Estimating Interest on Technical Debt by Monitoring Developer Activity Related to Code Comprehension

Vallary Singh
University of Delaware
Newark, DE USA
vallary@udel.edu

Will Snipes and Nicholas A. Kraft
ABB Corporate Research
Raleigh, NC USA
{will.snipes, nicholas.a.kraft}@us.abb.com

***Abstract*—Evaluating technical debt related to code structure at a fine-grained level of detail is feasible using static code metrics to identify troublesome areas of a software code base. However, estimating the interest payments at a similar level of detail is a manual process relying on developers to submit their estimates as they encounter instances of technical debt. We propose a framework that continuously estimates interest payments using code comprehension metrics produced by a tool that monitors developer activities in the Integrated Development Environment. We describe the framework and demonstrate how it is used to evaluate the presence of technical debt and interest payments accumulated for code in an industrial software product.**

***Keywords*—Technical debt; program comprehension; static analysis**

I. Motivation

Technical debt is a metaphor in which the consequences of decisions that affect the maintenance of a software system, such as decisions regarding architecture and code structure, are described with attributes of financial debt [1]. Economic models proposed by the technical debt community quantify debt using the concepts of principal and interest, where principal is the cost to repay the debt by reworking the code and interest is the cost accumulated by developers working around the debt while the principal is not repaid.

Technical debt has several sources identified in the body of work including debt related to architecture, code structure, code complexity, and code smells [2]. In this work we focus on the area of technical debt related to code structure and code smells (i.e., poor code structure or quality). Several approaches for estimating the technical debt principal are based on heuristics that use measures of structural code quality as inputs to models that estimate effort. For example, Nugroho et al. [3] provide a model for estimating principal using maintainability ratings based on measures obtained via static analysis of code, and a model for estimating interest using estimates of maintenance effort based on change history of code. Curtis et al. [4] also provide a model for estimating principal using measures based on static analysis of code, but in their model, principal is a function of the number of problems, the time/effort required to fix each problem, and the cost of fixing a problem.

Although such models provide the means to estimate debt, it may be difficult to justify reducing technical debt without detailed information about the impact of the debt on developer's day-to-day maintenance activities. Until the debt reaches a point at which it has a substantial impact on the progress or cost of maintenance, developers may be forced to work around areas of the code in which the debt is manifest [5]. Because most developer effort during software maintenance is spent on program comprehension activities such as reading and navigating code [6]–[11], understanding the impact of structural-quality-related debt on code comprehension is of critical importance. In this paper, we propose a framework to support continuous estimation of interest payments on technical debt by monitoring the effort that developers must expend to comprehend code as they complete change tasks.

In our proposed framework, principal estimation is based on measures of code maintainability obtained via static analysis, and interest estimation is based on activity data obtained by monitoring developer actions in the IDE. Our monitoring tool, $Blaze$ [12], records a temporal sequence of developer actions, including code navigation actions and edit actions, in a log. We analyze this log to understand class relationships and to quantify the effort spent by a developer to comprehend individual program elements while completing a change task. By combining this comprehension effort data with the code maintainability measurements, we can provide evidence of how technical debt impacts developer-code-comprehension effort and continuously update interest payment estimates.

By continuously assessing the interest payments on technical debt, the framework enables teams to prioritize debt removal efforts in real-time. Further, by measuring code comprehension on a per-class basis, we permit fine-grained analysis for each class of the cost to repay the debt by fixing the issues or continue paying the interest and working around the debt.

II. Framework

The framework we propose combines related metrics from code structure and code comprehension data sources. We selected some relevant code structure metrics related to maintainability and technical debt using work by Nugroho et al. [3], in which potential code maintainability issues are identified using static code metrics. We use some of the class-level static code metrics provided by $Understand$.[1] to identify code that may have maintainability concerns related to effort required to comprehend the code. In particular, the framework uses the following metrics:

[1] http://www.scitools.com

TABLE I. SAMPLE DATA FROM EVENT LOG

Time-stamp	User	Event	Artifact
22:04:51	N3	View.SourceFile	1acc7366.cs/10
22:04:52	N3	View.OnChangeCaretLine	1acc7366.cs/14
22:04:53	N3	View.OnChangeCaretLine	1acc7366.cs/16
22:04:58	N3	Menu.ViewCallHierarchy	1acc7366.cs/16
22:05:00	N3	View.OnChangeCaretLine	1acc7366.cs/20
22:05:19	N3	View.SourceFile	81c2db1a.cs/1
22:05:22	N3	Edit.Find	81c2db1a.cs/1
22:05:30	N3	Edit.FindNext	81c2db1a.cs/20

Count Class Coupled	number of other classes coupled to
Count Class Base	number of immediate base classes
Count Class Derived	number of immediate sub-classes
Count Line Code	number of lines in the class
Count Declared Method	number of local methods in the class

These class-level code-structure metrics relate to the comprehension metrics on which we based interest estimation. We define low-level code comprehension metrics based on developer actions as they work on code inside of or related to a class.

Data for calculating developer comprehension metrics comes from the $Blaze$ [12] monitoring tool. $Blaze$ anonymously logs developer actions in Visual Studio, including uses of menu commands, shortcut keys, and navigation commands (such as moving the insertion caret and scrolling). Developers at ABB volunteered to install $Blaze$ and to have their actions recorded. The data set we used for the initial study in this paper includes data from two developers that spans over 3 months.

Table I shows a sample of the log data, where each row contains the date (not shown) and time for an action, the unique ID for the developer who performed the action, the name of the action that was performed, and the name of and line in the file in which the action occurred.

Our approach to analyzing the developer activity data was to establish sessions that segment the stream of activity into periods in which the developer is focused on a particular class. We define a session as fixed length time window where the developer is investigating a certain class that we call the central class. The session time window begins the first time a developer visits a particular class and ends with the last time the developer visits that class within a fixed length time window. The length of the time window is a variable that we investigate in Section III.

Figure 1 illustrates the session concept. The session starts with the first visit to Class A under the green circle. After that, other classes C and E are visited, including a visit to Class A again before the last visit to Class A under the red hexagon. After the last visit, the session time window expires (assuming a fixed-time window). The second session starts with a visit to Class A. In this session, classes C and E are visited before the developer returns to Class A. Thus, there are two sessions for Class A, in which there are a total of five visits to Class A, three visits to Class C, and two visits to Class E.

Within a session, we calculate metrics related to comprehension effort as follows:

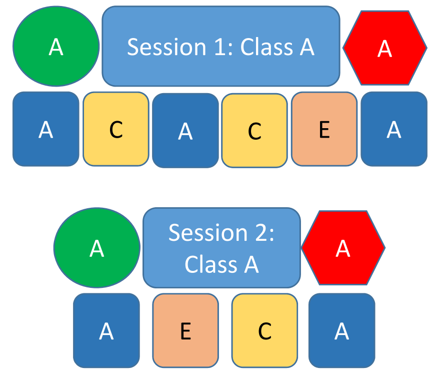

Fig. 1. Conceptual View of Sessions

 # Sessions
: number of time-window sessions for each class

 # Class Visits
: number of times the central class is visited in a session

 # Other Class Accesses
: number of unique (non-central) classes visited in a session

Time Spent in Class
: time spent in the central class for a session

Time Spent in Other Classes
: time spent in all other classes in a session

The class-level metrics for code structure are related to comprehension metrics through the name of the source file in the $Blaze$ data corresponding to the class. In cases where the source file contains multiple classes, the structure metrics were aggregated.

We can investigate how much the Feature Envy smell where a method makes too many calls to other classes to obtain data or functionality. We assess the level of Feature Envy using using the $ClassCoupled$ static metric and quantify the effect that has on comprehension using the $\#OtherClassAccesses$ metric and evaluate the effort this smell generates using the $TimeSpentinOtherClasses$ measure. We can estimate the impact of the Large Class smell, where a class is larger than typical, by using the $CountLineCode$ static metric. The assess the effect on developer comprehension, we can use the metrics $\#ClassVisits$, $\#Sessions$ to determine the difference in navigation behavior and evaluate the $TimeSpentinClass$ to assess the total effort resulting from the smell.

By defining the measurement framework, the calculation of interest payments on technical debt related to code structure will use time spent measurements for classes with static metrics that indicate code is difficult to maintain or contains smells. Thus static metrics will indicate the possible presence of technical debt in a class and comprehension effort metrics will quantify the effort to comprehend those classes. The study of correlation between static metrics and comprehension effort is planned for future research.

III. ANALYSIS

As previously mentioned, we first establish sessions to investigate developer activity. We define a session as a moving window of time during which a developer is investigating a

TABLE II. CLASS STRUCTURE DATA

Class	Count Class Base	Count Class Coupled	Count Class Derived	Count Line Code (Top % in project)	Count Declared Method (Top % in project)
A	10	272	1	0.05%	0.13%
B	3	98	0	0.12%	2.2%
C	1	32	0	0.7%	6.9%

TABLE III. CLASS DATA FROM LOGS FOR DEVELOPER X

Class	# Sessions	# Class Visits	# Other Class Accesses	Time Spent in Class	Time Spent in Other Classes
A	74	294	78	39 hr. 34 min. 23 sec.	16 hr. 39 min. 36 sec.
B	32	88	52	2 hr. 54 min. 20 sec.	8 hr. 54 min. 18 sec.
C	28	77	52	4 hr. 28 min. 25 sec.	9 hr. 15 min. 41 sec.

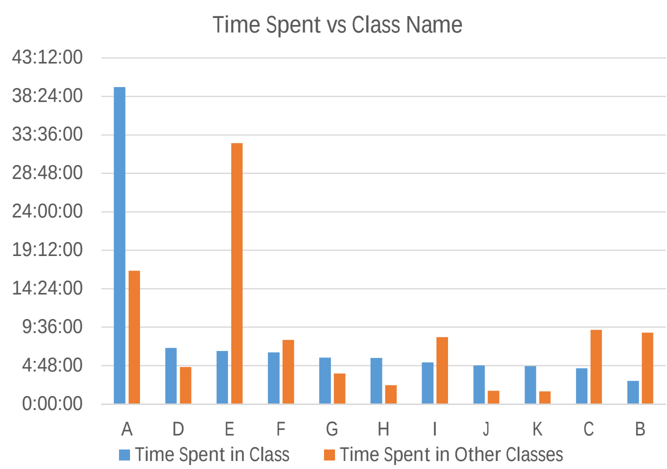

Fig. 2. Time Spent Vs Class Name

particular class. For this initial study, we investigated window sizes of 4 hours, 8 hours, 12 hours, and 16 hours. For each session we calculate the comprehension metrics listed in Section II. As a change task is completed, the key classes for the task will have large numbers of sessions, as well as large numbers of visits during those sessions. Time spent in the central class and in the other classes during a session helps to assess the effort required to understand the classes visited in the session.

The $Blaze$ log for a developer includes all of the navigation activities performed by the developer in the IDE, as well as time-outs for periods of inactivity. We first filtered each developer log, retaining only navigation activities related to classes, time-outs, and IDE exits. Next we calculated the time spent by the developer in each class. We observed that there were instances when the developer visited a class for less than one second before switching to another class. Working on the assumption that the developer could not attain any additional understanding in less than a second, we attributed such class visits to random clicks and removed all such log entries. We then calculated various parameters to help in identifying the central class, as well the other classes necessary for comprehending the central class. Finally, we calculated the comprehension metrics defined in Section II.

We observed that the change in the number of sessions for each class was small when switching between a 4-hour moving window and a 8-hour moving window. The small magnitude of this change indicates that developers do not often work in 8-hour windows, but rather tend to work in 4-hour windows. Moreover, the 8-hour moving window returned the same number of sessions as did the 12-hour and 16-hour windows. Thus, we decided to use the 4-hour sliding window for further analysis.

Table II lists the structural code metrics for each of the classes that we have used for this study. These code metrics were calculated for a project containing 9,888 classes. Class A had the largest $Count\ Class\ Coupled$ value among all the classes in the project. The other such code metrics for class A were all in the top 2% when compared to all the classes in the project. The large values for $Count\ Line\ Code$ and $Count\ Declared\ Method$ indicate class A may have the $Large\ Class$ smell [13] and suggest that it perhaps consumes a significant portion of the maintenance effort for the project.

In our analysis of the code comprehension data, we show the distribution of time for the most active classes in the data set. Figure 2 shows a graph of the time spent in class and time spent in other classes for the top 11 classes in the data set. We ordered the data in descending order of time spent in the class. While class A has the greatest time spent in the class and time spent in other classes, the ratio varies among other classes. In class E the time spent in other classes while in session is much more than the time spent in the class itself. This shows the fact that there are times when the time spent comprehending other classes is much more than the time spent in the class itself.

Table III lists data for three classes worked on by the developer that we studied. The five rightmost columns list the values for the comprehension metrics that we defined to help quantify technical debt. For developer X the number of sessions for class A is large, as are the numbers of (central) class visits and other class visits. This suggests that developer X visited class A often over a long period of time, and also visited other classes frequently while working on class A. The data show that developer X spent more than 39 hours working on class A and more than 16 hours on other classes during the 74 sessions for class A. The 16 hours that developer X spent during sessions for class A on other classes can be interpreted as the cost of comprehending class A, which can in turn be viewed as technical debt.

The second row of the table shows that during the 32 sessions for class B, developer X spent nearly 3 hours on class B but nearly 9 hours on other classes. This indicates that in the sessions for class B, the developer spent three times as long in other classes as he spent in the central class. Considering only time cannot lead to such conclusions, and we need to make sure that the central class for each session is actually central to the task at hand. In the case of class B, there are 32 sessions in which class B was visited 88 times. Thus, we can state that the developer continuously returns to class B, indicating that it is central to the task.

Considering both the structural code metrics and the comprehension metrics reveals that class A is large and is highly coupled to other classes in the project, and also reveals that the developer spent a large amount of time working on class A. Our preliminary investigation of the code and comprehension metrics revealed several interesting results. When consider the values of $Count\ Class\ Coupled$, $\#\ Other\ Class\ Accesses$, and $Time\ Spent\ in\ Other\ Classes$, we observe that for class A, which is the most highly coupled class in the project, $\#\ Other\ Class\ Accesses$ and $Time\ Spent\ in\ Other\ Classes$ are both large. However, for classes B and C, we observe that the values of $\#\ Other\ Class\ Accesses$ and $Time\ Spent\ in\ Other\ Classes$ are comparable, event though class C is less coupled than class B. Related to the $Large\ Class$ smell, we observe that variation in $\#\ Sessions$, $\#\ Class\ Visits$, and $Time\ Spent\ in\ Class$ aligns with changes in $Count\ Line\ Code$. Among all three classes, decreases in $Count\ Line\ Code$ correspond to decreases in the values of $\#\ Sessions$ and $\#\ Class\ Visits$.

Combining the $TimeSpentinClass$ and the $TimeSpentinOtherClasses$ data for Class A, we estimate the interest payments on that class as 45 minutes per session spent comprehending Class A and related classes. Against the average comprehension time per session we see this is an increase of x minutes per session meaning that y minutes can be attributed to the technical debt in that class. If we consider the technical debt in Class B, we calculate 22 minutes per session which against the average of x minutes per session results in y minutes attributable to technical debt.

IV. Conclusions and Future Ideas

In this paper we proposed a framework in which code maintainability data and comprehension effort data are combined to support continuous updates of interest payment estimates, which in turn supports real-time prioritization of debt removal efforts. The primary contribution of our proposed framework is the integration of developer activity data with static code metrics and the concomitant improvements in understanding of developer comprehension effort and in the accuracy with which interest payments can be estimated. An initial investigation of data that we collected from ABB developers demonstrates the feasibility of the framework and provides examples of how the developer activity data work in concert with structural code metrics to reveal new information about developer comprehension effort.

Our next step will be to conduct a large-scale statistical analysis of comprehension and structural metrics to better understand the correlations and levels of technical debt that drive increased comprehension effort. We plan to further develop the framework and to use it to answer a number of questions about the relationships between developer comprehension effort and technical debt. For example, we plan to further develop the framework to include other comprehension metrics such as the number of edits to a class that will allow evaluation of the Shotgun Surgery smell where multiple classes are modified for a change. To improve the accuracy of the comprehension data, we plan to detect the central class based on edit actions as well as navigation and search actions during a session. We also plan to conduct an observational study of developers to validate the estimates of interest payments during maintenance activities.

Acknowledgment

The authors would like to thank ABB Corporate Research for supporting this work.

References

[1] W. Cunningham, "The WyCash portfolio management system," *SIGPLAN OOPS Mess.*, vol. 4, no. 2, pp. 29–30, 1992.

[2] I. Ozkaya, P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," *IEEE Software*, vol. 29, no. 6, pp. 0018–21, 2012. [Online]. Available: http://dx.doi.org/10.1109/ms.2012.167

[3] A. Nugroho, J. Visser, and T. Kuipers, "An empirical model of technical debt and interest," in *Proc. 2nd Int'l Wksp. Managing Technical Debt*, 2011, pp. 1–8.

[4] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the size, cost, and types of technical debt," in *Proc. 3rd Int'l Wksp. Managing Technical Debt*, 2012, pp. 49–53.

[5] I. Ozkaya, P. Kruchten, R. Nord, and N. Brown, "Second international workshop on managing technical debt (MTD 2011)," in *Proc. 33rd Int'l Conf. Software Engineering*, 2011, pp. 1212–1213.

[6] R. Fjeldstad and W. Hamlen, "Application program maintenance study: Report to our respondents," in *Tutorial on Software Maintenance*, G. Parikh and N. Zvegintzov, Eds. IEEE Computer Society Press, 1982, pp. 13–30.

[7] T. Standish, "An essay on software reuse," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 5, pp. 494–497, 1984.

[8] A. von Mayrhauser, A. M. Vans, and A. E. Howe, "Program understanding behavior during enhancement of large-scale software," *J. Softw. Maint: Res. Pract.*, vol. 9, no. 5, pp. 299–327, 1997.

[9] A. Ko, B. Myers, M. Coblenz, and H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Trans. Softw. Eng.*, vol. 32, no. 12, pp. 971–987, 2006.

[10] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proc. 28th Int'l Conf. Software Engineering*, 2006, pp. 492–501.

[11] R. Tiarks, "What programmers really do — an observational study," *Softwaretechnik-Trends*, vol. 3, no. 2, 2011.

[12] W. B. Snipes, A. R. Nair, and E. Murphy-Hill, "Experiences gamifying developer adoption of practices and tools," in *Companion Proc. 36th Int'l Conf. Software Engineering*, 2014, pp. 105–114.

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