On the Limits of the Technical Debt Metaphor: Some Guidance on Going Beyond

Klaus Schmid

Software Systems Engineering, Institute of Computer Science,
University of Hildesheim, Germany
schmid@sse.uni-hildesheim.de

Please cite this publication as follows:
Klaus Schmid. "On the Limits of the Technical Debt Metaphor: Some Guidance on Going Beyond". In: *Fourth Workshop on Managing Technical Debt, Workshop at the International Conference on Software Engineering, 2013.* IEEE, 2013, p. 4.

The corresponding BibTeX-entry is:

```
@INPROCEEDINGS{Schmid13b,
  author = {Klaus Schmid},
  title = {On the Limits of the Technical Debt Metaphor:
           Some Guidance on Going Beyond},
  booktitle = {Fourth Workshop on Managing Technical Debt,
               Workshop at the International Conference on Software Engineering, 2013},
  year = {2013},
  pages = {4},
  publisher = {IEEE}
}
```

On the Limits of the Technical Debt Metaphor

Some Guidance on Going Beyond

Klaus Schmid
Software Systems Engineering, Institute of Computer Science
University of Hildesheim
Hildesheim, Germany
schmid@sse.uni-hildesheim.de

***Abstract*—Over recent years the topic of technical debt has gained significant attention in the software engineering community. The area of technical debt research is somewhat peculiar within software engineering as it is built on a metaphor. This has certainly benefited the field as it helps to achieve a lot of attention and eases communication about the topic, however, it seems it is to some extent also sidetracking research work, if the metaphor is used beyond its range of applicability. In this paper, we focus on the limits of the metaphor and the problems that arise when over-extending its applicability. We do also aim at providing some additional insights by proposing certain ways of handling these restrictions.**

***Index Terms*—Technical Debt, Software Architecture, Economic Model, Metaphor.**

I. Introduction

Technical debt has received significant attention in recent years as it addresses an important software engineering problem: a strong and never-relenting focus on delivering customer-relevant functionality may eventually lead to a situation where it is impossible to achieve exactly this as more and more technical problems in the implementation pile up. Cunningham initially addressed this issue by writing in [1]:

"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."

In [2], Ward Cunningham explained this metaphor further by describing that it was for him a way of motivating refactoring. Since the inception of the concept of *technical debt*, a lot has happened, workshops are given such as the "Managing Technical Debt"-series of workshops, complete tutorials [4], many publications, and special issues [5] have contributed to the current status of knowledge. Nevertheless, the metaphor has still a lot of influence, and in particular, as we see it, often a rather verbatim application of the concept of technical debt is attempted.

II. Shortcomings of the Technical Debt Metaphor

As we pointed out above, technical debt is a metaphor and thus constitutes an analogy as described by the definition of metaphor in the Merriam-Webster Dictionary [6]:

"a figure of speech in which a word or phrase literally denoting one kind of object or idea is used in place of another to suggest a likeness or analogy between them"

An analogy, however, means to compare two things and put them as alike even though they are actually not alike: in some respects the two things are different, and if we apply a reasoning that holds for (financial) debt to technical debt, we will reach incorrect or inappropriate conclusions. The point where the analogous conclusions no longer hold is often also called: the *analogy breaks down*. We will now ask in particular: where does the analogy break down?

In order to identify the shortcomings of the technical debt metaphor, we first look more precisely at the underlying concept of financial debt. Financial debt is characterized by several important parameters. First of all, there is the *owed capital* itself, i.e., the amount of capital that makes up the debt. In financial debt this can be always clearly and unambiguously quantified in a standardized unit system (a currency). Exactly this owed capital must be paid down (in addition to interest payments) to get rid of the debt. Thus, there is a well-defined inverse process: paying down the debt has the same quantity as the taken out debt.

On top of paying down the debt, there is also the need for an interest payment. This only depends on the time (interest period) and the interest rate. Thus, how much interest is necessary is directly proportional to time and independent of what one is doing (except for down-payments).

If we now look at technical debt as analogous to (financial) debt, we can identify shortcomings basically along all of the dimensions. Actually, a very important and fundamental shortcoming is having a standard unit of measurement. This does not apply to technical debt and gives in turn rise to many discussions. We deem it so important that we will actually devote the complete next section to this topic. Here, we will first discuss the remaining dimensions in turn.

Owed Capital: the owed capital is often compared (actually implicitly also by Cunningham in his original quote) to suboptimal development, e.g., a refactoring that is not performed, a non-optimal architecture that is accepted, and so

forth. As a consequence, a lot of research work sets out to identify such a sub-optimal structure in existing code, for example, by using reengineering techniques or code structure metrics.

If this is the main – or sometimes only – focus in determining technical debt, we see many problems with such an approach: first of all, what we find there, e.g., a specific software structure, does not directly relate to a debt, as it is extremely difficult to say what would be the right structure to have. Moreover, this "bad structure" by itself does not give rise to the interest payment, as we will discuss now.

Interest Payment: the interest payment is in the financial world due to an interest rate and the passing of an interest period and in consequence a purely time-dependent function. According to Cunningham's use of the metaphor the interest payment is the additional development effort that we spend on not quite right code; meaning the additional effort, beyond the effort that would be needed, if the code would be right. However, this has an implicit assumption that deserves to be made explicit and this assumption is that interest is only paid in the presence of further development effort. And this leads us directly to what we perceive as the major shortcoming of the technical debt metaphor:

Whether and how much interest needs to be paid, depends directly on the future development that is expected to be made on top of the current development result and not (only) on the structure of the system.

As a result, it is also not possible to relate technical debt directly to an interest rate or an interest period. Rather technical debt is intimately related to the future development we are going to make. We can put this differently: if we do not evolve or maintain the software further that is problematic (or touch the software in a way so that the specific choices made become relevant), we will not pay any interest: it is as if there would be no debt. On the other hand, if a lot of future evolution happens in this area, we will pay a lot of interest. Thus, a structural problem is not per se bad for future development; resolving it might even be a complete waste of effort. It rather depends on the specifics of future development. Freeman made a similar point in [10] where he compared technical debt to an unhedged call option.

As we argue in [9], the interaction of future development and current system realization also implies that technical debt should be measured as the difference in terms of development effort relative to an optimal realization of the system for future evolution. This in turn leads to significant problems in determining the "optimal" system, which casts some doubt on any approaches that aim only to find technical debt in terms of analyzing existing systems or code. We consider this problematic, first, because this perspective lacks the consideration of future development and second because for identifying problems in the code, we would also need to know what is the *optimal* system structure, something, which we can only define relative to future development.

As a consequence, one should look at existing system issues as *potential technical debt*, akin to a risk, while only the degree to which it is relevant to future evolution should be regarded as the relevant amount of technical debt (*effective technical debt*). This would still allow us to identify at least *potential* technical debt by looking at structures and architectures of existing systems. One could compare this to the dichotomy between a risk and a problem: a risk is something, which can potentially become a problem. In this comparison potential debt is a risk, while effective technical debt is a problem.

However, without taking into account the future evolution, it is still very difficult to assess what should be considered a structural problem of the system. The reason is that we cannot always even say whether something actually is bad code.

Let's look at the following situation: *we are working on an information system where we allow different access rights to data to different users, based on different categories of data. In order to support even more detailed rights management the developers decided to tag each individual data structure with access rights. Thus, even different data structures of the same type can have different access rights. Of course during future evolution this requires a higher amount of care – and thus effort – so that during all operations correct rights are used and that during transformations also the access rights are appropriately transformed.* The question now is: is this technical debt in the form of over-engineering or is it even a particularly smart approach? We claim, this cannot be adequately decided without looking at the future: if in the future there will be no use of this additional capability, then this was over-engineering and incurred technical debt along with interest payments. If in the (near) future this added capability will be exploited, it should not be regarded as technical debt as it actually improves development.

While this particular example may seem somewhat artificial, we use it only to a make a particular point: technical debt cannot be glanced alone from a specific structural assessment, but only in relation to future evolution as there is no absolute baseline what is good code. We believe this is a fundamental issue, although there might be special cases, which might nearly always be problematic: for example, if our code is full of code clones, this will probably always be negative, if any of the related code is ever touched again.

Thus, also the question whether a particular refactoring reduces or increases technical debt can sometimes only be analyzed by looking at future development. As a consequence, even for potential technical debt, we need to take future development plans into account. Thus, we can say that with respect to some future development scenarios we see a particular structural property as a potential technical debt, but without such a reference the structural property will often be difficult to be tagged technical debt. This also implies that internal qualities like good structure should not be defined without relating to future evolution. This is actually the approach taken by assessment approaches like ATAM [11].

As a consequence also the notion of *debt-free* becomes difficult. In financial debt the notion of debt-free is straightforward, but in technical debt, it becomes effectively undecidable, as it becomes nearly impossible to identify an optimal structure that is possibly free from unnecessary

development costs, at least without taking a specific scenario of future development as a given.

We can summarize these observations by the following lessons:

- Technical debt should be evaluated with respect to future evolution. Except for some extreme situations there is no such thing as an objective account of technical debt, independent of evolution.
- We need to differentiate between the structural issues[1] (the potential technical debt) and the effect it has on actual development (the effective technical debt). This is basically a cause-effect relationship and it can be seen as similar to a risk (a potential problem) and a problem in risk management.
- There is nothing like a technical-debt-free system. For each system, no matter how well design decisions were made, there can be future evolution steps that reveal that these design decisions led to problems.

So, to summarize our position on structural analysis as a basis for technical debt evaluation: *yes, it is useful as far as it can provide initial suspicion of potential technical debt. Then we need to assure that there indeed exist evolution scenarios for which this is potential technical debt (i.e., a sub-optimal implementation) and then in a specific development context we still need to determine, whether it will be the basis for effective technical debt.*

Addressing all problems that are only identified on purely structural grounds – without taking the evolution into account – will usually lead to significant investment that can never be recovered. Thus, this leads to gold-platting the system, instead of increasing its value.

After addressing these aspects and pointing out some of the shortcomings of the technical debt metaphor, we will now discuss the issue of valuation, which we deferred earlier.

III. Valuing Technical Debt

An important point in the context of financial debt is that we have a standardized currency – a monetary unit like the US$. This allows us to objectively measure debt and to also determine how to pay down debt (we need exactly the same amount). Unfortunately, none of this can be said of technical debt. There is no clear way of measuring it. This is expressed by different ways of handling the impact of technical debt in literature such as development costs or customer value [7]. Different ways of determining the impact of technical debt are:

- *Costs / Effort incurred* by technical debt. This seems to be closest to what was initially described by Cunningham.
- *Time-to-market*: time-to-market may increase, leading to negative effects to customer or seller (see below).
- *Loss of revenue*: if software hits the market at a later point, this may actually lead to missing a market window and thus revenue.
- *Lost value to customer*: development may be delayed and thus customers do get the value later – or not at all.
- *Loss of option value:* due to restrictions that are incurred by the technical debt some development options are destroyed, leading to a loss of option value.

These are just some examples and certainly not complete.

The list above shows some of the problems that are created by technical debt and we can use any of these metrics or any combination of them as a basis for determining technical debt. And some of these are indeed used by approaches. So what is the problem? The problem is simply that some of the effects are only rather indirectly related to the initial technical problem. We discussed this problem at quite some length in [8], so here we will only briefly summarize it. Probably the most directly related characteristic is the notion of cost / effort. This has also the big advantage that it is the same unit that created the initial problem, namely effort.

Already time-to-market is indirectly related. While it depends on the additional development effort required, in addition, we need to take the availability of personnel into account. Thus, the same difference in effort may lead to different delays [8]. The next two points lead us even further. They in turn depend typically on the time-to-market delay incurred, along with all the problems we discussed for it. In addition a customer or market valuation of the product is made and its change over time according to this valuation plays a role. This, of course, depends on parameters completely independent of the project.

Finally, the loss of option value goes even further, as here for any valuation additional assumptions on intermediary steps are required that exploit these options. Moreover, due to the leverage always associated with options, any imprecision in the input assumptions is leveraged as well. Thus, an error in input values of 10% (e.g., with respect to some development cost, which actually would be small), can easily be leveraged to 100% deviation. Further approaches may have additional, more complicated issues that need to be taken into account.

What this shows is: the more detailed the effect of technical debt is taken into account, the higher the value gets. In particular, if we add up several of these contributions, we can easily make the mistake to count basically the same underlying cost multiple times. Also the most direct correspondence is the cost saving and it is thus the fairest account for technical debt. Other ways of accounting for technical debt always bring external factors into the picture, which are typically hard to estimate and it is problematic to directly relate them to technical decisions in a project. The following points can be observed based on these arguments:

- again the metaphor breaks down: while in financial debt the valuation approach is clear and leads to a direct correspondence between how much debt is incurred and what must be paid back, this is not the case for technical debt.
- the more complex valuation mechanisms lead to problems like: they make hard to estimate assumptions on the development context and they introduce leverage, which will magnify any imprecision in the estimates.
- if we combine (in particular add) multiple individual valuation mechanisms, we may actually count the impact

[1] Some authors also refer to intrinsic or internal quality of the software.

of technical debt multiple times, leading to an exaggerated value of technical debt.

Especially the last point is very important when looking at a decision-making approach for technical debt. It implies that if we only measure technical debt in enough detail and add up the different value contributions any (temporary) deviation from an optimal path might seem inappropriate. However, this impression may only stem from adding different derived values that all come from the same source. As a consequence, while we do not say that necessarily evaluation should be restricted to looking only at cost/effort, much care must be taken when using a combined valuation procedure, as there is a risk that the valuation procedure itself may lead to wrong results.

A further valuation problem in technical debt relates to additivity. If we look at financial debt and take on two different debts, we end up with the sum of those. Not so in in technical debt. If we add two different debts, it can be smaller:

$$TD_{Sum} \leq TD_A + TD_B$$

We discuss this in more detail in [9]. Intuitively the main point is that taking on the second technical debt leads to changes of the system that might actually compensate for some of the aspects of the first debt, while on the other hand, it can never be more than the sum of both.

IV. Conclusion

In this paper, we discussed, what we regard as some of the more important problems with the technical debt metaphor. This should not be misconstrued as a fundamental negation of the usefulness of this metaphor, quite to the contrary. The technical debt metaphor has a significant value as it helps in communication and it provides some initial guidance, but just as with any analogy, it should not be used beyond the point where the analogy breaks down. Thus, we aimed to illustrate some of the points where the analogy breaks down and to some degree how to handle it, respectively what aspects to take into account when handling this. The interpretation of the technical debt problem given here provided also the basis for deriving an approach to managing technical debt, which is described in [9]. This approach relies on determining the effective technical debt and uses a decision procedure that is completely based on cost estimation, thus avoiding the valuation problem, we discussed in the previous section.

In summary, one can derive from the above discussions the importance of focusing on future evolution (respectively taking future evolution into account); something, which we find often neglected in current work on technical debt. Further, it is necessary to differentiate between potential and effective technical debt and in particular, it is important to accept that debt-free systems do not exist (at least not for a reasonable definition of debt-free). This should be taken into account when developing debt-analysis or debt-management approaches.

Finally, very much care must be taken for debt-management approaches, when valuing debt. The reason is that the valuation method has a strong impact on the outcome, and may even with best intentions misrepresent the underlying technical debt.

Acknowledgment

I would like to thank Ipek Ozkaya, Robert Nord, Felix Bachmann and David Garlan for patient discussions on technical debt. Further, I would like to thank Linda Northrop and the SEI/RTSS who made my stay and this work possible. I would further like to thank the Brazilian National Council for Scientific and Technological Development (CNPq), which supported this work partially.

References

[1] W. Cunningham, “Experience Report – The WyCash Portfolio Management System”, Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum), OOPSLA’92, pp.-39-30, 1992. Also online available at: http://c2.com/doc/oopsla92.html.

[2] W. Cunningham, “Ward Explains Debt Metaphor”, Online available at: c2.com/cgi/wiki?WardExplainsDebtMetaphor, January 2009, last checked: 8.2.2013.

[3] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka. “Managing technical debt in software-reliant systems”. In Proceedings of the FSE/SDP workshop on Future of software engineering research, FoSER ’10, pages 47–52, New York, NY, USA, 2010. ACM.

[4] P. Kruchten. Tutorial: Technical debt — from metaphor to theory and practice. Online available at: http://pkruchten.files.wordpress.com/2012/08/kruchten-120821-techdebt.pdf, Last visited: 20.01.13, 2013.

[5] P. Kruchten, R. Nord, and I. Ozkaya. Technical debt: From metaphor to theory and practice. IEEE Software, 29(6):18–21, November/December 2012.

[6] “metaphor”, Mirriam-Webster Dictionary, online available at: http://www.merriam-webster.com/dictionary/metaphor, last checked: 14.2.13

[7] R. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas. “Search of a metric for managing architectural technical debt”. In 10th Working IEEE/IFIP Conference on Software Architecture (WICSA 2012), pp. 91-100, 2012.

[8] K. Schmid. “An initial model of product line economics”. In Proceedings of the 4th International Workshop on Product Family Engineering (PFE4), pages 38–50, 2001.

[9] K. Schmid. “Technical Debt – From Metaphor to Engineering Guidance: A Novel Approach based on Cost Estimation”. Technical Report, University of Hildesheim / Software Systems Engineering, No. 1/2013, SSE 1/13/E, 2013. http://media/fb4/informatik/AG_SSE/PDFs/publikations/Schmid13a.pdf.

[10] S. Freeman. “Bad code isn’t Technical Debt, it’s an unhedged Call Option”. Blogpost at http://www.higherorderlogic.com/2010/07/bad-code-isnt-technical-debt-its-an-unhedged-call-option/, last checked: 6.3.2013.

[11] L. Bass, P. Clements, R. Kazman. Software Architecture in Practice. 3rd Edition. Addison-Wesley. 2012.