From Lasagna to Spaghetti, A Decision Model to Manage Defect Debt

Abdullah Aldaeej
University of Maryland Baltimore County
Department of Information Systems
aldaeej1@umbc.edu

Carolyn Seaman
University of Maryland Baltimore County
Department of Information Systems
cseaman@umbc.edu

ABSTRACT

In this paper, we propose a model that formalizes the role of software evolution in characterizing Technical Debt (TD) by defining a series of software product states, where each successive state represents an increased level of maintenance code churn, and thus presumably an increased level of change difficulty. We also propose a way to use these states to estimate TD principal and interest and use this information in decision making during release planning. In addition, we illustrate our model using bug report data from the Eclipse-Birt project.

CCS CONCEPTS

• **Software and its engineering → Software evolution**;
• **Information systems →** Decision support systems

KEYWORDS

Technical debt management, Defect debt, Software evolution, Release planning, Decision making

1 INTRODUCTION

Technical Debt (TD) is a metaphor that is used to help communicate software refactoring needs to non-technical management and other stakeholders [1]. It is defined as the trade-off between the short-term benefit and the long-term impact of delaying software maintenance tasks [2]. By some definitions, its scope is limited to liabilities concerning internal system quality, primarily maintainability and evolvability, in design and implementation constructs [3]. In modern software projects, it is common that TD is incurred to achieve short-term benefits such as time to market, and reduce maintenance cost. However, the incurred debt will have future impact on the project. Therefore, it is important to manage TD throughout the project.

One of the fundamental components of TD management is measuring and monitoring the debt [4]. TD is measured by estimating two primary characteristics, principal and interest. Previous work estimates TD principal and interest based on different aspects, such as software quality deficiency [5,6,7], project maintenance data such as issue tracking data and change data [6,7,8,9], incorporating expert judgment [10,11], and tracking developer activities from log sessions [12].

Despite the variety of approaches to characterize and quantify TD principal and interest, none so far explicitly consider the role of software evolution while performing such estimation. According to Lehman's Laws on software evolution, software undergoes continuous changes that lead to increased complexity of the software unless refactoring is done to reduce it [13]. In this paper, we propose a model that estimates defect debt principal and interest based on the evolution of the entire software product. The proposed model is based on the idea that preventive maintenance (i.e. TD repayment) is likely to be more expensive in the future than it would be currently because the code base continually degrades and becomes harder to work with over time, due to the normal effects of evolution. We adopt the mechanism in [14] which represents a software product in a finite number of states that reflect the level of deterioration of the product due to maintenance. The main contribution of this paper is to suggest and demonstrate a new approach to estimating TD principal and interest during release planning.

2 RELATED WORK

Several approaches have been proposed in the technical debt literature for measuring TD principal. One of the common approaches is estimating TD principal based on quality assessment of software source code [5,6,7]. In this approach, the underlying software is analyzed using some static analysis tools to determine the quality level of the software. TD principal is interpreted as the amount of effort needed to improve software quality to the ideal level. Another approach estimates TD principal based on the project maintenance data such as issue tracking data or change data. TD principal is measured as the effort to fix defects using statistical models [8]. In addition, TD principal can be estimated based on expert knowledge [10,11]. In this approach, experts utilize their experience to provide rough estimation of the principal.

Measuring TD interest is more complex than TD principal. This complexity stems from the uncertainty about the future impact of debt. Previous studies estimate TD interest using different approaches. It can be estimated directly from experts who evaluate the impact of TD in the business [5,15], or by mining software repositories [9], or using log sessions to track developer activities and identify code comprehension metrics [11].

3 THE PROPOSED MODEL

We propose a model that estimates defect debt principal and interest based on the evolution of the software product. We focus on defect debt to simplify our analysis, although future work will extend this approach to other types of TD. The estimation is performed during release planning to facilitate monitoring the long-term impact of defect debt, and hence aid decisions about paying off defect debt. The model is illustrated using an example based on bug report data from the Birt project. The example is intended to be exploratory in nature, and does not include a formal evaluation of the model.

The Birt dataset is available in PROMISE Repository [1]; thanks to researchers who uploaded the data [16]. This dataset includes 4178 fixed bugs within time range between 2005 and 2013. In addition to the bug data, we also made use of additional attributes that allowed us to identify debt, measure the degree of maintenance and software state, and measure the maintenance effort. In this example, we use code churn added and modified as a surrogate for maintenance effort.

3.1 Software State

The proposed model is motivated by the idea in [14] that models the evolution of software in a finite number of states S = {0, 1, 2, …, Z} which represents a sequential level of deterioration. The estimation of defect debt principal and interest is performed based on the current software state. The model assumes that software is initially delivered with an optimal quality at state 0 [14]; the implication of this assumption is discussed in section 4. We call this state a lasagna state, a metaphor we use to describe a well-designed software product. Software deteriorates over time based on the amount of maintenance it has undergone. The state Z represents the worst state in which the software must be refactored. We call this state a spaghetti state.

In our model, the software state is identified by the degree of maintenance (DM) which is the percentage of the initial software size that has been touched due to defect fixing. DM is used as a measure for software deterioration by accumulating code churn of defect fixing activities, and dividing them by the initial software size. When a software product is refactored, it starts a new deterioration cycle where software state sets to 0, the software size after the refactor will be the new initial size, and DM is reset. The model assumes that the software state implies the level of difficulty to work with code. As the software state increases, the software code will be more difficult to comprehend. Hence, the effort to make changes will increase accordingly.

Table 1 illustrates the representation of the software state which is adopted from [14]. Each state has a maintenance degree threshold which triggers the transition to the next state. In state 0, the threshold is 10%; and as the software state increases, the DM range is incremented by five. DM threshold for state 'i' is determined as (10 + 5 * i) [14], where DM threshold accumulates over software states. The rationale behind this increment is that as the software deteriorates over time, we expect to receive more defects. As the software state moves toward the spaghetti state, the frequency of defect fix and the amount of code churn are expected to be larger. In addition, the actual size of the software is expected to be larger than the initial software size, while we consider the latter for computing DM. Thus, the threshold increment is performed to adjust for the increase in the software size and the code churn.

Table1: Software state definition

Software State	Degree of Maintenance (DM)
0 (lasagna)	< 10%
1	>= 10% AND < 25%
2	>= 25% AND < 45%
3	>=45% AND < 70%
…	…
Z (spaghetti)	

3.2 Data Preparation

The spaghetti state (state Z) varies between different software products. To determine the spaghetti state, we first observe the software evolution by studying historical data on all releases that have been implemented on the software. Then, we identify the major release that has the largest code churn. We assume that this release is a refactoring release, and all debt has been cleared out at this point. The spaghetti state will be determined as the software state when the major release occurred.

In the Birt project, we identified 30 releases within the time-range of the dataset; which are archived in the Birt project website [2]. We further collected the code churn for each release, considering all commits tagged to each release. The largest code churns were found in the last three releases –releases 28 to 30. However, we need at least five releases after the major release to exercise our model. So, we identified the largest code churn prior to release 28. This was release 13, and hence it was selected as the major release upon which to define the spaghetti state.

Once the major release is identified, we divide the dataset into two parts. Fig. 1 shows the high-level organization of the Birt project data after identifying the major release. The horizontal line represents the timeline whereas the vertical lines refer to releases.

[1] http://openscience.us/repo/defect/bug-reports/birt.html

[2] https://www.eclipse.org/birt/

The first part is the historical data concerning with data before the major release. This part is used to calibrate the model. The initial software size in the historical data is the size of the first release. The second part contains data after the major release. It is used to apply the model and simulate decisions at each Release Planning (RP). After the major release, the software begins a new deterioration cycle starting from state 0. The total code churn and DM are set to zero, and the initial software size is the size of the major release.

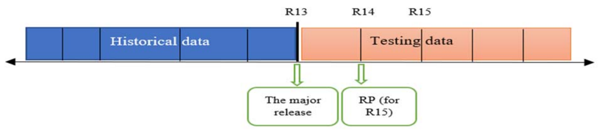

Figure 1: An Overview of the Birt project data

3.3 Historical data

The historical data is used to determine the finite number of software states, distribute defect data based on software state, and measure the effort to fix defect debt per state. First, the historical bug data is sorted in chronological order based on commit time. Then, DM and software state are measured at the bug report instance level in order to distribute the defect data over software states. In addition, we measure DM and software state at the release level to generate the transition probability matrix since the model performs the estimation at the release time. Table 2 shows DM and software state at the release level in the Birt project. When the major release (release 13) is implemented, DM is 71.5%. Based on the representation scheme of software state, the major release implemented when the software state is 4. This means that state 4 is the spaghetti state.

Table 2: DM and software state at each release

Release	DM	State
1	10.26%	1
2	10.70%	1
3	22.14%	1
4	22.99%	1
5	25.50%	2
6	29.20%	2
7	45.90%	3
8	46.70%	3
9	51.50%	3
10	52.30%	3
11	62.90%	3
12	69.10%	3
13	71.50%	4

In addition, we apply a Markov chain model on the sequence of states in Table 2, using *markovchainFit* function [17], to generate the transition probability matrix presented below. We exclude state 4 from our analysis because the last state does not have sufficient data like all the prior states. The major release can be implemented at any time during last state.

$$P = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0.75 & 0.25 & 0 \\ 0 & 0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix} \end{array} \qquad (1)$$

This matrix is a single-step probability matrix that includes the probability from state i (row) to state j (column) in the next release (one release ahead). For instance, when the software state is 1 at release 1, the probability that in the next release (release 2) the state remains at 1 is 0.75, or moves to state 2 is 0.25. The matrix is upper triangular since there are no backward transitions [14].

Finally, we measure the average effort to fix defect debt per state. First, defect debt is identified based on the mechanism in [8]. A defect is considered as debt if it is not fixed within the same release where it is reported. For example, a bug reported in Release (R= i) and fixed in Release (R > i) will be consider a debt bug. Table 3 presents the descriptive statistics of debt bugs fixed in each software state in the Birt project. It shows the frequency of debt bugs and the descriptive statistic of code churn related to the debt bug per software state. According to the statistics, there is no debt at state 0 which means that only regular bugs cause the first 10% degree of maintenance. Also, the code churn in each state is highly deviated from its mean, and there are some extreme outliers. These outliers unrealistically increase the mean value. Therefore, using the mean in this case will be misleading. Instead, we use the median to measure the effort per state since it is more robust against outliers [18].

Since the software state in our model implies the level of complexity to work with code, we assume that the Average Effort (AE) to fix debt per state (i) increases monotonically with the state. To apply this assumption, we multiply the median effort to fix debt per state by the state number for all state i > 0. Equation (2) below illustrates how AE is calculated for each state i. The result of this equation is presented in the last row in Table 3. Since the model uses code churn as a surrogate for maintenance effort, the result of equation (2) means that the higher the state the more code churn entailed to fix debt. This relative measure (AE) is not meant to be an accurate estimate of raw effort, but as a meaningful measure to compare with other values calculated in the same way.

$$\begin{aligned} AE(i) &= median(i) \times i && for\ i > 0 \\ AE\,(0) &= median\,(0) && for\ i = 0 \end{aligned} \qquad (2)$$

3.4 Testing data

We used the testing data to apply the model and simulate decisions at each Release Planning (RP) as if the model had been used. We assume that release (i+1) is planned at the time that release (i) is issued, and so decision makers have full knowledge of what was done during release i. After the major release, the software product begins a new deterioration cycle starting from state 0. Every release issued after the major release is simulated as an RP for the next release.

Table 3: Statistics of debt per software state in Birt project

		Software State			
		0	**1**	**2**	**3**
# of debt bug		0	110	87	240
Code Churn (effort)	Min	--	1	0	0
	1Q	--	10	10	7
	Median	--	**42.5**	**38**	**29**
	3Q	--	122	153	158
	Max	--	6029	9000	13890
	Mean	--	217	374.6	232.9
	SD	--	656.7	1163.9	950.9
	AE	**--**	**42.5**	**76**	**87**

At each RP, we use the testing data up to the RP time to determine the current software state, based on maintenance code churn since the major release. We identify the debt bugs and apply the model to estimate the defect debt principal and interest at the RP time. The debt principal is estimated one-time when the debt is identified at the RP. Based on the current software state, we leverage the Average Effort (AE) to fix debt per state from the historical data as well as the single-step transition probability matrix to generate the expected debt principal. The step in our model reflects how many releases ahead from the current RP time that we intend to estimate the debt principal for. We use the single-step to indicate the effort to fix the debt in the next release (the one being planned). The calculation of the debt principal is performed in this way to mimic the scenario where the debt is included to be fixed as part of the current RP. For example, at RP 1 in the Birt project, we identify 35 debt bugs (Table 4). The estimated principal for each debt is (42.5 * 0.75 + 76 * 0.25 + 87 * 0) = 50.87.

To estimate the debt interest, we first estimate the debt principal at two releases ahead, by using the double-step transition probability matrix. This indicates the effort to include the debt fixing as part of the next RP, not the current one, i.e. where the debt is not fixed in the current RP but carried to the next. The debt interest is calculated as the difference between debt principal and the principal at two releases ahead. Therefore, at each RP, we estimate two principals: the debt principal if the debt is fixed in the current RP, and the debt principal if the debt is fixed in the next RP. The difference between the two represents the amount of extra effort needed as a result of delaying the bug fixes one release.

In the Birt project, we applied the model at eight successive releases after the major release (after release 13). Each release is considered as an RP for the next release. Appendix A[3] demonstrates the process we performed to estimate debt principal and interest at each RP. First, we identify debt bugs as the bugs that are reported but not fixed at the RP time. After that, we calculate DM which indicates the current software state. Based on the current state, we leverage the effort to fix debt per state as well as the transition probability matrix from the historical data to estimate the debt principal and interest.

[3] https://swe.umbc.edu/~aldaeej1/Appendix%20(A).pdf

The exponent of the matrix is incremented in the case where the software state remains unchanged. This operation is used to adjust the transition probability after each forward step. It also used to avoid having the same principal and interest in the RPs where the software state is the same. In the Birt project example, the DM at the last RP (PR 8) is 9.6%. This means that the software remains at state 0 in all the eight simulated RPs. In the historical data however, the software state moves to 1 at the first RP (Table 2). Since there is no state 0 in our model that is calibrated based on the historical data, we start the testing data with state 1. So, the software state is 1 in all the eight RPs. In this situation, we continually increment the exponent of the matrix P after each RP.

The result of applying the model at eight successive RPs in the Birt project (following the process in Appendix A) is available in Table 4. The calculation details are available in Appendix B [4]. The model generates information in a sequential process, where the result table is expanded after each RP. We noticed that the debt principal is monotonically increased after each RP. However, it increased at slower rate at the later RPs. Since the interest is the difference between two principals at two successive RP times, the debt interest also increases more slowly at the later RPs. This indicates that the model behaves well intuitively based on the concept of software state. As the software state remains unchanged for several RPs, the difficulty to work with code will increase at a slower rate; so, the debt interest shrinks accordingly.

Table 4: The model results

RP_id	# of debt identified	Principal	Interest at RP 1	Interest at RP 2	Interest at RP 3	Interest at RP 4	Interest at RP 5	Interest at RP 6	Interest at RP 7	Interest at RP 8
1	35	50.87	7.66	14.13	19.28	23.13	26.43	28.8	30.62	31.98
2	31	58.53		6.47	11.62	15.47	18.77	21.14	22.96	24.32
3	23	65			5.15	9	12.3	14.67	16.49	17.85
4	18	70.15				3.9	7.15	9.52	11.34	12.7
5	54	74					3.3	5.67	7.49	8.85
6	33	77.3						2.37	4.2	5.55
7	10	79.67							1.82	3.18
8	2	81.49								1.36

The model results are used to support release planning decisions related to paying off defect debt. This decision can be made using the real option analysis [19] which is drawn from the economic theory to compare the short-term benefit value (saving the debt principal) and the long-term impact value (the debt interest). If the amount of debt interest is the same as the principal (the ratio of interest to principal is 1:1), then fixing the debt is a feasible option. For example, in the first RP, we identify 35 debt bugs. The principal of each of the 35 debt is 50.87; and their interest is 7.66. Thus, at the first RP, decision makers will be informed that if debt is incurred to the next RP, there will be an extra cost of 7.66. Since the ratio of interest to principal is very low (7.66: 50.87), then it is more feasible to incur the debt to the next RP. In different situations, this model also allows decision makers to view interest over many successive releases in the

[4] https://swe.umbc.edu/~aldaeej1/Appendix%20(B).pdf

future, not just one. If the 35 debt identified in RP1 incurred over many future RPs, the model will update the debt interest at the time of each future RP (i.e 14.13 in RP2, 19.28 in RP3, and so on).

4 LIMITATIONS

There are many limitations that affect the validity of our model. First, the model considers code churn related to only bug fixing. There are other non-bug fixing activities such as refactoring and new features that result in tremendous changes on the source code. In addition, the bug fixing activities are associated with corrective maintenance. This type of maintenance has lower code churn in comparison to the other maintenance activities [20]. Thus, the degree of maintenance in our model will increase relatively slowly, especially for large size software. However, we think that code churn related to bug fixing most appropriately leads to the deterioration of the software, and thus represents at least part of the effect of maintenance on the code base quality.

In addition, the model estimates debt principal and interest based on only the software state, which is in turn based on the amount of change the code base has undergone. Future work will expand the definition of "state", possibly drawing on other quality concerns that are related to technical debt, such as the number of code smells, or the levels of quality metrics.

The model assumes that the initial software is delivered without any quality issues. This assumption does not always hold especially in the context of open source software projects. In our example, the Birt project incorporates a great deal of maintenance activities at the beginning of the project. (exceeds 10% at the first release planning – Table 2). On the other hand, the project received fewer maintenance activities after the major release (release 13). We plan to experiment with different ways of splitting the data into historical and testing partitions. One approach could be to utilize testing data after a second major release instead of the first.

In the model, all defect debt identified in a certain release planning have the same fixing effort. The model does not distinguish between different instances of debt. No finer-grained debt estimation is considered. In the Birt project example, the 35 debt bugs identified in RP 1 have the same principal (50.87) and hence the same interest. In future work, we plan to address this limitation by adding weight to each debt item based on the severity level or other attributes.

Finally, the model does not account for the partial fix of debt. If project managers pay off part of the debt, the model will consider that as any other defect fixing activities, which further deteriorates the software. Future work will explore some techniques to move software state backward as the result of partial fix of debt.

5 CONCLUSIONS

We propose a new model to estimate and manage defect debt based on software evolution. The model estimates debt principal and interest at each release planning period based on the current software state. The purpose of the model is to provide an insight to software project managers while making decisions related to paying off defect debt during release planning. We illustrate how the model is applied in the Birt project. For future work, we plan to address the limitations of the model and expand it to incorporate other types of technical debt.

REFERENCES

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

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

[3] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)," *Dagstuhl Reports*, vol. 6, no. 4, pp. 110--138, 2016.

[4] C. Seaman and Y. Guo, "Measuring and Monitoring Technical Debt," in *Advances in Computers*, vol. 82, p. 22, 2011.

[5] J. L. Letouzey and M. Ilkiewicz, "Managing Technical Debt with the SQALE Method," *IEEE Software*, vol. 29, no. 6, pp. 44–51, 2012.

[6] A. Nugroho, J. Visser, and T. Kuipers, "An Empirical Model of Technical Debt and Interest," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, pp. 1–8, 2011.

[7] A. Chatzigeorgiou, A. Ampatzoglou, A. Ampatzoglou, and T. Amanatidis, "Estimating the breaking point for technical debt," in*2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pp. 53–56, 2015.

[8] S. Akbarinasaji, A. B. Bener, and A. Erdem, "Measuring the Principal of Defect Debt," in *Proceedings of the 5th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, New York, NY, USA, pp. 1–7, 2016.

[9] D. Falessi and A. Reichel, "Towards an open-source tool for measuring and visualizing the interest of technical debt," in*2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pp. 1–8, 2015.

[10] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the Principal of an Application's Technical Debt," *IEEE Software*, vol. 29, no. 6, pp. 34–42, 2012.

[11] Y. Guo, R. O. Spínola, and C. Seaman, "Exploring the costs of technical debt management – a case study," *Empir Software Eng*, vol. 21, no. 1, pp. 159–182, 2016.

[12] V. Singh, W. Snipes, and N. A. Kraft, "A Framework for Estimating Interest on Technical Debt by Monitoring Developer Activity Related to Code Comprehension," in*2014 Sixth International Workshop on Managing Technical Debt*, pp. 27–30, 2014.

[13] M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, Sep. 1980.

[14] M. S. Krishnan, T. Mukhopadhyay, and C. H. Kriebel, "A Decision Model for Software Maintenance," *Information Systems Research*, vol. 15, no. 4, pp. 396–412, 2004.

[15] Y. Guo, C. Seaman, and F. Q.B. da Silva, "Costs and obstacles encountered in technical debt management – A case study," *Journal of Systems and Software*, vol. 120, pp. 156–169, 2016.

[16] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports (N)," in*2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 476–481, 2015.

[17] G. A. Spedicato, T. S. Kang, S. B. Yalamanchi, and D. Yadav, "A Package for Easily Handling Discrete Markov Chains in R". Retrieved from https://cran.r-project.org/web/packages/markovchain.

[18] P. J. Rousseeuw and M. Hubert, "Robust statistics for outlier detection," *WIREs Data Mining Knowl Discov*, vol. 1, no. 1, pp. 73–79, 2011.

[19] L. T. Miller and C. S. Park, "Decision Making Under Uncertainty—Real Options to the Rescue?" The Engineering Economist, vol. 47, no. 2, pp. 105–150, 2002.

[20] I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "Quantifying the Effect of Code Smells on Maintenance Effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2013.