# Estimating the Breaking Point for Technical Debt

Alexander Chatzigeorgiou[1], Apostolos Ampatzoglou[2], Areti Ampatzoglou[2], Theodoros Amanatidis[1]

[1] Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece
[2] Department of Mathematics and Computer Science, University of Groningen, Netherlands
achat@uom.gr, a.ampatzoglou@rug.nl, areti.ampatzoglou@rug.nl, tamanatidis@uom.edu.gr

***Abstract*—In classic economics, when borrowing an amount of money that causes a debt to the issuer, it is not usual to have interest which can become larger than the principal. In the context of technical debt however, accumulated debt in the form of interest can in some cases quickly sum up to an amount that at some point, becomes larger than the effort required to repay the initial amount of technical debt. In this paper we propose an approach for estimating this breaking point. Anticipating how late the breaking point is expected to come can support decision making with respect to investments on improving quality. The approach is based on a search-based optimization tool that is capable of identifying the distance of an actual object-oriented design to the corresponding optimum one.**

***Keywords—technical debt; search-based software engineering***

## I. INTRODUCTION

In classic quality management Harrington popularized the concept of "*cost of poor quality*" or "*poor quality costs*" [1], referring to costs generated as a result of producing defective material. Poor Quality Costs include not only the costs involved in filling the gap between the actual and optimum product quality, but also the effort required to rectify any defects in delivered products. In recent years, the software engineering community has embraced a similar concept, namely technical debt, to communicate the aforementioned costs in software development. Technical Debt (TD) is a collective term for a number of imperfections and deficiencies in software, which can lead to increased maintenance effort [2]. The term itself was coined by Cunningham [3] who drew an analogy between technical and economic debt, in the sense that speeding up development by providing not-quite-right code, counts as a debt that will have to be paid back in the form of harder future maintenance.

Research on TD and its management has become intense in the last five years, reflecting the interest of the community to address issues related to quality management from the perspective of their associated costs and benefits [4]. An overview of the research on Technical Debt Management (TDM) can be found in two recent literature reviews [5], [6]. Despite the research volume on the subject during the last years, TDM state of research and practice still exhibits major challenges, such as: (a) the quantification of the underlying TD amount (i.e., in the valuation of interest and principal [6]), and (b) the strategies that can be used for managing the increasing amount of TD (because of the accumulation of interest) [7]. The focus of this study will be on both these challenges. Although the quantification and management of debt/interest has been extensively studied by the TD community, in most of the cases the proposed solutions are at a theoretical level (see [5], [6], and [7]).

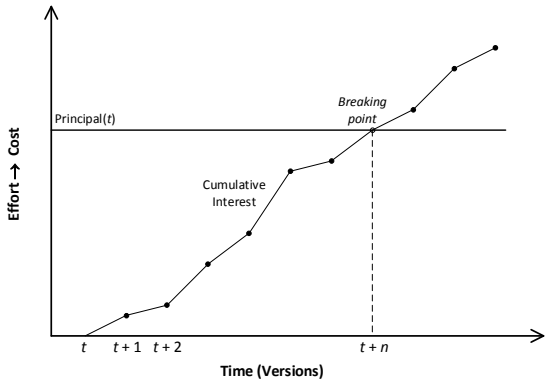
In particular, we focus on a theoretical framework named *FItTeD* (Framework for managing Interest in Technical Debt), proposed by Ampatzoglou et al. [7]. As captured by FItTeD, the accumulated interest can potentially sum up to an amount that becomes larger than the effort that was required for repaying TD at earlier stages of development (principal). Under this perspective, project managers should be able to calculate the time point at which the accumulated TD interest will reach the amount of the corresponding principal. This time point is called 'breaking point' in the sense that at this time, any savings resulting from the decision to not repay TD will vanish due to increased maintenance effort during evolution [7]. Another interesting instance in the growth trajectory of software (inflection point) has been defined as the time at which maintenance effort exceeds effort of implementing new functionality [8].

In this paper we extend previous work [7] by instantiating FItTeD based on a search-based software engineering methodology and implementing a supporting tool. The breaking point is obtained as the ratio of principal over the average interest per version. In particular, for any actual system it is possible to extract a set of refactoring suggestions that can be applied, so as to increase quality. The application of these refactorings will yield an optimum design as captured by some fitness function. The distance between the actual and the optimum system serves as an indicator of the TD principal. The ratio of quality of the optimum over the actual system is used to estimate the ratio of increased effort over the original effort (interest) to maintain the two systems. The approach is presented in Section III and illustrated on an OSS system in Section IV.

## II. GOAL

The goal of the proposed approach is to assist software project managers in their decision making, by providing an estimate of the time point at which accumulated interest from TD will exceed the initial savings obtained by not repaying the principal. The value of this information will be illustrated with the help of Figure 1. Let *Principal*(*t*) be the amount of effort saved by not repaying the existing TD at time point *t*, or in other words, the cost that has not been spent on refactoring an existing software system to reach the optimum design-time quality level [6].

Since the decision on whether repayment of TD should be performed has to be taken at (any) particular time point, we represent this amount as a straight horizontal value. Although principal will probably increase over time since software quality usually decays, at the time point *t* it can only be viewed as a constant value. Assuming that TD has not been repaid at time point *t*, future maintenance tasks will demand additional effort (compared to the effort that would be spent, had the TD been repaid). This additional effort is known as the interest, and for evolving systems it will accumulate as depicted in Figure 1.




* In this illustration we assume that no further repayment is performed during evolution

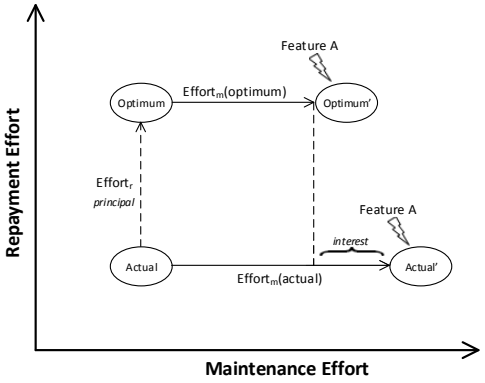
Fig. 1. Breaking point at which cumulative interest exceeds principal

The approach allows software project managers to estimate in how much time the decision to not repay TD (at point *t*), will incur a cumulative interest that becomes larger than the principal. In case this time point is too distant in the future (*n* versions after today's time point *t*), then probably, it makes little sense to resolve TD at time point *t*. In case this breaking point is expected to come in the near future and especially when it lies within the foreseeable product evolution, the manager should consider the benefit of addressing TD liabilities.

## III. Approach

### *III.A Estimation of Principal*

In general, as Curtis et al. [9] pointed out, TD-Principal is calculated as a function of three variables - the number of must-fix problems, the time required to fix each problem, and the cost for fixing a problem. Concerning the first variable, we assume that any given object-oriented software system, has an actual design quality which can be assessed by an appropriate fitness function. In our case, the selected fitness function is a measure of both coupling and cohesion. This function assesses how well the entities in a system (methods and attributes) have been allocated to the classes [10]. Local search optimization algorithms can then be applied to derive the optimum design, i.e. the design consisting of the same constituent entities that optimizes the selected fitness function. Thus, the 'distance' between the actual and the optimal design, in terms of their difference in the fitness function value, can be considered as the principal that has to be paid to transform the actual system to the corresponding optimum one. Obviously, relying on particular metrics to assess TD poses a construct validity threat. However, one can employ any kind of fitness function, even aggregate ones, to capture the desired aspects of quality. To ensure the applicability of this concept, we have developed a tool that is able to assess the number and type of required refactorings to achieve the conversion to the optimal design (see Section III.C). This notion of distance is illustrated in Figure 2.




* In this illustration we assume that while adding features quality does not further decay

Fig. 2. Increased maintenance effort for technical debt item

The relevant effort to fill the gap between the actual and the optimum design is effort devoted to the *repayment* of TD and therefore it is denoted as $Effort_r$. This repayment activity consists in the application of refactorings aiming at the improvement of quality from the perspective of the employed fitness function. Taking into account past data concerning the number of refactorings that can be performed at the unit of time and the developer cost per hour it becomes possible to obtain an estimate of the principal in terms of money (see Section IV).

### *III.B Estimation of Interest*

The effort related to the enhancement of functionality or the correction of bugs is devoted to system *maintenance* and is denoted as $Effort_m$. Under usual conditions, the effort required to maintain the actual system (e.g. to add a new feature A) will be larger than the corresponding effort to add the same feature in the optimum system (Figure 2). The difference between the two efforts corresponds to the interest that is accumulated during this transition from version *t* (prior to the addition of feature A) to version *t*+1. Due to the nature of software evolution it is not possible to anticipate what kind of changes will be performed in future versions of a software product. To address this limitation and to keep the model simple, we consider past changes in the history of the system. Any kind of measure for the quantification of past effort can be used. Here, we extract the average number of added lines of code between successive versions. Added lines reflect the effort for the introduction of new functionality and to some extent the effort for the modification of existing modules. Let us assume that on average *k* lines of code are added between any two successive versions.

It is reasonable to assume that performing *k* additions on a system with a superior design quality (expressed by the value of the employed fitness function) will be easier than performing the same changes on a system with lower quality. For the sake of generality we assume that this maintenance effort will be proportional to the design quality:

$$Effort_m = c \cdot FitnessValue \quad (1)$$

where *c* is an arbitrary constant.

According to the proposed approach we know the fitness value for the actual and the optimum systems, and thus we can take the ratio of the theoretical effort (if maintenance been performed on the optimum system) over the actual effort:

$$\frac{Effort_m(optimum)}{Effort_m(actual)} = \frac{c \cdot FitnessValue(optimum)}{c \cdot FitnessValue(actual)} \Rightarrow$$
$$Effort_m(optimum) = \frac{FitnessValue(optimum)}{FitnessValue(actual)} \cdot Effort_m(actual) \quad (2)$$

Therefore, if the actual maintenance effort is obtained as the average of past maintenance effort (as the number of added lines $k$), then the optimum effort can be directly deduced. As a result, the interest that is accumulated between any two successive versions can be obtained as the difference between the actual and the optimum effort and is equal to:

$$Interest = \Delta Effort = k \cdot \left(1 - \frac{FitnessValue(optimum)}{FitnessValue(actual)}\right) \quad (3)$$

Once again, taking into account past data concerning the number of added lines of code that can be achieved at the unit of time and the developer cost per hour, it becomes possible to obtain an estimate of the interest, in terms of money. Having the principal and the interest that is accumulated in one version, it is straightforward to obtain the number of versions after which the breaking point will be reached as:

$$versions = \frac{Principal(\$)}{Interest(\$)} \quad (4)$$

It should be noted, that beyond the assumed linear nature of interest increase in this simplified model, other economic parameters, such as inflation are not considered, but can be easily integrated into the proposed analysis. Moreover, a deterministic prediction of future effort is an ambitious task, because numerous factors, e.g., business requirements and technology changes come into play. It would be interesting to consider a stochastic approach where such factors are handled as probabilities.

### *III.C Tool Support*

In the proposed approach it is assumed that software quality can be assessed by means of a fitness function capturing coupling and cohesion. Obviously, quality has many facets and thus it is questionable whether it can be quantified in terms of a single function that expresses all aspects of quality [2]. Nevertheless, the quality assurance team can examine TD from various perspectives by employing appropriate measures or even extract a polynomial of various quality attributes (as in the case of models such as QMOOD [11]). In our study, quality is captured by the *Entity Placement* metric that simultaneously quantifies coupling and cohesion in the same terms [10]. The metric is based on the 'distance' that each system entity has from its own class and from other system classes. In a well-designed system, entities should have a low distance from the class in which they reside and a large distance from all other classes.

An object-oriented system can be considered as a set of entities (methods and attributes) which can be freely moved between classes. By treating the challenge of finding the placement of entities that minimizes the Entity Placement metric as a search-space exploration problem [12], we have developed a software tool, JCaliper, which is capable of extracting the optimum allocation of entities to classes. JCaliper employs search-based techniques such as Hill Climbing, Tabu Search and Simulated Annealing [13] to find the optimum design. To transform an existing system to its corresponding one, a number of refactoring operations should be applied such as Move Method, Move Attribute, Extract Class, Pull-Up Method etc. [14]. JCaliper automatically extracts the number, type and sequence of refactorings required to obtain the optimum design. Therefore, the 'distance' between the actual and the optimum design in terms of their Entity Placement value can be substantiated as a series of actual refactorings to be performed. This number of refactorings is a concrete computation of the amount of TD, from the perspective of the selected quality measure.

## IV. CASE STUDY

JUnit[1] is an open-source framework to write repeatable test cases, written in Java. To illustrate the proposed approach we have employed version 4.10 consisting of 147 classes and 786 entities. The Entity Placement for the actual and the optimum system as derived by JCaliper is shown in Table I. The distance in the Entity Placement, the percentage improvement that can be achieved by paying off the relevant TD and the number of required refactorings to obtain the optimum system are also shown. To provide an estimate of the required effort and cost for repaying the principal, we consider that a refactoring takes on average five to ten minutes to perform [15]. According to the US Bureau of labor statistics report for 2014, the mean hourly wage for software developers/programmers is $45.81 [16], leading to a rough estimate of 5.73$ per refactoring.

TABLE I. ESTIMATE OF PRINCIPAL (JUNIT)

| **Entity Placement** | | | | | **Effort** | |
|---|---|---|---|---|---|---|
| **actual** | **optimum** | **distance** | **Improvement** | **#Refactorings** | **time** | **Cost** |
| 0.819[a] | 0.692 | 0.127 | 15.5% | 330 | 41.25h | 1,891$ |

[a] In contrast to what is shown in Fig. 2, for Entity Placement, a lower value corresponds to a better design

To obtain an estimate of the interest that is accumulated between versions, we performed a software evolution analysis of 16 versions of JUnit with the help of the SEAgle platform [17]. SEAgle provides an overview of the evolution of repository activity and source code metrics, which indicate that the average number of added lines of code between any two successive versions is 848 (we note that the total number of added lines of code is far from evenly distributed among versions).

It is rather risky to associate a number of LOC with a particular effort, especially for an open-source project. Nevertheless, for the sake of completeness and since this example serves only illustration purposes we assume a productivity for Java, approximately equal to 25 LOC/hour (including designing, writing and testing the corresponding code) [18], a figure which is also the upper bound in the productivity found in a case study on extreme programming in practical settings [19].

Based on this productivity estimate, the required effort to perform maintenance between any two successive versions and the associated cost is summarized in Table II. The cost in dollars is estimated using the same hourly wage as in the case of

[1] http://junit.org/

refactoring effort (although it would be reasonable to assume that maintainers have a different salary). Under the assumptions stated earlier, the lower effort for maintaining the optimum JUnit 'edition' is obtained by considering the ratio of the optimum system quality over that of the actual system (eq. (2)).

TABLE II. ESTIMATE OF INTEREST PER VERSION (JUNIT)

| | Actual Effort | | Optimum Effort | |
|---|---|---|---|---|
| **average added LOC** | **time** | **Cost** | **time** | **Cost** |
| 848 | 33.9h | 1,553$ | 28.6h | 1,310$ |

The difference between the effort required to maintain the actual and the optimum version corresponds to the debt's interest and is equal to (interest can also be directly obtained from (3)):

$$Interest = Effort(actual) - Effort(optimum) = 243\$$$

By setting the principal and the interest accumulated during each transition into eq. (4) we obtain the number of versions that will elapse until the evolution reaches the breaking point, or in other words the time point at which the accumulated interest will reach the original principal:

*Elapsed time to breaking point* = 7.78 versions

To consider whether this breaking point is too distant in time or not, we have analyzed the 9 latest versions of JUnit and found that a new version is released every 8.4 months on average. In other words, for this project, the breaking point will occur approximately 5.4 years from the time of analysis. To provide another view, we can estimate the annual interest that has to be paid, which is equal to 347$. Considering the principal as the amount that has been 'borrowed' (by not repaying technical debt), this figure translates to an annual interest rate of 18.35%. If the aforementioned estimates are accurate, it is more than evident that this form of loan is extremely expensive. The analysis can also be carried out for partial repayment of the principal, a scenario which might be more realistic, given that not all parts of a software design are subject to frequent maintenance. In that case, project managers can select a portion of the initially suggested refactorings to estimate the principal, and also adapt the ratio of quality between the optimum and the actual system to obtain a more appropriate breaking point.

## V. LIMITATIONS

Since the approach depends upon a number of assumptions, we summarize here the most important issues to be considered:

- TD has other dimensions beyond coupling and cohesion
- Maintenance effort should account for other activities beyond addition of LoC (e.g. modifications, deletions)
- Future maintenance effort cannot be predicted solely on the basis of past maintenance tasks

Moreover, it should be noted that cost estimates for refactoring application and introduction of new code, do not reflect commonly agreed figures, and thus are also subject to change.

## VI. CONCLUSIONS

In this paper we have introduced an approach for estimating the time (breaking point) at which the accumulated interest of technical debt in object-oriented systems will exceed the corresponding principal. The approach is based on the extraction of the optimum design for a particular system, relying on the optimization of an appropriate fitness function that serves as an indicator of quality. In the context of Technical Debt Management, the knowledge of this breaking point can provide support to the decision making process of software project managers. Estimating when any savings, obtained by not investing on technical debt repayment, will be vanished, can guide decisions to select one particular form of investment over the other.

## REFERENCES


[1] H. J. Harrington, *Poor-Quality Cost: Implementing, Understanding, and Using the Cost of Poor Quality: 11*. CRC Press, 1987.

[2] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," *IEEE Softw.*, vol. 29, no. 6, pp. 18–21, Nov. 2012.

[3] W. Cunningham, "The WyCash Portfolio Management System," in *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications*, NY, USA, 1992, pp. 29–30.

[4] E. Lim, N. Taksande, and C. Seaman, "A Balancing Act: What Software Practitioners Have to Say about Technical Debt," *IEEE Softw.*, vol. 29, no. 6, pp. 22–27, Nov. 2012.

[5] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *J. Syst. Softw.*, vol. 101, pp. 193–220, Mar. 2015.

[6] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "The financial aspect of managing technical debt: A systematic literature review," *Inf. Softw. Technol.*, vol. 64, pp. 52–73, Aug. 2015.

[7] A. Ampatzoglou, A. Ampatzoglou, P. Avgeriou, and A. Chatzigeorgiou, "Establishing a framework for managing interest in technical debt," *5th International Symposium on Business Modeling and Software Design* (BMSD), Milan, Italy, 2015.

[8] N. Ramasubbu and C. F. Kemerer, "Managing Technical Debt in Enterprise Software Packages" *IEEE Trans. Softw. Eng.*, vol. 40, no. 8, pp. 758-772, August 2014.

[9] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the size, cost, and types of Technical Debt," *3rd International Workshop on Managing Technical Debt (MTD '12)*, Zurich, Switzerland, 2012, pp. 49 - 53

[10] N. Tsantalis and A. Chatzigeorgiou, "Identification of Move Method Refactoring Opportunities," *IEEE Trans. Softw. Eng.*, vol. 35, no. 3, pp. 347–367, May 2009.

[11] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, Jan. 2002.

[12] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based Software Engineering: Trends, Techniques and Applications," *ACM Comput Surv*, vol. 45, no. 1, pp. 1–61, Dec. 2012.

[13] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Harlow: Pearson Education Limited, 2013.

[14] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, 1 edition. Reading, MA: Addison-Wesley Professional, 1999.

[15] W. C. Wake, *Refactoring Workbook*, 1 edition. Boston: Addison-Wesley Professional, 2003.

[16] U.S. Bureau of Labor Statistics, "National Occupational Employment and Wage Estimates," United States, Mar. 2015.

[17] T. Chaikalis, E. Ligu, G. Melas, and A. Chatzigeorgiou, "SEAgle: Effortless Software Evolution Analysis," *30th IEEE Int. Conference on Software Maintenance and Evolution (ICSME)*, 2014, pp. 581–584.

[18] L. Prechelt, "An Empirical Comparison of Seven Programming Languages," *Computer*, vol. 33, no. 10, pp. 23–29, Oct. 2000.

[19] P. Abrahamsson and J. Koskela, "Extreme programming: a survey of empirical data from a controlled case study," *Int. Symposium on Empirical Software Engineering (ISESE), 2004,* pp. 73–82.