An Empirical Model of Technical Debt and Interest

Ariadi Nugroho
Software Improvement Group
P.O. Box 94914 1090 GX
Amsterdam, The Netherlands
a.nugroho@sig.eu

Joost Visser
Software Improvement Group
P.O. Box 94914 1090 GX
Amsterdam, The Netherlands
j.visser@sig.eu

Tobias Kuipers
Software Improvement Group
P.O. Box 94914 1090 GX
Amsterdam, The Netherlands
t.kuipers@sig.eu

ABSTRACT

Cunningham introduced the metaphor of technical debt as guidance for software developers that must trade engineering quality against short-term goals.

We revisit the technical debt metaphor, and translate it into terms that can help IT executives better understand their IT investments. An approach is proposed to quantify debts (cost to fix technical quality issues) and interest (extra cost spent on maintenance due to technical quality issues). Our approach is based on an empirical assessment method of software quality developed at the Software Improvement Group (SIG). The core part of the technical debt calculation is constructed on the basis of empirical data of 44 systems that are currently being monitored by SIG.

In a case study, we apply the approach to a real system, and discuss how the results provide useful insights on important questions related to IT investment such as the return on investment (ROI) in software quality improvement.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Cost Estimation*

General Terms

Economics, Management, Measurement

Keywords

Estimation, Measurement, Cost, Maintenance, Effort, Software Economics

1. INTRODUCTION

The metaphor of technical debt first appeared at OOPSLA 1997 and was coined by Ward Cunningham in his experience report [8]. The metaphor describes a phenomenon in which technical quality issues in a software system can lead to future problems if not resolved immediately. At the software development level, the metaphor provides useful guidance when a trade off of code quality must be made against another factor, e.g., delivering functionality more quickly to achieve time to market objectives.

From an IT management point of view, the technical debt metaphor links the notion of technical quality of software to business value. Technical debt is not visible and has a negative value to the business [4]. This is particularly true because the technical quality of software is not something that is directly observable—although its effects have been shown to be significant on some quality aspects such as software maintainability and reliability [9].

This paper looks into the technical debt metaphor and clearly defines the notions of technical debt and interest. The term technical debt is defined as the cost to improve technical quality up to a level that is considered ideal. The interest of technical debt is the extra cost spent on maintaining software as a result of poor technical quality.

An approach to quantify technical debt and interest is developed. By applying the approach to software systems, IT managers can answer questions about their IT investment:

- How large is my technical debt?
- How much interest am I paying on the debt?
- Is the debt growing, and how fast? What is the consequence of holding onto a debt for future maintenance?

The advantage of the proposed approach is that it is based on a sound and robust quality assessment method of software quality. Furthermore, the approach uses data of 44 systems to empirically construct an estimation model that becomes the core part of the technical debt calculation. A real software system is used as a case study to show how the approach can be used to answer important questions related to IT investment.

The organization of the paper is as follows. In Section 2, the notions of technical debt and interest are defined, and an approach to quantify them is discussed. Section 3 discusses the application of the proposed approach to an actual system. In Section 4, the advantages and limitations of such an approach are discussed. Section 5 discusses related work and finally, in Section 6, conclusions and future work are outlined.

2. DEFINING AND MEASURING TECHNICAL DEBT

In this section, the notion of technical debt will be discussed. In particular, the technical debt metaphor will be

MTD'11, May 23, 2011, Waikiki, Honolulu, HI, USA

evaluated from the perspective of software economics, leading to the definition of its terms. Furthermore, an approach to measure technical debt model will be proposed.

2.1 Redefining the Metaphor

There have been some initiatives to define the notion of technical debt. These initiatives are concerned with challenges faced by software projects to deliver quality software that deliver added values to organizations.

Steve McConnell classified technical debt into two basic forms: unintentional and intentional [17]. Unintentional debts generally result from poor coding practices. Intentional debts, on the other hands, are debts committed with informed decisions. These could be classified further into *short term* (e.g., to quickly release software into production) and *long term* (e.g., improvement that is planned gradually). Martin Fowler suggested a somewhat similar categorization of technical debt by considering the intention (deliberate or inadvertent) and awareness (reckless or prudent) of committing to a debt [11]. These perspectives on technical debt provide a better understanding about the nature of debts, precautions and actions to avoid and mitigate them.

At the level of IT management, organizations are interested in the economic consequences of technical debt and business risks that they may pose. Neglecting technical debt can increase inefficiency in running software systems and create difficulties in extending them to support business operations. Cost overhead caused by such issues is considered as the interest of technical debt. Following this economic perspective, the notion of technical debt is defined as: *the cost of repairing quality issues in software systems to achieve an ideal quality level*—in practical terms, an ideal quality level represents the highest achievable level of quality defined in a software quality model adopted by an organization. This type of repair involves changes that are not always desired by organizations because the problems and their impacts are not visible in the short term. The amount of debt will depend on the gap between the current quality level and the ideal one. Technical debt grows over time if not resolved because the quality of changes performed on top of systems of poor technical quality tend to be poor [10].

The corresponding notion of interest is defined as *the extra maintenance cost spent for not achieving the ideal quality level*. Maintenance encompasses activities such as adding new functionality and fixing bugs. Maintenance is different from technical quality repair in that it involves visible changes and their impacts are immediately visible (e.g., fixing bugs or adding new features). Extra effort spent on retrofitting new functionality or fixing bugs are examples of interest on technical debt. Hence, the interest of technical debt is not the same as maintenance costs. Systems without technical issues will still spend some effort on maintenance.

The concepts of debt and interest are fundamental to the notion of technical debt in this paper. In looking at technical debt, this paper focuses on technical quality (i.e., maintainability). Obviously, technical debt can also be assessed from other perspectives such as functional quality. In Figure 1, these concepts are visualized. The figure shows the nature of technical debt that grows over time if not addressed. As the technical debt grows, the interest will also grow. Without technical debt, maintenance costs remain stable at an optimal level.

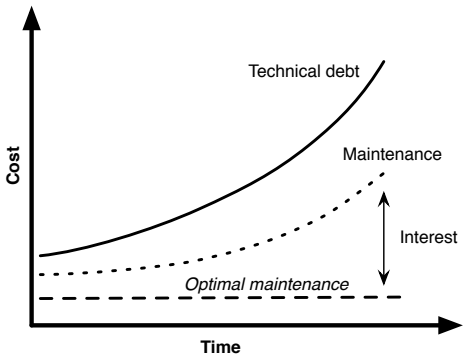

Figure 1: Technical debt and its interest grow over time if not resolved

2.2 Software Maintainability as a Measure of Technical Debt

To calculate technical debt, a method is needed to determine the level of quality of a software system. Additionally, an "ideal" level of quality should be determined. To this aim, SIG/TÜViT's software quality assessment method is used. This method is developed as a layered model for measuring and rating the technical quality of a software system in terms of the quality characteristics of ISO/IEC 9126 [14].

In the first layer, source code analysis is performed to collect measurement data about the software system. The analysis involves well-known metrics such as lines of code (LOC), code duplication, McCabe's cyclomatic complexity, parameter counts, and dependency counts. These metrics are collected at the low level system elements such as lines, units (e.g., methods or functions), and modules (e.g., files or classes). For more details on how the metrics are measured please refer to [13].

Subsequently, these metrics are mapped onto ratings for properties at the level of the entire software product, such as volume, duplication, and unit complexity. These ratings take values in the interval between 0.5 and 5.5, which can be rounded to an entire number of stars between one and five. This constitutes a unit-less ordinal scale that facilitates communication and comparison of quality results at the level of entire software products.

The mapping functions for volume and duplication are straightforward translations of LOC-based metrics to ratings. The remaining mapping functions make use of *Risk Profiles* as intermediate device. Each profile is a partition of the volume of a system (measured typically by LOC) into four risk categories: low, moderate, high, and very high risk. For example, if 4,000 LOC of a 100,000 LOC system sit in methods with a McCabe number higher than 50, the volume percentage in its very high-risk category is 4.0%. Risk thresholds for metric values (e.g., McCabe > 50) have been chosen based on statistical study of a large set of representative systems [1]. Likewise, thresholds have been chosen to define a mapping of risk profiles to quality ratings.

For the extraction of measurement values from source code, the Software Analysis Toolkit (SAT) of SIG is used. The SAT offers source code analysis capabilities for a wide range of languages as well as generic modules for aggregat-

	maximum relative volume		
rating	moderate	high	very high
⋆ ⋆ ⋆ ⋆ ⋆	25%	0%	0%
⋆ ⋆ ⋆ ⋆	30%	5%	0%
⋆ ⋆ ⋆	40%	10%	0%
⋆ ⋆	50%	15%	5%
⋆	-	-	-

Table 1: Example of risk profile and its mapping to star ratings

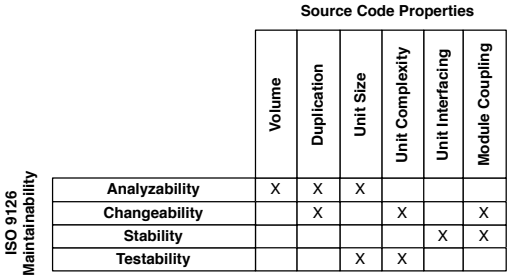

Figure 2: SIG's quality model for software maintainability

ing and mapping source code metrics in accordance with the SIG quality model.

Source code property ratings are mapped to ratings for ISO/IEC 9126 sub-characteristics of maintainability following dependencies summarized in a matrix (see Figure 2). In this matrix, a × is placed whenever a property is deemed to have an important impact on a certain sub-characteristic.

The sub-characteristic rating is obtained by averaging the ratings of the properties where a × is present in the sub-characteristic's line in the matrix. For example, *changeability* is influenced by *duplication*, *unit complexity* and *module coupling*, thus its rating will be computed by averaging the ratings obtained for those properties. Finally, all sub-characteristic ratings are averaged to provide the overall *maintainability* rating.

The quality model has been used by SIG/TÜViT to perform certifications, assessments, and monitoring of software quality [2]. The model is calibrated yearly and the data set used for the calibration has now reached more than 170 systems. The calibration allows adjustment to the metric thresholds so that their representativeness of the quality of contemporary software systems is sustained.

2.3 Quantifying Technical Debt

As discussed previously, technical debt represents the cost of improving the quality of software to the level that is deemed ideal. When estimating the cost of repair, what is being estimated is actually the *Repair Effort (RE)* spent to perform the repair work. If repair work is performed to increase quality to the ideal level, then RE will represent the amount of technical debt.

Two factors are taken into account in estimating RE, namely *Rework Fraction* and *Rebuild Value*.

1. **Rework Fraction (RF)**. RF is an estimate of the percentage of lines of code that need to be changed to improve the quality of software to a higher level. The mapping between quality changes and the estimated RF is provided in Table 2. The way in which the values are derived is described in Section 2.3.1.

2. **Rebuild Value (RV)**. RV is an estimate of effort (in man-months) that needs to be spent to rebuild a system using a particular technology. This is determined by the following formula:

$$RV = SS * TF$$

System Size (SS) represents the total size of a system measured in lines of code. Alternatively, SS can be measured using functional size (i.e., Function Points). *Technology Factor (TF)* represents language productivity factor. This factor provides a conversion from source code statement to effort (i.e., man-months per source statement) through 'backfiring'. Please refer to [15, 18] for further discussion on the subject.

Target/Source	1-star	2-star	3-star	4-star	5-star
1-star					
2-star	60%				
3-star	100%	40%			
4-star	135%	75%	35%		
5-star	175%	115%	75%	40%	

Table 2: Estimated Rework Fraction

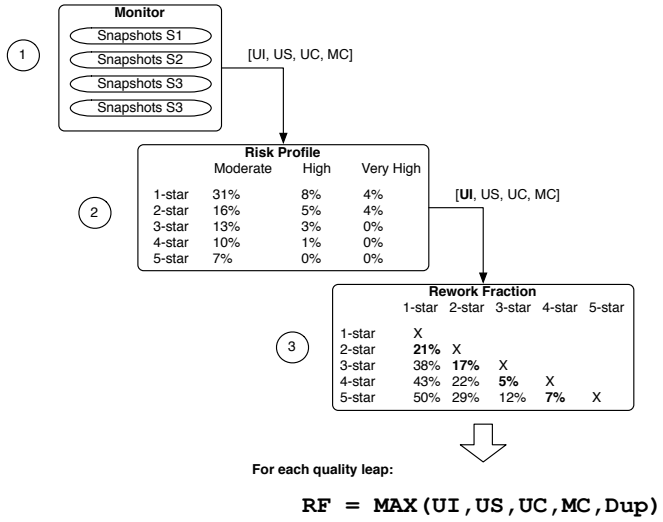

Figure 3: Approach for estimating Rework Fraction from the Software Monitor

2.3.1 Estimating Rework Fraction

Table 2 provides estimates of RF for improving the quality of software based on SIG's software maintainability rating. The figures represent the percentage of lines of code that will need to be changed in order to raise the quality by one level. For example, to improve a system from 2-star to 3-star rating would require changes as much as 40% of the code. The steps in which the percentages are empirically determined are visualized in Figure 3.

Estimating RF begins by collecting data of systems being monitored (step 1). Forty-four systems are used in the analysis, and in total there are 969 system snapshots. The

	Risk Categories		
	Moderate	High	Very High
Propagated	3%	0%	-
1-star	31%	8%	4%
2-star	16%	5%	4%
Changed	18%	3%	0%

Table 3: Calculating rework fraction for UI metric to increase quality level from 1-star to 2-star

period when these snapshots were taken is between 2008 and 2010. Note that for this study only Java systems are used in the analysis. Systems built using other technologies might have significantly different rework characteristics, and thus are excluded to avoid biases.

For each snapshot of every single system, Risk Profiles of the following metrics are collected: Unit Interfacing (UI), Unit Complexity (UC), Unit Size (US), Module Coupling (MC), and Duplication (Dup). The percentages in the Risk Profile (step 2) represent the mean percentage of code that falls into three risk categories (Moderate, High, Very High) for each maintainability rating (1 - 5). These risk categories are determined based on different thresholds for different metrics (recall the discussion in Section 2.2). Note that Duplication has no risk profile because it is measured at the system level.

RF is determined by calculating the difference in Risk Profile between two maintainability ratings. Consider the example to obtain RF for UI metric in Table 3. At the 1-star level, 4%, 8%, and 31% of LOC are in the very high, high, and medium risk category, respectively. To increase the quality level from 1-star to 2-star, there is no change in the very-high risk category, and thus 0% of the code is propagated to the high risk category (it is assumed that a percentage of code can only be propagated to the next immediate risk category). However, in the high risk category, 3% of the code will need to be changed ($8\% - 5\%$), and this amount of code will be propagated to the moderate risk category. This propagation leaves 34% of the code ($31\% + 3\%$) in the moderate risk, and to move to 2-star level, 18% of the code will need to be changed and then propagated to the lower risk category. Using this method, it can be estimated that to move from 1-star to 2-star, 21% of the code ($18\% + 3\%$) will need to be changed.

For Duplication, estimating rework fraction is done by calculating the average difference of the percentage of duplicated code between two quality levels at the system level. For example, if, on average, 3-star and 4-star system snapshots have 10% and 3% duplicated code respectively, then it is inferred that the amount of code that needs to be changed/refactored to improve the level of quality from 3-star to 4-star would be 7% of the total LOC.

Finally, the value of RF for a quality change is obtained by taking the maximum value of RF from the measured metrics for a given quality change. The principal quality changes are 1–2, 2–3, 3–4, and 4 to 5-star. The maximum values of these quality changes are the backbone of the RF table, based on which RF of other quality changes (e.g., 1–3) can be easily calculated—that is, by adding the RF of 1–2 and 2–3.

The *max* function is used based on the assumption that the maximum percentage of code changes would also include other changes in other metrics. Such assumption will result in an optimistic estimation.

An alternative to using the maximum value would be summing all the percentages of change. This approach, however, will result in a pessimistic estimation. Another alternative, namely using the average value tends to give an overly optimistic estimation. Thus, using the maximum value is a midway of these alternatives.

Tech.	**Source**	**Target**	**SS**	**RV**	**RE**
Java	⋆ ⋆	⋆ ⋆ ⋆	136.5K	186.7	67.2
Cobol	⋆ ⋆	⋆ ⋆ ⋆	275.7K	316.4	113.9
C++	⋆ ⋆	⋆ ⋆ ⋆	317.4K	332.4	119.7
Java	⋆ ⋆	⋆ ⋆ ⋆ ⋆ ⋆	136.5K	186.7	193.3
Cobol	⋆ ⋆	⋆ ⋆ ⋆ ⋆ ⋆	275.7K	316.4	327.5
C++	⋆ ⋆	⋆ ⋆ ⋆ ⋆ ⋆	317.4K	332.4	344.1

Table 4: Examples of Repair Effort (RE) (in man-months) for different technologies (Functional size = 3000 FPs; RA = 10%)

The estimated RF values in Table 2 also indicate the amount of technical debt as a percentage of system size. Assuming that the ideal level of quality is 5-star, the technical debt of a software system with a 2-star quality level would be equal to the cost of repair to improve the level of quality to 5-star. In this case, the technical debt is 115% of the total system size.

2.3.2 Calculating Repair Effort

Having defined RV and estimated RF, Repair Effort (RE) can be calculated as follows:

$$RE = RF * RV$$

Some factors such as team experience and the use of advanced tooling typically increase productivity in refactoring a system. For example, in a context where a refactoring team is highly experienced and equipped with advanced tooling, RE can be discounted as much as 20%-30% of the original value. These context-specific aspects of a project are not accounted for in the basic RE calculation, and therefore an adjustment to RE is introduced. This adjustment is called *Refactoring Adjustment (RA)*, and it is incorporated in the RE calculation as a percentage discount:

$$RE = RF * RV * RA$$

To try the RE calculation, let us consider an example of a Java system with a functional size of 3000 function points (136.5 KLOC) that currently scores 2-star for its maintainability rating. There is an urge to improve the quality of the system, and the system owner wants to know the cost of such improvement. Let us assume that the system is going to be improved to 3-star. Furthermore, the renovation project is equipped with advanced tooling, and this is expected to reduce the effort as much as 10%. The first row in Table 4 shows the repair effort (RE) to increase the quality level to 3-star ($TF_{java} = 0.00136$). For the sake of comparison, examples of different technologies with a similar functional size are also provided in the table (note that the Function Point to source statement conversion is based on the SPR Programming Language Table [18]).

The result of calculating RE for the Java system (i.e., from 2-star to 3-star), as depicted in the first part of Table 4, shows that it would take 67 man-months (5.6 man-years) to do the repair. The table also shows that Cobol

and C++ systems with similar functional size and quality target require more effort to repair—that is, 113.9 and 119.7 man-months respectively.

The second part of the table shows RE for the three technologies if the ideal level of quality (i.e., 5-star) is to be achieved. In other words, the RE in the second part of the table represent the technical debt of the systems given their current level of quality is 2-star.

In practice it is often the case that software systems are composed of different technologies. For such systems, repair effort would be the sum of repair effort of the respective technologies.

2.4 Quantifying the Interest

It has been discussed earlier that the interest of technical debt is the *extra cost* spent on maintenance due to technical quality issues in the code. Hence, it follows that the interest of technical debt is the difference in maintenance effort between a particular quality level and the ideal level. In the case of SIG's quality model, 5-star would be the ideal quality level and as such, the interest would be the difference between maintenance effort spent at the 5-star level and any of the lower quality levels.

To estimate *maintenance effort (ME)* at various quality levels, the following formula is used:

$$ME = \frac{MF * RV}{QF}$$

The above formula shows that ME is a function of *Maintenance Fraction (MF)*, Rebuild Value (RV), and *Quality Factor (QF)*. MF represents the amount of maintenance effort spent on a system in a yearly basis, measured as a percentage of lines of code that is estimated going to change (added, modified, or deleted) yearly due to maintenance. Essentially, MF can be measured based on historical maintenance data, which can be different from system to system. Nevertheless, historical data of systems being monitored by SIG suggests that yearly maintenance involves roughly 15% changes in the code.

QF is a factor that is used to account for the level of quality. It is assumed that the higher the level of quality, the smaller is the effort that needs to be spent on maintenance. This assumption is justified by previous research, which reveals that performing changes on systems with higher code quality is more efficient (see for example in [12, 6]). QF is determined using the following formula:

$$QF = 2^{((QualityLevel-3)/2))}$$

The above formula is a simplified model to account for the level of quality. The formula gives the following factors for quality level from 1-star to 5-star respectively: 0.5, 0.7, 1.0, 1.4, 2.0. These factors are in line with the findings of earlier work reported in [3]. Using these factors as denominators in the equation gives diminishing values for ME as the level of quality increases.

RV is calculated the same way as described in Section 2.3.

To put the above formulas into perspective, let us consider the Java system example given previously. Given the same system characteristics, and assuming 15% yearly maintenance and 5-star is the highest level of quality, the interest (i.e., the extra cost spent on maintenance) paid at the 2-star quality level can be calculated as follows:

$$ME_{2-star} = \frac{15\% * 186.7}{0.7} = 40$$

$$ME_{5-star} = \frac{15\% * 186.7}{2} = 14$$

The above calculations show that the maintenance effort spent at the 2-star level is nearly three times as much as that spent at the 5-star level. Having said that, it can be concluded that at the current quality level (2-star), the Java system has 193 man-months worth of debt, and with this amount of debt, 26 man-months $(40 - 14)$ extra effort would be spent on maintenance on a yearly basis. Assuming that a maintenance staff costs $100,000 yearly, then the technical debt and its annual interest would be $1,608,333 and $216,666 respectively.

Recall earlier discussion about the growth of technical debt over time. One should be aware that as the technical debt grows, the interest will also grow. The growth of technical debt can be represented as growth in system size. Hence, for time t and growth rate r the RV of a system can be defined as follows:

$$RV = SS * (1 + r)^t * TF$$

As an example, consider 5% of the 15% yearly maintenance will lead to an increase in the technical debt. The RV of the Java system after three years would be:

$$RV_{t=3} = 136,500 * (1 + 0.05)^3 * 0.00136$$

After three years, if the quality of the Java system remains at 2-star level the RV will grow to 216.1 man-months (initially 186.7 man-months). Following this growth, the technical debt and its annual interest after three years would be $1,810,000 and $276,666 respectively.

3. CASE STUDY

In this section a case study is presented to illustrate how the proposed approach can be applied to answer practical questions related to software quality improvements.

The case study is a system that was recently assessed by SIG. Its main functionality is for designing transportation infrastructure. When the assessment was done the system was 18 years old. The system was developed using various technologies including Java, C, C#, C++, PHP. The size of the whole system is 760,000 lines of code.

Following the software assessment, one of the subsystems—let us call it subsystem X for the rest of this paper—was put into SIG's software monitor as a follow up of SIG's recommendation to improve the quality of the system. This monitoring provided specific and repeated measurements on subsystem X, and for this reason subsystem X was used as a case study rather than the entire system. Subsystem X was developed in Java and when first monitored its size was 125,345 LOC, which amount to roughly 2754 FPs.

The result of the quality assessment revealed that subsystem X has a 3-star quality level. Using the proposed approach to calculate technical debt and its interest, the following questions will be addressed:

- What is the current technical debt and interest, and how much will they be in 5 years?

		Year									
		1	2	3	4	5	6	7	8	9	10
3-star	Debt*	102.9	109.3	116.1	123.4	131.1	139.2	147.9	157.2	167.0	177.4
	Interest*	11.1	12.5	14.0	15.6	17.3	19.0	20.9	22.9	25.0	27.3
	Maintenance effort*	22.3	23.7	25.2	26.7	28.4	30.2	32.1	34.1	36.2	38.4
	Required resources **	1.9	2.0	2.1	2.2	2.4	2.5	2.7	2.8	3.0	3.2
4-star	Debt*	54.9	58.3	61.9	65.8	69.9	74.3	78.9	83.8	89.1	94.6
	Interest*	4.6	5.6	6.6	7.8	8.9	10.2	11.5	12.9	14.4	16.0
	Maintenance effort*	15.8	16.7	17.8	18.9	20.1	21.3	22.7	24.1	25.6	27.2
	Required resources**	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.0	2.1	2.3
	Repair Effort*	48.0									
	ROI	-86.4%	-72.0%	-56.6%	-40.3%	-23.0%	-4.6%	15.0%	35.8%	57.8%	81.3%
	Cash flows***	-345.6	57.8	61.4	65.2	69.3	73.6	78.2	83.1	88.3	93.8
	NPV***	-337.2	-282.2	-225.2	-166.1	-104.8	-41.3	24.4	92.6	163.3	236.6

* in man-months; ** in man-years; *** in thousands of dollars

Table 5: Technical debt of subsystem X on a 10-years horizon

- How much should be invested in order to improve the system quality to 4-star, and how long does it take to recoup this cost?
- Given that there are a fixed number of resources to perform yearly maintenance (let us assume 2 FTE), what would be the consequences of not investing in repairing the system in the long run?

Table 5 provides the results of technical debt calculation of subsystem X in a 10-years horizon. The table provides two scenarios: 1) keeping the quality level at 3-star level; 2) improving the quality level to 4-star. In the last row of the table, financial projections of return on investment (ROI) and net present value (NPV) for investing in quality improvement to the 4-star level are provided.

It is assumed that yearly maintenance (MF) is 13%, and 48% of this maintenance effort leads to system growth (i.e., 6.2% annually). These figures are estimated based on historical data of subsystem X. Furthermore, considering the development environment (e.g., the availability of refactoring tools), it is assumed that code quality improvement will require 20% less effort than a rewrite (RA=20%).

3.1 How Much Debt Does the System Have?

As discussed previously, to calculate the technical debt at a certain level of quality an ideal level of quality should be defined. In this case, 5-star is used as the ideal level of quality. Given this assumption, the results in Table 5 show the amount of technical debt at the 3-star and 4-star level.

Referring to Table 2, the amount of rework (RF) to improve the quality from 3-star to 5-star is 75% of the total system size. Using the formulas defined in Section 2.3.1, the technical debt at the 3-star level is determined to be 102.9 man-months ($857,500). Taking into account system growth (i.e., 6.2 % annually), the technical debt will grow to 131.1 man-months ($1,092,500) in the 5th year.

Similarly, Table 2 also shows the amount of interest—that is, the amount of extra effort spent on maintenance at both 3-star and 4-star quality levels. At the 3-star level, the extra effort spent on maintenance in the first year is 11 man-months ($91,666), and in five years this amount will increase to 17 man-months ($141,666).

3.2 How Much Should be Invested in Repair? When Will it Pay Back?

Previously, the technical debt at 3-star level was determined, which is basically equal to the amount of investment needed to improve the level of quality to 5-star. Another scenario is to improve the quality to 4-star level, which can be preferable given the lower amount of investment required (i.e., only 35% of LOC need to be touched).

Table 5 shows that the required RE to improve system quality to 4-star is 48 man-months ($400,000). The table also shows the debt and interest need to be paid over 10 years. It is shown that the amount of debt and interest over time at the 4-star level are nearly half that of 3-star. By calculating the saving on interest as a result of moving to 4-star, and also taking into account the repair effort, the return on investment (ROI) can be determined. Table 5 shows that the repair effort invested to improve the quality level to 4-star will pay back in terms of positive ROI of 15% in seven years.

Another way of assessing the worthiness of an investment is by looking at its net present value (NPV). NPV is the sum of present values of future cash flows given an interest rate (i.e., the rate of return that can be obtained on an investment with similar risk). Hence, NPV is the present value of savings obtained from lower maintenance minus the initial repair cost. Table 5 shows that the NPV of the investment is positive if the system will have a lifespan of at least another seven years ($24,464)—note that the NPV with an n-year horizon is calculated by assuming that the system is going to be decommissioned after year n.

Figure 4 visualizes the amount of repair effort and ROI if the quality of the system is to be improved to 4-star. The RE and ASM lines represent the repair effort and accumulative saving on maintenance effort at a given year, respectively. The RE1 line represents the one-off repair effort in the first year for improving system quality. To determine the break-even point, one should look at the intersection between RE1 and ASM lines. Figure 4 shows that the break-even point of the investment is somewhere between year six and seven. Note that after the first year the RE line in Figure 4 indicates the increasing amount of repair effort over time, and thus it is a projection of repair effort *if* the repair is to be done later in the future.

3.3 What Is the Consequence of Not Repairing for Maintenance Capability?

In practice, it is common to have a fixed amount of resources devoted to perform maintenance. For subsystem X, let us assume that two staff members were assigned to perform regular maintenance, which generally include bug-fixing and implementation of new features.

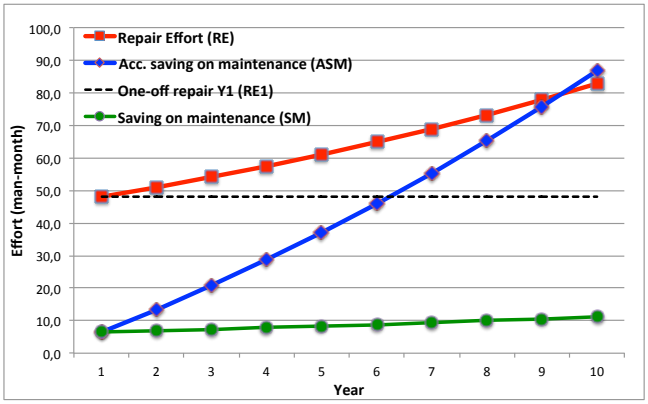

Figure 4: Repair effort and ROI of increasing the level of quality from 3-star to 4-star level

Table 5 shows the estimated amount of maintenance effort each year and the required resources to perform the maintenance activities. At the 3-star quality level, in the first year two people performing maintenance is still sufficient. However, the required resources increase over time because the amount of maintenance is also expected to increase due to the increasing technical debt. In the second year, available resources are all consumed for performing maintenance, and in the third year the situation gets worse because the required resources are higher than resources that are available. By improving the level of quality to 4-star, maintenance activities can still be accomplished with 2 FTE until year 8. Considering that a resource buffer is crucial, project managers can use such resource usage estimation to adjust yearly maintenance effort accordingly.

4. DISCUSSION

In this section, the advantages of the proposed approach to quantify technical debt, its limitations, and the contributions of this paper will be discussed.

The main advantages of the technical debt estimation proposed in this paper are three-fold. First, it is based on a sound and quantitative approach for measuring software quality from source code. A sound and objective approach to measure the quality level of software systems is a fundamental factor for the estimation of technical debt. Furthermore, having a clear and justifiable quality model behind the measurement helps in reasoning and providing advices for quality improvements. The second advantage is that the estimation of technical debt is based on empirical data. Snapshots of systems that are currently under monitoring are used to analyze the amount of effort that needs to be spent on quality improvement to achieve a certain level of quality. Such an empirical approach provides more confidence on the reliability of the estimation. Finally, the proposed estimation model is, by design, simple. It does not require many inputs and it does not rely on too many assumptions. As a result, the proposed approach should be applicable and easy to use.

Two limitations of the approach are identified. The first limitation concerns the level of precision of the input for quality level. Currently, the quality rating that is used to base the estimation is the rounded star rating—that is, 1-star to 5-star. This approach may lead to less accurate estimates compared to the approach in which the number behind the commas is taken into account (e.g., 1.5-star). The second limitation is related to the use of Refactoring Adjustment (RA). This adjustment relies solely on expert judgment of evaluators based on the context of a particular system, which might result in different numbers for different evaluators. Nevertheless, one solution to address such limitation is by introducing clear assessment criteria to determine RA. This way, relevant contexts of a system or project can be assessed quantitatively to determine the discount value for refactoring adjustment.

The contribution of this paper for software engineering research is as follows. First, this paper should raise the awareness of software engineering researchers, particularly those in the field of software engineering economics, about the importance of measuring and estimating the cost of quality of software systems. Second, this paper proposes a practical approach to estimate the cost of achieving higher quality software that is transparent and repeatable, and thus can be further validated by other researchers.

For the practice of software engineering, the approach for estimating the cost of improving software quality and the amount of savings obtained in return can be used as a basis to take informed decisions related to quality improvement initiatives. The approach provides valuable information such as cost and benefit analysis of a repair investment, which is very useful for both software engineering practitioners and project executives to take the right decision in managing their software portfolios.

5. RELATED WORK

To the best of our knowledge little work has been done to define and measure technical debt in software systems. In this section, the discussion on related work is focused on existing approaches for quantifying technical debt.

Chin, Huddleston, and Gat defined technical debt as the cost of holding onto debt during active development and maintenance [7].

$$TD_{development} = \sum_{n=0}^{a-1} RI * (1 + CI)^n$$

$$TD_{maintenance} = RI * (1 + CI)^a * m$$

RI (Recurring Interest) represents the cost for holding onto a debt, which is essentially the amount of money spent on maintenance. CI (Compounding Interest) represents additional debt that accrues over time due to poor technical quality. Finally, a and m represent the number of years of active development and maintenance respectively. The total technical debt is defined as $TD_{development}$ plus $TD_{maintenance}$.

Notice that Chin et al., defined the interest of technical debt as the cost of maintenance. Such definition implies that regardless of the quality level, systems always have to pay interest. Obviously, this is not true because systems with an ideal quality level will have zero debt, and therefore should pay no interest.

The work of Letouzey and Coq [16] defined a software quality model called SQALE (Software Quality Assessment based on Lifecycle Expectations), which was based on the ISO 9126 standard. Quality characteristics incorporated in the quality model include testability, changeability, and reliability. The SQALE method defined the so called remedi-

ation indices. These indices represent the amount of effort that is required to resolve non-conformities from the generic requirements of the quality sub-characteristics. Aggregation of remediation effort to a higher level (e.g., quality characteristics or system level) was done by summing the lower level indices. This remediation effort can then be translated into financial value, which represents the amount of technical debt in a system. One limitation of the SQALE method is that there is no clear justification for the remediation indices. Ideally, the remediation indices should be based on empirical data. Furthermore, the SQALE method only provides estimation of debt, but not interest.

Another way of quantifying technical debt is presented in a CAST report [5]. The proposed approach looked at the density of coding violations with regard to issues of security, performance, robustness, and changeability of the code. Coding violations were classified into high, medium, and low violations. Furthermore, it is assumed that only 50%, 25%, and 10% of the high, medium, and low violations respectively are actually being fixed. With the assumed fixing effort of 1 hour for each violation, technical debt can then be calculated as follows: (10% of low severity violations + 25% of medium severity violations + 50% of high severity violations) * number of hours to fix * cost per hour fix. Based on assessments of 288 systems, the report suggested that the evaluated systems, on average, have roughly one million dollars technical debt. The report did not provide enough information about the code measurement, which makes it difficult to further assess the validity of the method. Nevertheless, the reported finding that stated Cobol systems have the lowest technical debt compared to other technologies is counter intuitive.

6. CONCLUSIONS AND FUTURE WORK

In this paper, the notions of technical debt and interest have been defined and put into the perspective of IT management. Furthermore, an approach to estimate the amount of technical debt and its interest has been defined.

The method to estimate technical debt comprises of two parts: 1) estimation of repair effort; 2) estimation of maintenance effort. SIG's quality model for software maintainability is used to assess the quality of software systems. The quality ratings resulted from the assessment are used for estimating technical debt. Based on empirical data of 44 systems, the fraction of rework to improve software quality is estimated. This estimation is based on changes in quality ratings of system snapshots that are calculated using the quality model. This estimated rework fraction is a fundamental part for determining the cost of repair to achieve an ideal quality level, which represents the technical debt.

Estimation of maintenance effort gives information about extra cost spent on maintenance due to poor technical quality. Additionally, by calculating the saving in maintenance effort obtained by improving software quality, the return on investment (ROI) of a repair work can be estimated. Overall, the proposed approach to estimate technical debt provides a clear picture about the cost of repair, its benefits, and the expected payback period.

Further work will be focused on refining the approach based on feedback obtained from its application in ongoing projects. Additionally, formal validation of the model using empirical data is needed. Such validation can be performed to assess the accuracy of the cost estimation model.

7. REFERENCES

[1] T. Alves, C. Ypma, and J. Visser. Deriving metric thresholds from benchmark data. In *26th Int. Conf. on Software Maintenance*. IEEE, 2010.

[2] R. Baggen, K. Schill, and J. Visser. Standardized code quality benchmarking for improving software maintainability. In *Proc. of the 4th International Workshop on Software Quality and Maintainability (SQM'10)*, 2010.

[3] D. Bijlsma. Indicators of issue handling efficiency. Master's thesis, University of Amsterdam, 2010.

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

[5] CAST. CAST worldwide apllication software quality study: Summary of key findings, 2010.

[6] S. Chidamber, D. Darcy, and C. Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. *Software Engineering, IEEE Transactions on*, 24(8):629–639, 2002.

[7] S. Chin, E. Huddleston, W. Bodwell, and I. Gat. The economics of technical debt. *Cutter IT Journal*, 23(10):11–15, 2010.

[8] W. Cunningham. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1993.

[9] B. e Abreu and W. Melo. Evaluating the impact of object-oriented design on software quality. In *Software Metrics Symposium, 1996., Proceedings of the 3rd International*, pages 90–99. IEEE, 2002.

[10] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *Software Engineering, IEEE Transactions on*, 27(1):1–12, 2002.

[11] M. Fowler. Technical debt quadrant, 2009.

[12] R. Harrison, S. Counsell, and R. Nithi. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *Journal of Systems and Software*, 52(2-3):173–179, 2000.

[13] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pages 30–39. IEEE, 2007.

[14] International Organization for Standardization. ISO/IEC 9126-1: Software engineering - product quality - part 1: Quality model, 2001.

[15] C. Jones. Backfiring: Converting lines of code to function points. *Computer*, 28(11):87–88, 2002.

[16] J. Letouzey and T. Coq. The SQALE Analysis Model An analysis model compliant with the representation condition for assessing the Quality of Software Source Code. *VALID*, 2010.

[17] S. McConnell. 10x software development.

[18] Software Productivity Research, LLC. *SPR Programming Languages Table Ver. PLT2007c*, December 2007.