Towards a Technical Debt Management Framework based on Cost-Benefit Analysis

Muhammad Firdaus Harun, Horst Lichter
RWTH Aachen University, Research Group Software Construction
Aachen, Germany
e-mail: {firdaus.harun, horst.lichter}@swc.rwth-aachen.de

***Abstract*—Technical debt (TD) is a metaphor of bad software design or immature artifacts of a software system. The metaphor has been quite intensively researched especially on how to identify the TD symptoms, (e.g., system deficiencies or architecture violations) explicitly. Although the TD identification is quite important in the TD management process, a systematic management of TD and how to reduce it should also be considered important in each release of the development project. Otherwise, the software becomes more and more unmaintainable. In this paper, we introduce a framework to manage and reduce the TD of software systems. As it is based on quantification and a cost-benefit analysis, it is called Cost-Benefit based Technical Debt Management (CoBeTDM). CoBeTDM defines explicit phases focusing on the most important aspects of TD management: identification, monitoring, and prioritization. Overall, CoBeTDM should support managers to take the right decisions regarding the software evolution and the reduction of the collected TD at the right time.**

Keywords–technical debt management; code smells; architecture smells; refactoring; cost-benefit analysis.

I. Introduction and Motivation

It is a must to implement a payback strategy (when and how to determine to pay it back) to reduce technical debt for every software organization. It has been reported that TD exists in most of the software systems [1]. If we do not cautiously manage the debt or have no strategy to pay it back, the system may finally go to the “bankruptcy” phase, i.e., the software is unmaintainable and the maintenance cost will increase continuously. In general, refactoring is one of the strategies to pay it back. Refactoring has typically been used as a mean to improve detailed design and code quality. In this paper, refactoring will be referred to as an effort to improve existing software either on code or architecture-level without changing the behaviour of the system.

Commonly, project managers are always juggling on the decision making either to add new features or to make changes, (i.e., maintenance or refactoring) in a release cycle. It is always complicated to decide, which refactoring task should be done first or could be postponed. Therefore, quantification of refactorings should be implemented to identify, which effort can achieves maximum benefit and minimize risk. A simple cost-benefit analysis is a simple approach that could be applied to quantify it as introduced [2]. Borrowing from economic domain, a cost is a principal that indicate effort estimation to resolve a TD item and a benefit is an interest that indicate less probability impact to the software system. However, the quantification cannot answer the question “How the refactoring effort could be paid off to the identified TD, i.e., Return On Investment (ROI)?”. ROI is a predictor that shows a particular refactoring may improve the design and save the maintenance cost in the future. Besides the unanswered question of ROI, it lacks of risk factors consideration and misses the payback strategy over releases. Therefore, to reduce technical debt and to sustain software quality in software development continuously, a wise decision making should be made based on a cost-benefit analysis.

In this paper, we want to introduce an approach of technical debt management based on cost-benefit analysis. The remainder of this paper is organized as follows: Section II presents the research goals. Section III describes our approach to Technical Debt Management and its phases. Section IV discusses relevant related work and Section V concludes the paper.

II. Goals

In order to support software development organizations to systematically manage the TD of their software systems, we propose an approach called *Cost-Benefit based Technical Debt Management* (CoBeTDM). Its overall goal is to provide a framework to manage and reduces TD based on cost-benefit analysis for each release. To achieve this main goal, the following sub-goals should be fulfilled:

1) Provide a **debt item model** (see Table I) that comprises all information of code and architecture smells and the effort needed to resolve them.
2) Quantify **cost and benefit** for each possible refactoring of a particular debt item. This enables to select the “best” refactoring based on the expected ROI.
3) Provide a **structured process** on how to strategically pay back the TD based on quantified cost-benefit of refactoring effort either tactically or proactively.
4) Develop a **toolbox** to support TD management and to monitor the identified debt items.

CoBeTDM defines four phases as shown in Figure 1 (see Section III for details):

1) **Identification & Assessment**: Here, the focus is to identify and measure the worst smells as well as to model them by means of debt items.
2) **Monitoring**: In order to know the development of TD and its trend, it has to be monitored continuously.
3) **Quantification & Prioritization**: Based on a cost-benefit analysis of each possible refactoring associated with a debt item, the quantified refactorings are prioritized based on their ROI.
4) **Repayment**: Selected refactorings will be inserted into backlog for current or later releases in order to reduce the TD.

III. Cost-Benefit based Technical Debt Management (CoBeTDM)

Relevant and accurate data is needed to quantify TD related cost and benefit for a software system. It is to support managers to take the right decisions. To provide this data, a collection of metrics that characterize code and architecture smells could be applied.

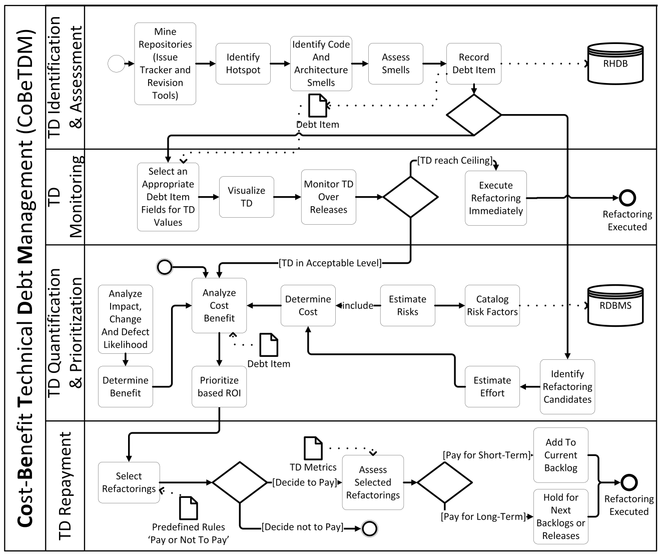

Figure 1. CoBeTDM Process

Modeling Debt Items. We propose a data structure (called *debt item*) to store all relevant and accurate information of all detected code and architecture smells. It will be stored in Release History Database (RHDB) - a database that stores data model that link between bug tracking system and versioning system. The data structure is depicted in Table I.

TABLE I. DEBT ITEM DATA STRUCTURE

Field	*Description*
Id	Unique identifier of debt item
Issue\Case	Task IDs or Case IDs, which represent a critical artifact (hotspot)
Dependency	Case IDs that depends on this debt item
Frequent Change	How many modifications have been made for one release?
Class	Class name
Code Smells	List of detected smells and its metrics values
Architecture-level	Architecture elements such as class, package, module or layer name
Architecture Smells	List of detected architecture smells and its metric values
Worst Smells	Sum of frequent change + code smells value + architecture smells value
Principal	Effort estimation to resolve this debt item
Interest	Extra effort estimation to resolve this debt item
Impact	Other artifacts that are impacted
When-to-Release	Release number
Responsible	A person or unit responsible for this debt item

A. TD Identification and Assessment

The identification of deficiencies of a software system is a must in the early phase of TD management. In CoBeTDM, the detection of bad smells is done in the following two steps: 1) **Hotspot detection**: Here, the goal is to find frequent changes, (i.e., unstable) software artifacts, which might be critical for the evolution of the system; 2) **Code and architecture bad smells detection**: For all identified hotspots, the worst code and architecture smells will be detected.

Hotspots detection. Hotspot detection is an approach to find the most critical artifacts of a software system. In this paper, the critical artifact means the module becomes unstable for certain releases, (i.e., frequent change over releases) and indicates strong increase in size and complexity (using metrics such as Lines of Code and McCabe Cyclomatic Complexity). It is important to detect the hotspot due to the symptom cost more than other code deficiencies. It is because we consistently have to pay back to tame it for every release. The hotspots detection can be supported by a dedicated mining repository approach where data from bug tracking and versioning tools are extracted, filtered and classified by tracking any frequent changes of contained artifacts. Currently, we manually map their IDs between Bugzilla and the Git repository. Then, we examine these artifacts by analyzing its size and complexity trend over releases. As a result, the artifacts that have many changes, (i.e., high maintenance activities) within the release could be detected as potential hotspots. We quantify criticality of an artifact by the number of changes that have been made, (i.e., Git log entries) performed for fixing bugs, (i.e., different severity levels of bugs) that were reported for specific releases. E.g., up to release 1, `CriticalPackage` of Application X got 200 modification from 130 bugs rated critical. Besides that, the identified artifact has a significant increase in size and complexity. From 4,000 LOC in release 0.9. increases to 10,000 LOC in release 1.0. Furthermore, the complexity of the package increases from 30 in release 0.9 to 50 in release 1.0. This symptom can be called as a critical artifact or hotspots.

Code Smells Detection. To detect code smells of the identified hotspots, we use a tool called *iPlasma* introduced by [3]. The tool shows a list of smells and its metric values. The highest metric values for each smell will be selected and prioritized. This data is recorded into a debt item to be used in the next phase. For instance, the `CriticalPackage` as detected as critical artifact previously will be assessed by *iPlasma*. The tool will detect any possible bad code smells. E.g., `CriticalPackage` contains `GodClass`, which has been detected as God class. The class has for example, 453 methods, defines 114 attributes and is more than 3500 lines long. It may also contain other smells, e.g., code duplication, data class etc., in this particular case, we focus on God class due to its refactor effort is quite high [4] compared to other smells. The tool will show relevant metrics for God class such as Access to Foreign Data (ATFD), Weighted Method Count (WMC) and Tight Class Cohesion (TCC). Each metrics value will be shown, e.g., as WMC (107), TCC (0.0) and ATFD (28). The metric values then will be recorded into *Code Smells* field in a debt item as shown in Table II.

Architecture Smells Detection. Next, the identified smells will be analyzed to detect architecture smells. The metrics introduced by [5] can be applied at class-, package- or subsystem-level. These smells can be detected by using existing tools such as Sonargraph-Architect [6]. The metric values produced by the architecture analysis tool will be stored as well into their respective debt items. In previous example, `CriticalPackage` was detected as critical artifact and contains `GodClass`. The class might contain cyclic dependency with other classes both within or outside the package. To detect the smells, the aforementioned tool can be used. For example, Sonargraph-Architect can detect it between classes or packages visually. It also displays the information regarding number of cycles and artifacts name. Then, cyclic value will be recorded into *Architecture Smells* field in the debt item.

TABLE II. DEBT ITEM EXAMPLE

Field	*Description*
Id	DI001
Issue\Case	#1234, #1235, #1236: Critical bugs of Application X
Dependency	#4321 #4322: Other critical bugs of Application X
Frequent Change	200 modification
Class	`GodClass`
Code Smells	God Class: WMC(107), TCC(0.0), ATDF(28)
Architecture-level	`CriticalPackage`
Architecture Smells	Cycle Dependency: Cyclic(10)
Worst Smells	200 + (107+0.0+28) + 10 = 345
Principal	16 hours (code smells) + 4 hours (cyclic dependency) = 20 hours: 1) *cost_to_split_a_class = 8 hours*. At least 2 classes will be partitioned for refactoring; It means 8×2; 2) *cost_to_cut_an_edge_between_two_files = 4 hours*. At least 2 files will be cut for refactoring. Both estimation based on [7]
Interest	2 hours, i.e., estimation extra work
Impact	15 classes and 2 packages
When-to-Release	Current: 1; Next: 1.1
Responsible	Mr. X

Assessing Bad Smells. After collecting the data from both code and architecture smells detection, we can analyze the obtained metric values to detect, which artifacts contain worst smells (i.e., highest in identified smells). For this means, we propose to apply the following formula *Worst Smells of Detected Critical Artifact = Most Frequent Changes + Highest Metrics of particular Code Smells + Highest Metrics of particular Architecture Smells*. See *Worst Smells* field in Table II. The worst smells value, then, will be compared with other debt items. The high value will be prioritized first instead of the low value. Besides that, the value could be used for TD monitoring as we explain in the next section.

B. TD Monitoring

To answer important questions such as: 1) How much TD do we have right now or in the current release?; 2) Is the TD at an acceptable level or not? 3) Does the TD continuously grows for each release?; 3) What is an acceptable **threshold** value of TD of each release? What is a maximum TD (**debt ceiling**) or minimum TD (**debt baseline**) for each release?; 4) How to react when the TD reaches the ceiling?; the TD has to be monitored continuously. Therefore, the TD data and its trend has to be visualized appropriately. The first idea is shown in Figure 2. Currently we are developing ideas and solutions for a systematic TD monitoring approach. Examples are: 1) A dedicated dashboard used to visualize TD data based on the managers' information needs. For example, the worst smells for certain release, high or low impact of debt item etc.; 2) A process to conduct semi-structured interview with managers or lead developers in order to gain information such as acceptable and minimum vs. maximum TD; 3) Development of a risk mitigation strategy framework that could be applied if TD reaches debt ceiling.

C. TD Quantification and Prioritization

In this phase, the debt items are quantified to perform a cost-benefit analysis. By cost, we mean the estimated effort and extra effort, (i.e., principal + interest) of a particular possible refactoring for a debt item. The cost value is stored together with the estimation risk, (i.e., judgment by experts) that may resulted from the refactoring. Then, it should be cataloged and stored in the database, (e.g., RDBMS) in order to be referred in the future. Then, the benefit is estimated, i.e., the less effort of refactoring, which gives positive impact. Currently, the benefits values are estimated based on the impact analysis in particular dependency analysis. In addition, we also add defect and change likelihood as properties, while calculating benefits. The less value of both likelihood are potentially has less frequent of the same symptoms in the future. It means that the refactoring effort for maintenance and correction will decline.

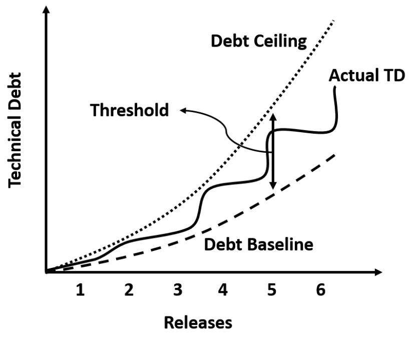

Figure 2. TD Trend over Releases

Firstly, analyzing the changes that could be affected by the dependency of artifacts, (e.g., classes or packages) on particular refactoring candidate - impact analysis. The changes might alter and create new artifacts for e.g., operations, classes or packages, which require a cost to do that. Therefore, the more dependencies the artifacts are, the more cost should be spent. For e.g., see `DI001` in Table III, `GodClass` depends on the other five classes and two external packages. Two points or weight will give to the fifteen classes and five points to 2 packages, (i.e., (2×15) + (5×2) = 40 points) as shown in *Refactoring Impact* column in Table III. Secondly, the defect likelihood could be analyzed by computing on how many defect fixes affected by the detected smells. The likelihood could be computed by detecting the smells, (e.g., specifically the god class) from certain periods, (e.g., from April to July). Then, count the number of defects that lead to fix in the god class in this time period and divide by the number of all defects that were fixed in the same time period. The higher the value the more likely a defect will be indicated in the god class. For instance, see column *Defect Likelihood* for `DI001`, it has been detected that the `GodClass` was god class from particular period. Assume the likelihood of 0.5, it means every second fixed defect will lead to changes in this god class. Thirdly, the change likelihood could be analyzed by computing on how likely a class is to be modified when a change to the software is executed. The same computation method will be used as defect likelihood for this purpose. It means the higher the value, the more likely that maintainability effort is higher for the god class [4]. For example, if change likelihood of 0.1 shows that the class was, on overage, modified with every 10th change to the software. By computing the impact analysis, defect and change likelihood represent as a weight, it will, then, multiply by raw benefit. The raw benefit is an effort estimation that can be saved in terms of maintenance work in the next release. An expert will give this raw estimation. Then, the total benefit will be calculated. Based on the estimated cost and benefit values, the ROI value is calculated by (adopted from [8] where ***ROI = (Saving Effort and Less Impact of Proposed Refactoring/Effort of Proposed Refactoring))***, i.e., ratio of total Benefit to the total Cost. If the ROI value is greater than or equal to one, the refactoring is cost effective,

TABLE III. COST-BENEFIT ANALYSIS EXAMPLE

No.	Debt Item ID	Refactoring	Cost (Principal + Interest)	Risk (R) in Hour	Total Cost (Cost + R)	Refactoring Impact (RI)	Change Like. (CL)	Defect Like. (DL)	Weight (RI X CL X DL)	Raw Benefit (RB) in Hour	Total Benefit (Weight X EB)	ROI (TB / TC)	Rank
1.	DI001 -God Class -Cyclic Dependency	-Extract Class -Cut Dependency	22	-Regression bugs (2) -Testing (2)	26	40	0.5	0.1	2	5	10	0.4	1
2.	DI002 -Long Method Class -Inheritance too Deep	-Extract Method -Delegation	10	-Merge conflict (3) -Testing (2)	15	15	0.1	0.3	0.45	5	2.25	0.3	2
3.	DI003 -Duplicate Code -Cyclic Dependency	-Extract Method -Cut Dependency	8	-Build breaks(2) -Testing	12	10	0.25	0.11	0.28	3	0.84	0.06	3

i.e., the debt is paid off. Finally, the ROI values are prioritized. Based on the example (see Table III), `DI001` seems promising to be paid first instead of `DI002` and `DI003`. The ROI `DI001` value is bigger than the latter, (i.e., the refactoring effort could reach ROI) and it may give positive impact to the system.

D. TD Repayment

In the last phase, refactorings, which has been prioritized in the previous phase are added to the current backlog of the software system. By implementing the refactoring, the gap between the software as "it is" and the hypothesized "ideal" state could be closed. Although, there is no general agreement that refactoring could realize the idea, [9] claimed that by applying Test-Driven Development and continuous refactoring, the TD could be reduced systematically by releases. But, the questions "Which refactoring should be implemented first or later?" and "Should pay or not to pay?" are still open. Currently, we are still investigating how to strategically pay back based on TD metrics as introduced by [10].

IV. Related Works

Technical debt management. A few researchers have been focusing on how to manage TD. For example, [11] proposed a TD management framework, which aids managers to decide, which items should be implemented either first or later. A simple cost-benefit analysis is applied and less impact to the project is put at the top, i.e., prioritization. However, the approach does not consider risk factors in estimating the cost. Unlike CoBeTDM, it integrates risk factors [12] in the analysis due to uncertainty that may always happen. [13] introduced a tool to manage TD in terms of code violations. It guides to select the smells that should be refactored first based on pyramid data - the lowest part needs to be considered first. In contrast, CoBeTDM considers not only code but also architecture smells as the latter ones have high negative impact on the software quality.

Hotspot, code-, architecture-smells detection. We have adopted existing metrics [3] [5], which are quite useful to characterize smells on code- and architecture-level. However, these metrics do not integrate with each other. Our approach combines both metric sets to determine worst smells and identify very critical artifact as proposed by [14] for hotspot detection.

V. Conclusion

This paper introduces a TD management framework based on cost-benefit analysis, called CoBeTDM. It offers a systematic way of reducing the technical debt by quantifying cost and benefit of refactorings. It also considers with relevant risk factors. Until now, the CoBeTDM process is performed manually. But, we have started to develop a toolbox to support CoBeTDM and to monitor the TD trend in order to react early enough if the TD becomes critical.

References

[1] CAST, "Cast Worldwide Application Software Quality Study: Summary of Key Findings," 2010.

[2] C. Seaman, Y. Guo, N. Zazworka, F. Shull, C. Izurieta, Y. Cai, and A. Vetro, "Using technical debt data in decision making: Potential decision approaches," in 2012 Third Int. Workshop on Managing TD (MTD). IEEE, Jun. 2012, pp. 45–48.

[3] M. Lanza and R. Marinescu, OO Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of OO Systems. Springer, 2006, ISBN: 978-3-540-24429-5.

[4] N. Zazworka, C. Seaman, and F. Shull, "Prioritizing design debt investment opportunities," in Proceeding of the 2nd working on Managing technical debt - MTD '11. New York, New York, USA: ACM Press, May 2011, p. 39.

[5] M. Lippert and S. Roock, Refactoring in Large Software Projects: Performing Complex Restructurings Successfully. Wiley, 2006, ISBN: 978-0-470-85892-9.

[6] Hello2morrow, "Sonargraph Architect," 2013, URL: https://www.hello2morrow.com/products/sonargraph/architect [accessed: 2015-08-09].

[7] Sonarqube, "Technical Debt Calculation," March 09, 2011, URL: http://docs.sonarqube.org/display/PLUG/Technical+Debt+Calculation [accessed: 2015-09-08].

[8] R. Leitch and E. Stroulia, "Assessing the maintainability benefits of design restructuring using dependency analysis," in Proceedings. 5th Int. Workshop on Enterprise Networking and Computing in Healthcare Industry). IEEE Comput. Soc, 2003, pp. 309–322.

[9] J. Kerievsky, Refactoring to Patterns. Pearson Higher Education, 2004, ISBN: 0321213351.

[10] N. Ramasubbu, C. Kemerer, and C. Woodard, "Managing Technical Debt: Insights from Recent Empirical Evidence," IEEE Software, vol. 32, no. 2, Mar 2015, pp. 22–25.

[11] Y. Guo, R. O. Spínola, and C. Seaman, "Exploring the costs of technical debt management a case study," Empirical Software Engineering, Nov. 2014.

[12] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and benefits at Microsoft," IEEE Transactions on Software Engineering, vol. 40, no. 7, 2014, pp. 633–649.

[13] J.-L. Letouzey and M. Ilkiewicz, "Managing TD with the SQALE Method," IEEE Software, vol. 29, no. 6, Nov. 2012, pp. 44–51.

[14] M. DAmbros, H. Gall, M. Lanza, and M. Pinzger, "Analysing Software Repositories to Understand Software Evolution," in Software Evolution SE - 3. Springer Berlin Heidelberg, 2008, pp. 37–67.