no. 1, pp. 19–23, 1995.

[6] H. Dayani-Fard and e. al, *Legacy Software Systems: Issues, Progress and Challenges*. IBM: Technical Report TR-74.165-k, 1999.

[7] K. Liu and e. al, "Report on the first sebpc workshop on legacy systems." Durham University, 1998.

[8] R. S. Pressman, *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.

[9] I. Sommerville, *Software Engineering. International computer science series*. Addison Wesley, 2008.

[10] M. Feathers, *Working effectively with legacy code*. Prentice Hall Professional, 2004.

[11] J. Holvitie, V. Leppänen, and S. Hyrynsalmi, "Technical debt and the effect of agile software development practices on it-an industry practitioner survey," in *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*. IEEE, 2014, pp. 35–42.

[12] A. Nugroho, J. Visser, and T. Kuipers, "An empirical model of technical debt and interest," in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 1–8.

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

[14] A. Kennedy, I. De Leon, and D. Storey, *Pro CSS for High Traffic Websites*. Springer, 2011.

[15] J. Knodel and M. Naab, "What is the background of architecture?" in *Pragmatic Evaluation of Software Architectures*. Springer, 2016, pp. 11–20.

[16] F. Shull, D. Falessi, C. Seaman, M. Diep, and L. Layman, "Technical debt: Showing the way for better transfer of empirical results," in *Perspectives on the Future of Software Engineering*. Springer, 2013, pp. 179–190.

[17] B. R. Williams, *PCI DSS 3.1: The Standard That Killed SSL*. Syngress, 2015.