Technical debt decision making: Choosing the right moment for resolving technical debt

Ronald Kruizinga and Ruben Scheelder, Master students, University of Groningen

Abstract— Technical debt, software development compromises in maintainability to increase short term productivity, is an often discussed topic with respect to decision making, due to its prevalence and accompanying costs. Typically taking up 50-70% of a project's time [17], maintanance should be candidate one when it comes to lowering project cost. Many approaches for handling technical debt and deciding when to tackle it are available. These can be used in agile context but are not restricted to it. In this paper we perform a literature review of four approaches to decision making with respect to technical debt.
These approaches consist of the Simple Cost-Benefit analyisis which has its roots in the financial sector, another describes the Analytic Hierarchy Process often found in decision-making literature, the next provides an evaluation for the key factors that play a role in whether the debt should be paid immediately or be deferred and finally a highly formalized decision evolution approach. Each of the considered approaches have their own advantages and disadvantages. Over the course of this paper we explain how these methods work and we discuss the strengths and weaknesses of each method, where we compare them on seven factors, amongst which the feasibility of adoption in a company and the accuracy and completeness of the approach.

Index Terms—Technical debt, Decision making, Cost analysis, Software systems, Formalized evolution

1 Introduction

Technical debt (TD) was first mentioned by Ward Cunningham 1992 [2]. It can be defined as *software development compromises in maintainability to increase short term productivity*. Cunningham compares TD to debt and interest as used in the financial sector. Just like financial debt, technical debt comes with an increasing interest cost. Companies typically have change management teams that determine what changes are to be included in the next iteration/evolution of a system. An iteration then consists of a set of changes to the system also known as evolution items. These can be new features, bugfixes, workarounds, refactorings and more.

Technical debt can have different causes. It can be caused by complex requirements, lack of skill of the developer, lack of documentation and simply by choice, all of which impacts the way in which the TD should be handled.

The problem of technical debt becomes considering the size of TD in existing software [3]. Curtis et al. found that on average \$3.61 of technical debt exists for every line of code in over 700 large scale application [3]. Nugroho et al. estimated during a case study a TD interest of 11 percent in a large scale application, which would grow to 27 percent in 10 years [10]. The impact of interest becomes apparent given that that maintenance activities consume $50-70\%$ of a typical project development [17].

Agile working methods have gained more and more traction over the last few years. [5] Although a variety of methods (SCRUM, XP, FDD) is available and refactoring is already an integrated part of these methods [6], none offer a good solution to managing technical debt. Managing technical debt requires the answer to one question, which is the research question of this paper: How do you determine the right moment to fix technical debt?

Managing technical debt principally comes to down to a trade off between a robust future proof product that takes longer to develop and a quickly finished product which is hard to maintain. In this paper we explore different ways of making this trade off. We first discuss related work before listing several approaches to TD decision making. We start with a simple cost benefit tradeoff and an approach sourced in decision making literature and then go over more complex mathmatical appraches, taking into account different a variety of factors in their tradeoff, including not only technical factors (like code metrics) but also commercial ones (such as customer satisfaction). After that we discuss and compare the approaches while placing them in a wider context to find whether there exists a right moment to fix technical debt and when to fix it.

- *Ronald Kruizinga is a master student Computing Science at Rijksuniversiteit Groningen. E-mail: r.m.kruizinga@student.rug.nl.*
- *Ruben Scheedler is a master student Computing Science at Rijksuniversiteit Groningen. E-mail: r.j.scheedler@student.rug.nl.*

2 Related Work

Martini et al. performed a case study of Architectural Technical Debt (ATD) [8]. They interviewed several types of actors in the software development process (Architects, developers, scrum masters) and found that ATD comes with objectionable consequences like vicious circles, in which fixing the ATD in a quick manner brings even more ATD.

They were not the first to establish the danger of TD. A growing amount of studies is being performed on the subject of TD. Managing TD mainly consists of three procedures.

- grouping
- quantifying
- decision making

Grouping is the process of translating a system into actionable units. Most related studies focus on TDM in an agile environment [12] [1] [7]. Therefore, we see a general trend in managing technical debt using a backlog similar to the one used in agile development for feature issues but instead containing technical small independent debt items [12]. This item log takes care of the grouping process.

Quantifying is the process of attaching values to TD items based on certain metrics. Li et al. [7] performed a mapping study towards the current understanding of TD and TDM. It reviews over 90 papers on the topic of TD(M) and gives an overview of approaches used in managing technical debt. For this they use TD dimensions introduced by Tom et al. [16]. This research distinguishes different dimensions of TD among which code, architecture and environment. Li et al. extend the dimensionality into subtypes of TD [7]: requirements, architecture, code, documentation, versioning and more. Their overview of tools used to manage TDM shows a major focus solely on code TD. Integrated Development Environments, for example, offer refactoring for poorly written code.

Interestingly, Nord et al. conducted a research in finding a metric to express technical debt [9] and although available tools focus on primarily on code TD, this research found that code level metrics are insufficient to express TD. It proposes to focus more on architecture-related debt to help optimizing releases which requires to look to at non-functional properties like complexity and performance.

Although tools exist for managing code TD, other types are harder to manage. Especially when it comes to decision making: given a list of feature items and TD items, which should be selected for the following release?

Seaman et al. propose to measure technical debt in terms of interest and principal with interest being the extra time required to achieve changes in the system by incurring TD and principal being the cost (in time) to fix the TD [13]. We will elaborate on this method in Sect. 3.1.

Ho et al. propose a similar process [4], which uses documentation, testing, maintainability and complexity as metrics to quantify TD and embed the monitoring of these dimensions in a process that weighs features against paying of TD.

Nord et al. propose a somewhat different mathmatical model for the task [9]. A model is proposed in which again implementation cost is taken into account, but now combined with the amount of dependencies a (new) component has. The dependencies are the key to determine whether to refactor or whether to postpone it.

Martini et al. introduces an impressive collection of properties to be used to prioritize items [8]. Most interestingly it adds a commercial perspective to the decision making process by introducing aspects like customer long-term satisfaction and competetive advantage.

In the following section we will elaborate on the decision making processes introduced by Seaman et al. [13] and Martini et al. [8] to provide an overview of the state-of-the-art of decision making on TD.

3 METHODS

This study compares 4 different approaches for TDM. In order to properly answer our research question, we will compare these approaches on multiple criteria, taking into consideration that not all of these approaches have been formally tested or used in a realistic environment. The approaches considered in this comparison are:

- Simple Cost-Benefit Analysis
- Analytic Hierarchy Process (AHP)
- Defining Decision Factors
- Formalized Evolution

In this section we will describe these different approaches to TDM.

3.1 Simple Cost-Benefit Analysis

In Seaman et al. [14], the Simple Cost-Benefit Analysis is introduced, which is based on traditional financial cost-benefit analysis. It is the simplest approach to managing TD and therefore provides us with a good baseline for comparison with the other approaches.

This approaches focusses on a ”technical debt list”, which contains items representing a task that is not completed (satisfactorily) and thus might cause problems in the future. Examples of this include missing tests, needed refactorings or missing documentation.

Each of these items has a **principal** and an **interest**. The principal describes how much work is needed to complete the item. The interest consists of two parts, the **probability**, which describes the chance that the debt will lead to problems if not paid, and the **amount**, which describes how much extra work the debt will cause if not paid. The probability itself depends on the timeframe for which the analysis is done. A module (software element) that contains TD might not change in the upcoming month, but could be highly likely to change before the end of a year.

These three metrics are then assigned estimated values of high, medium or low. These are rough estimates, but sufficient in making preliminary decisions that can be worked out later in more details. Example usage of a slightly more fine-grained approach can be seen in Fig. 1, which contains 9 god classes that should be refactored. A possible strategy for paying the debt would be to start in the upper left corner, which are high interest classes that require little effort(low principal) and then moving down and right.

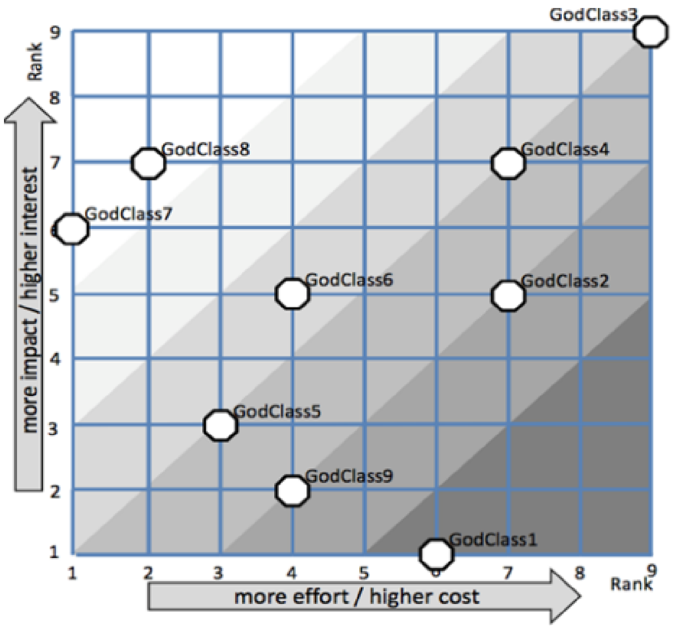


Fig. 1. Cost-Benefit matrix for nine god classes in an industrial software application. [14]

This approach increases in reliability when more data is available, especially on the impact of the TD on the rest of the application. However, even with limited amounts of data, expert opinion can be used to provide for a decent approximization of the interest of an item. In addition, this approach integrates well with software metrics, making it easy to adopt into a workflow.

3.2 Analytic Hierarchy Process

Seaman et al. [14] introduce a second process to work with TD, the Analytic Hierarchy Process (AHP). This process comes from the decision sciences and is a well supported approach in literature [11].

It structures a problem solving process by comparing alternatives with regards to certain criteria, resulting in a ranking of these alternatives. It structures this process by making a hierarchy of criteria and then performing pair-wise comparisons of alternatives under these criteria, resulting in a priority vector for the alternatives.

In Fig. 2 an example is available, where the alternatives (choices) X, Y and Z are compared with respect to the criteria (factors) A, B, C and D, in order to reach a certain goal. Factors can be quantative and qualitative which makes factors like ’product performance’ possible with qualitative values like ’much better’ and ’worse’.

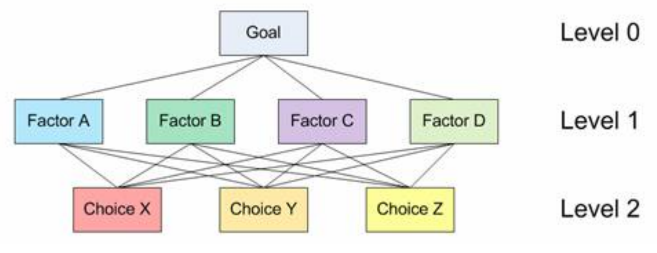


Fig. 2. Example of AHP.

Many of the pair-wise comparisons are objective and quantitative, and can thus be automated, resulting in an approach that reduces the amount of human input required. In applying AHP to the TD problem, the alternatives would be the instances of TD in the system and the output of the process would be a ranking of these items, prioritising items that should be paid off first.

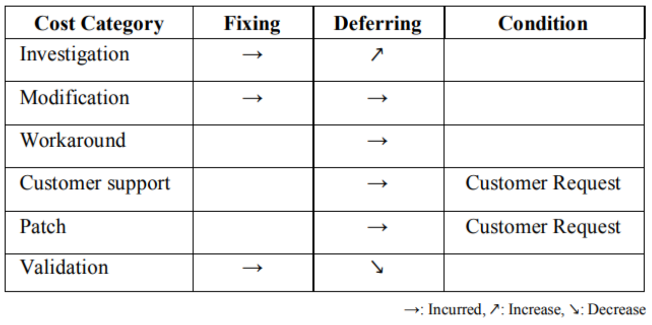
3.3 Defining Decision Factors

Snipes et al. [15] approach the TD problem from a different direction, as they do not consider the effects of the debt on related modules, but only on the module in which the debt occurs. The authors performed a survey under 7 Change Control Board (CCB) members for a product of over one million lines of source code.

This has lead to the classification of six categories of TD costs.

- Investigation: The cost of diagnosing and verifying a problem.
- Modification: The cost of implementing and verifying a fix.
- Workaround: The cost of providing workarounds for a known failure.
- Customer support: The cost of providing support to customers that encounter the problem.
- Patch: The cost of providing a temporary fix to affected customers.
- Validation: The cost of testing the system in its entirity.

Not all of these costs have the same importance, as Investigation cost is estimated to be 50-70% of the cost of fixing a defect [15] and Validation cost is between 20-30%. These costs also change when a fix is deferred, as deferring a fix might lead to more customer support or patching, in addition to a workaround being required. However, Validation costs are lowered, as validating the workaround is easier than validating a fix. The effect of deferring on this costs can be found in Fig. 3.



Cost Category	Fixing	Deferring	Condition
Investigation	→	↗	
Modification	→	→	
Workaround		→	
Customer support		→	Customer Request
Patch		→	Customer Request
Validation	→	↘	

→: Incurred, ↗: Increase, ↘: Decrease

Fig. 3. Change patterns of defect costs [15].

According to the respondents, there are multiple key factors that play a role in deciding when to fix TD. In order of importance, they are:

- Severity: The importance of capabilities affected by the defect.
- Workaround existance: Whether it is possible to implement a workaround and defer the fix.
- Urgency by customer: Has the fix been specifically requested by the customer.
- Effort to implement: Estimated effort versus resources available and the schedule.
- Risk of fix: Estimation of the extend of code and functionality will be effected and will need to be tested.
- Scope of testing: The impact of the change and the amount of testing required.

However, while these factors play an important role in TD decision making, Snipes et al. [15] also propose a more formal way of deciding whether the TD should be paid or deferred to a later date. In order to do this, they developed a cost-benefit analysis based on the key factors identified. In the equation seen in Fig. 4, P is the cost of Investigation, Modification and Validation, representing the costs of paying of the debt right now. It therefore is the principal of the debt.

The interest is formed of multiple components. I_w represents the cost of defining a workaround and I_c the cost of Customer support. $I_{pr} * I_p$ represents the probability that a customer will request a patch, multiplied by the cost of the patch. I_{fr} then is the probality that the deferred defect will eventually be fixed, making $P * I_{fr}$ the expected cost to pay off the debt.

$$\rho = \frac{P}{I_w + I_c + I_p * I_{pr} + P * I_{fr}}$$

Fig. 4. Equation for Cost Benefit Analysis Ratio [15].

Whenever the cost-benefit ratio $\rho < 1$ it is better to pay the principal immediately and fix the defect right now, otherwise, there is a financial advantage to deferring the fix. However, this equation is too simplistic, as it does not consider the effects of time on factors such as I_c or I_{pr}, nor the opportunity cost of the other tasks that could be done instead.

3.4 Formalized Evolution

Schmid [12] provides us with a highly formalized approach to reasoning about technical debt and its resolution. He defines technical debt in Definition 1.

Definition 1. $TD(S,e) = max\{CC(S,e) - CC(S',e) | S' \in Sys(S)\}$

In this formula, *TD(S,e)* is the technical debt for a system *S* together with an evolution step *e*, which is a single change of the system. *Sys(S)* describes all systems that are behaviorally equivalent to S. Finally, *CC(S,e)* describes the cost of performing evolution step *e* on system *S*. This means that the TD in Definition 1 describes the cost of performing the change on the current system *S* compared to the most optimal implementation of *S* with the same behavior.

However, evolution does not usually occur as a single change, but commonly as a sequence of evolution steps. Given an evolution sequence $\vec{e} \in E^*$, where E^* describes the set all possible evolution sequences for a system, then we conclude the new Definition 2.

Definition 2. $TD(S,\vec{e}) = max\{CC(S,\vec{e}) - CC(T,\vec{e}) | T \in Sys(S)\}$

This also describes the technical debt as the cost of a sequence of changes relative to an optimal implementation.

Now the question remains whether it is better to pay this TD immediately or at a later moment. In order to do this, we do not just consider the planned evolution sequence ($\vec{e}_{plan}$), but also the potential evolutions ($\vec{e}_{pot}$) following it. For simplicity, we assume that refactoring or restructoring can only be done at the beginning or the end of a sequence. If c_{rest} represents the cost of restructuring, we can visualize this as in Fig. 5.

$$\begin{tikzcd}
S \bullet \arrow[r, "\vec{e}_{plan}"] \arrow[d, dashed, "c_{rest}"'] & \bullet \arrow[r, dashed, "\vec{e}_{pot}"] \arrow[d, dashed] & {} \\
S' \bullet \arrow[r, "\vec{e}_{plan}"] & \bullet \arrow[r, dashed, "\vec{e}_{pot}"] & {}
\end{tikzcd}$$

Fig. 5. Development alternatives [12].

This means that it is beneficial to refactor immediately iff

$$CC(S,\vec{e}_{plan}+CC(S_{plan}\vec{e}_{pot})>=c_{rest}+CC(S',\vec{e}_{plan})+CC(S'_{plan}\vec{e}_{pot}) \tag{1}$$

In Equation 1, S_{plan} is the state of system S after $\vec{e}_{plan}$. If the cost of implementing $\vec{e}_{plan}$ and $\vec{e}_{pot}$ is greater than the cost of refactoring and then implementing $\vec{e}_{plan}$ and $\vec{e}_{pot}$ onto this restructured system, it is beneficial to do so.

However, this procedure has a fundamental flaw, as there is not a single path for future development, but potentially infinite different paths. In order to replace the precise values previously used, we thus introduce the notion of *expected cost* and *expected technical debt*.

For each $\vec{e} \in E^*$, where E^* describes the set all possible evolution sequences for a system, p describes a probability measure over E^*, i.e., $p(\vec{e}) \in [0...1]$, such that $\sum_{\vec{e}\in E^*} p(\vec{e}) = 1$. We use this to define the Expected Cost of a system S in Definition 3.

Definition 3. $\widehat{CC}_p(S) = \sum_{\vec{e}\in E^*} p(\vec{e}) * CC(S,\vec{e})$

The *expected technical debt* $\widehat{TD}_p$ is defined correspondingly (by using $\widehat{CC}_p$ in Definition 2). The calculation to determine whether it is beneficial to restructure changes accordingly to:

$$CC(S,\vec{e}_{plan})+\widehat{CC}_{pot}(S_{plan})>=c_{rest}+CC(S',\vec{e}_{plan})+\widehat{CC}_{pot}(S'_{plan}) \tag{2}$$

S' is the state after performing the restructuring corresponding with c_{rest}. While Equation 2 might seem trivial, it shows that the *estimated technical debt* is always impacted by the uncertainty of development, which means that we can never predict the optimal way of handling TD.

3.4.1 Approximating formal optimization

The analysis of formal evolution so far relies on a number of optimizations that can lead to an infinite search space for indentifying and handling TD. Three problems are encountered that can be solved using an approximation:

- The number of behaviorally equivalent systems can be very large, potentially infinite, which makes finding an optimal system, used in Definition 1 and Definition 2, highly time consuming. This problem will be addressed by introducing a fixed relative system.
- The number of potential evolution sequences grows exponetially with the length of those sequences. Under certain circumstances we can restrict our analysis to single evolution steps.
- After applying the previous two optimizations, the number of potential evolution steps might still be infinite or too large to consider properly. Thus we look at whether it can be replaced with a finite set and what the consequences are of doing so.

In order to select a fixed relative system, we need a system that approximates an optimal solution and is good enough to fill the role of optimal solution. Often, an expert solution is available that can function as an optimal solution S'. We use this to define *relative technical debt* RTD as per Definition 4.

Definition 4. $RTD(S,S',\vec{e}) = CC(S,\vec{e}) - CC(S',\vec{e})$

Expected relative technical debt $\widehat{RTD}$ can be defined correspondingly using the *expected cost* $\widehat{CC}$, which can again be used to calculate whether in it beneficial to restructure right now.

Using an expert solution instead of a theoretical optimal solution brings this method a step closer from theory to practice. However, the CC function still requires exhaustive probability calculation over an infinite number of evolution steps and sequences. To deal with this, we replace the potentially infinite space of sequences with a representative finite subset.

Definition 5. $c_{rest} \leq \sum_{e<\vec{e}_{plan}} RTD(S,S',e) + \widetilde{RTD}_n(\vec{e}_{plan}(S),\vec{e}_{plan}(S'))$

Definition 5 allows for approximating c_{rest} (the restructuring cost of an evolution plan). It consists out of two components. First, the sum of the relative technical debt (RTD, Definition 4) for each evolution step e. That is, the technical debt caused by only the specific step onto the system.

Second, the approximated relative debt of the sequence as a whole. The idea is that evolution steps implemented together may do even more harm than implementing them seperately (e.g. a harmful dependency between the two). This type of relative debt is relatively hard to define, but is it can be approximated using the following definitions:

Definition 6. $\tilde{p}(e) = \begin{cases} \sum_{\vec{e}\in E^*, e<\vec{e}_{plan}} p(\vec{e}) & e \in \tilde{E} \\ 0 & e \in E \setminus \tilde{E} \end{cases}$

Definition 7. $\widetilde{STD}_{\tilde{E}(S)} = \sum_{e\in\tilde{E}} \tilde{p}(e) * TD(S,e)$

Where earlier definitions for (R)TD are correct, they are not practically usable due to their infinite nature. In Definition 6 and Definition 7 E^* is the set of all potential evolution sequences (infinite). E is the set of all potential evolution steps (infinite) and $\tilde{E}$ is a finite representative subset of E, containing n sequences. This is critical since it allows to estimate the complex interdependent TD of evolution steps using a finite set of possible evolution sequences. $\widetilde{RTD}_n$ can be derived from Definition 7 and Definition 4.

Although Definition 5 is usable through the use of only a finite set of potential evolution paths, the formula is still computationally complex. Schmid [12] proposes that the debt coming from the interdependence of evolution is negligible. This allows for simplication of Definition 5 into the following:

Definition 8. $c_{rest} \leq \sum_{e\in E'} RTD(S,S',e) * p'(e)$

$E' = \tilde{E} \cup \{e | e \in e_{plan}\}$ and p'(e) is identical to p(e) except that it returns probability 1 for all steps e already part of the evolution plan e_{plan}.

This final definition allows for relatively simple decision making since it gives a clear definition of the technical debt that can be compared to the restructuring cost. It sums the RTD of all evolution steps part of the plan and the RTD of all other potential evolution steps, weighted by their likelihood of occurance. This concludes the formal method proposed by Schmid.

4 Discussion

We now go over the different methods that are described in Sect. 3 by comparing them on several properties.

- Accuracy: how accurate a method predicts the TD state of an item.
- Automation: in what capabilities can the method be automated.
- Completeness: the capability of a method to consider all relevant factors when quantifying a TD item.
- Feasability: how likely it is that companies will adopt a method and/or how easy it is to integrate in the workflow.
- Flexibility: how easy it is to change the method to suit the needs of the company.
- Transparancy: how clear it is to the users how the result has been reached.
- Workload: the effort required to use the method.

Table 1 summarizes our findings. Every method is given a score for every property that ranges from 1 (very low) to 5 (very high).

Table 1. Overview of methods and their scores. Methods are abbreviated as follows: Simple Cost Benefit Analysis (SCBA), Analytic Hierarchy Process (AHP), Defining Decision Factors (DDF) and Formalized Evolution (FE).

	SCBA	AHP	DDF	FE
Accuracy	1	3	3	5
Automation	2	4	2	3
Completeness	1	4	2	5
Feasability	3	4	3	2
Flexibility	2	5	3	3
Transparency	5	2	4	2
Workload	5	3	3	1
Total	19	25	20	21

4.1 Simple Cost Benefit Analysis (SCBA)

What makes Simple Cost-Benefit Analysis stand out is its simplicity. It has only three components (principal, interest amount and interest probability) which makes it attractive to add to workflow. Therefore, SCBA scores high on feasability, workload and transparency.

However, the simplicity is also its weakness. The equation used by SCBA does not consider factors beside principal cost and interest in the cost of developing new features. Take for example the factor 'performance'. If incurring technical debt decreases the performance of a system by 10 percent, this will only affect end users of the system but will not generate a growing interest of development cost. However, it might lead to losing customers and thus a different form of interest. Completeness and accuracy scores of this method are therefore low.

4.2 Analytic Hierarchy Process (AHP)

The strength of AHP lies in its flexibility and its pragmatism. AHP sets no constraints to factors that are used, making it flexible and also potentially complete. Furthermore, it breaks down the question of factor priorization into small comparisons: each factor is directly to compared to other factors and the algorithm generates a general priorization from that using eigenvectors. Besides setting up the matrix for comparing features, most of the operations can be automated.

The disadvantages of AHP are not obvious yet they are present. The flexibility brings great potential but it requires the user of the method to know how to configure it. Other methods come with their own balancing based on research which (although less flexible) might yield better results. The maths underlying the method also make it less transparent to some users.

4.3 Defining Decision Factors (DDF)

DDF does not stand out positively nor negatively. It achieves some accuracy and completeness by considering all steps part of fixing or postponing an issue. However, both suffer from the fact that the effect on the rest of the system is left out of the equation.

The input on DDF then also requires broad knowledge of the system. Factors like 'urgency by customer' and 'risk to fix' are best answered by different people in an organization. This makes the method more complex to adopt.

Finally, the method is very clear. This increases the transparency of it and also has a positive effect on feasability and (partial) automation.

4.4 Formalized Evolution (FE)

This approach provides some significant benefits, but also has its fair share of drawbacks. The formalization offers a precise analysis of technical debt, which also shows that we require an infinite searchspace and need to identify theoretically optimal systems, which is not feasible in practice. Approximation can alleviate these issues, but also causes deviation from the 'real' values. The major drawback of this method is the workload it brings. It requires to predict many evolution paths to calculate the restructuring cost. It is possible to use very few, but this directly impacts the accuracy of the method.

Furthermore this method is build on one abstract cost function (CC). This function is flexible in the sense that no hard definition of it is given besides: *"the (minimal) cost required to perform an evolution step on a system"*. If we assume that this only entails the cost to implement an evolution step, it is both transparent and relatively easy to use, since cost is something businesses are already familiar with. However, it also means that similar to SCBA and DDF some factors might be left out of the equation.

Automation of this method is doable as it is mostly a mathmatical model that directly allows comparison of alternatives. However, the implementation costs are still required as an input and might be hard to calculate depending on how many factors are included in it.

In general, this method provides strong theoretical foundations for further decision making approaches and research, while also being usable in realistic environment. Although the workload to use it is high and the algorithm is not easy to understand, it can yield very accurate results.

4.5 Necessity of Fixing Debt

An important note to make is that although most relevant research has underlined the need for refactoring, it is not imperative that investing time in decision making is beneficial.

Definition 9. $T_{saved} = T_{td} - T_{dm}$

In Definition 9 T_{td} represents the time saved by refactoring, T_{dm} the time taken for decision making and finally T_{saved} is the nett time saved for the company.

This formula makes clear that although a decision making method can be very good in finding the right moment to fix TD (thereby saving time in the future) it needs to be proportional to the TD it proposes to fix. Otherwise, the process as a whole is still not beneficial. It also shows that improving the decision making process ad infinitum is not always beneficial as it might increase the workload so much that the total T_{saved} becomes negative.

5 Conclusion

Technical debt is becoming a major issue in software development and a factor in its decision making. Therefore, methods are required to deal with technical debts that help with finding the right moment to resolve technical debt. Research has been conducted on the topic of technical debt already, providing a variety of methods to deal with it. These methods typically solve two problems: how do we quantify technical debt and how do we weight candidates?

We discussed four methods that attempt to answer these questions. Simple Cost-Benefit Analysis (SCBA) is a simple model that assigns principal and interest to technical items, just like in finance. Defining Decision Factors (DDF) categorizes the cost of technical debt into six defining factors which are used to choose between incurring and resolving TD. AHP provides a more general tactic that is grounded in decision-making literature. Users can define all factors they deem relevant. AHP orders them and is able to rank technical item importantance based on that feature importance and scores given to the technical items. Finally, Formalized Evolution (FE) builds a mathmatical and statistical model based on the cost of implementing an item and the debt it brings. It provides an accurate prediction of whether a refactoring will pay of, given the correct item costs and potential evolution paths.

All four methods have their benefits and disadvantages. SCBA is easy in use but not complete. DDF provides good insight in the cost of technical debt, but requires precise input and lacks non-technical factors. AHP is flexible and well automatable. However, it requires so much input that the approach itself does not provide much more than a framework for decision making. FE is very accurate but more complex in use and moreover very time consuming, although some parts can be automated. Given all aspects of the methods, we found AHP to be the best tactic.

Although these methods are very promising, more research is needed in the form of case studies that apply these methods in practice. Furthermore, the feasability of the decision making process itself should be subject to research especially when the complexity of it increases.

ACKNOWLEDGMENTS

The authors wish to thank A, B, and C.

REFERENCES

[1] W. N. Behutiye, P. Rodrguez, M. Oivo, and A. Tosun. Analyzing the concept of technical debt in the context of agile software development: A systematic literature review. *Information and Software Technology*, 82:139 – 158, 2017. doi: 10.1016/j.infsof.2016.10.004

[2] W. Cunningham. The wycash portfolio management system, 1992.

[3] B. Curtis, J. Sappidi, and A. Szynkarski. Estimating the size, cost, and types of technical debt. In *Proceedings of the Third International Workshop on Managing Technical Debt*, MTD '12, pp. 49–53. IEEE Press, Piscataway, NJ, USA, 2012.

[4] T. T. Ho and G. Ruhe. When-to-release decisions in consideration of technical debt. In *2014 Sixth International Workshop on Managing Technical Debt*, pp. 31–34, Sep. 2014. doi: 10.1109/MTD.2014.10

[5] J. Jeremiah. Survey: Is agile the new norm?, 2015.

[6] D. Leffingwell. *Scaling software agility: best practices for large enterprises*. Pearson Education, 2007.

[7] Z. Li, P. Avgeriou, and P. Liang. A systematic mapping study on technical debt and its management. *The Journal of Systems & Software*, 101:193–220, 2015.

[8] A. Martini and J. Bosch. Towards prioritizing architecture technical debt: Information needs of architects and product owners. In *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, pp. 422–429, Aug 2015. doi: 10.1109/SEAA.2015.78

[9] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas. In search of a metric for managing architectural technical debt. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pp. 91–100, Aug 2012. doi: 10.1109/WICSA-ECSA.212.17

[10] 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*, MTD '11, pp. 1–8. ACM, New York, NY, USA, 2011. doi: 10.1145/1985362.1985364

[11] T. L. Saaty. *Decision making for leaders: The analytical hierarchy process for decision in a complex world*. Lifetime Learning Publications, 1982.

[12] K. Schmid. A formal approach to technical debt decision making. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '13, pp. 153–162. ACM, New York, NY, USA, 2013. doi: 10.1145/2465478.2465492

[13] C. Seaman and Y. Guo. Chapter 2 - measuring and monitoring technical debt. vol. 82 of *Advances in Computers*, pp. 25 – 46. Elsevier, 2011. doi: 10.1016/B978-0-12-385512-1.00002-5

[14] C. Seaman, Y. Guo, N. Zazworka, F. Shull, C. Izurieta, Y. Cai, and A. Vetr. Using technical debt data in decision making: Potential decision approaches. In *2012 Third International Workshop on Managing Technical Debt (MTD)*, pp. 45–48. Zurich, June 2012. doi: 10.1109/MTD.2012.6225999

[15] W. Snipes, B. Robinson, Y. Guo, and C. Seaman. Defining the decision factors for managing defects: A technical debt perspective. In *2012 Third International Workshop on Managing Technical Debt (MTD)*, pp. 54–60. Zurich, June 2012. doi: 10.1109/MTD.2012.6226001

[16] E. Tom, A. Aurum, and R. Vidgen. An exploration of technical debt. *J. Syst. Softw.*, 86(6):1498–1516, Jun 2013. doi: 10.1016/j.jss.2012.12.052

[17] H. Van Vliet. *Software engineering: principles and practice*, vol. 13. John Wiley & Sons, 2008.