A Proposed Sizing Model for Managing 3rd Party Code Technical Debt

Will Snipes
ABB Corporate Research
will.snipes@us.abb.com

Srini Ramaswamy
ABB, Inc.
srini@ieee.org

ABSTRACT

Commercial software development projects frequently build code on third-party components. However, depending on third-party code requires that projects keep current with the latest version of each component. When projects do not stay current, they begin to incur a form of technical debt where API calls that have been deprecated remain in the code base. At some point, projects must upgrade the third-party component to remain on a supported version of the component. Then the projects incur the cost of paying down the debt that was built up over time. The model described herein intends to estimate the cost of paying down the debt for aging third-party components.

The model is a sigmoid curve that exponentially increases the size of changes required to migrate to the new version as a function of time asymptotically approaching the size for replacing the entire component. The longer the number of elapsed years, the greater the increase in the principal measured as the number of affected lines of code in the user of the third-party software component. This exponential increase in principal is reasonable when we consider longer time horizons are more likely to require replacement of the third-party component entirely due to newer technologies becoming available. Effects of the model were estimated using Monte Carlo simulation due to limited examples of third-party technical debt available for modeling.

CCS CONCEPTS

• **Software and its engineering** → *Maintaining software*;

KEYWORDS

software components,software sizing,measurement,technical debt

ACM Reference Format:
Will Snipes and Srini Ramaswamy. 2018. A Proposed Sizing Model for Managing 3rd Party Code Technical Debt. In *TechDebt '18: TechDebt '18: International Conference on Technical Debt , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3194164.3194179

TechDebt '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5713-5/18/05…$15.00
https://doi.org/10.1145/3194164.3194179

1 INTRODUCTION

Technical debt is defined as making technical compromises that are expedient in the short-term, but that create a technical context that increases complexity and cost in the long-term [2]. Technical debt has two cost factors, the amount of effort required to correct the deficiencies (the debt's principal) and the effort required to maintain code in the presence of the deficiencies (the debt's interest payments) [5]. These factors can be applied in a variety of use cases to describe technical debt due to design decisions, architecture decisions, and decisions related to testing. We propose to apply technical debt factors to decisions related to third-party component management.

Software development projects frequently build code on third-party components, which accelerates the development cycle and leverages code with proven qualities such as security, extensibility, and reliability. However, depending on third-party code requires that a project keeps current with the latest version of each component. When a project does not stay current, it incurs technical debt where deprecated API calls remain in the project's code base. As a project builds new functionality, these deprecated API calls may increase in number in the code base, making the technical debt grow. At some point, the project must upgrade the third-party component to remain on a supported version of the component. Then the project incurs the cost of paying down the debt that was built up over time.

In this paper, we discuss aging third-party components in an industrial software project and present an idea on how to model the size of impact when projects fall behind the current version of these components. We consider factors such as the size of the code base using the third-party component, the degree of API deprecation between the version used and the current version of the third-party component, and the age in years of the third-party software component version being used. These factors become parameters of a model that estimates the size of the impact of upgrading an aging third-party component to the current version.

The remainder of this paper is organized as follows: Section 2 discusses related work, Section 3 discusses the concepts and measurements included in the model, Section 4 provides details on the proposed model and simulation results, Section 5 discusses the results and their implications for future work, and Section 6 highlights conclusions.

2 RELATED WORK

The body of related work studies some factors and practices of managing third-party code. We found some common ideas in the factors driving technical debt in third-party software. We also found support for factors related to the contribution of aging to software decay. Finally, a report on tooling for managing third-party software

demonstrates that industry has challenges managing upgrades to third-party components.

Bavota, et al. [3] investigate the evolution of library usage in the Apache ecosystem and evaluate the influences on projects to upgrade their dependent libraries. Bavota, et al. find that an upgraded library has limited impact to the using project (5% of changes to the project) unless the library is a framework type of library. They also find that changes to API interfaces discourage upgrading to the latest version of a library because of the required changes in the project code to support the new API. This leads to the idea that deprecated APIs create technical debt in the code that interfaces with the third-party component. Thus one of our key metrics becomes the percent of API deprecation in the newer version of the third-party code.

Kula, et al. [7] evaluate the trusted (latest release) adoption vs. latent (prior release) adoption of libraries in open source Java programs. They found that 82% of existing systems adopt the latest release when a new version of a library is issued. In an earlier work, Kula et al. [6] study library aging as a factor of usage in the Maven ecosystem. They found that library adoption follows one of three forms (linear, second order, or higher order models) and that typically library usage grows until there is a replacement for the library such as a new version of the library. These papers support the idea that software products using third-party libraries should upgrade as new versions are released.

Supporting the idea of technical debt being created by aging third-party software, Parnas reported on various factors that contribute to the decay of software as it ages [8]. Because the project using libraries does not evolve to the new version, it suffers from the cause Parnas terms "lack of movement" where the project does not evolve with changing requirements of its user base. This leads to costs described by Parnas as the "Inability to keep up" where changes to migrate to the new version of a software package become spread across a larger portion of the project's code base.

Annosi et al. described how one corporation built tooling to manage their dependencies on third-party software across product lines [1]. Ericsson used a tool to drive decision making regarding when to update versions of off-the-shelf software components in their system. The benefit of such a tool has been the greater facilitation of the decision process by exposing data on change and bug history to developers when they are deciding to upgrade off-the-shelf software. The existence of a tool for managing third-party software supports the case that industry needs help managing third-party software.

3 BACKGROUND

Technical debt has two components, the principal (the cost to repay the debt) and interest, the ongoing cost of working around the debt. We consider that aging third-party components incur a principal cost related to the cost to migrate the API calls in the using code to the newer version of the API. With each successive new version of the third-party code that comes out, the project incurs more principal due to additional deprecated API calls.

When the project builds more and more functionality on the old versions only testing on the old versions there arises a greater chance the project will not be compatible with the new version of third-party code. The more versions the project is behind, the greater chance there is of having to do significant changes. So the cost of deferring updates becomes the accumulated principal cost of maintaining compatibility when the project has to upgrade.

The cost to move to a new version of code includes the cost to update references to deprecated APIs, address other incompatibility issues that are systemic, and regression test the software with the new third-party code. The cost involved when APIs become deprecated depends on the scope of use of the third-party component in the software. If the third-party component is used extensively as in the case of a framework, the cost can potentially be spread across the whole application size. If the third-party component is abstracted by interface classes, the cost may be limited to the interface itself. The regression test cost is considered related to the size of the affected code base as well.

The following factors were considered as drivers of the cost to upgrade aging third-party components. The scope of impact (size) relates to the amount of code in the project that interfaces with the third-party code. The accumulated principal cost for a given scope of impact is a function of the age and the degree of deprecation.

Size: reflects the size of the project or project component that uses the third-party software. In the case of a framework, the size may be the entire size of the project. For a well-encapsulated third-party component where the interface has an abstraction layer in the project, the size may be limited to the abstraction layer.

Degree of Deprecation: the percentage of API method interfaces used by the component that are deprecated by a new release of a third-party component. Deprecated method calls include calls that no longer exist and previously existing calls that are recommended to avoid in the API. The deprecated functionality is estimated as the percentage of functions deprecated in third-party code between the version being used and the currently supported version.

Age: number of years elapsed between the release date of the version used in the subject code and today. The age component expresses the concept that the longer a project waits to upgrade, the higher the accumulation of dependencies and the higher the estimate to address those dependencies.

We considered models representing exponential growth in the size of the impact of aging third-party components. The sigmoid model is chosen because of three characteristics that make it useful in this application. The model provides a period when the third-party component has aged very little where the size of impact does not grow as quickly. The sigmoid model then has a rapid growth that approaches but does not exceed the complete size of the affected software in the subject project. These three characteristics led us to propose a sigmoid model as the model of technical debt due to aging third-party components.

4 MODEL

The model presented below estimates the number of affected lines of code when third-party software is updated to the current version. These affected lines of code include code that is modified or must be retested when the third-party software is updated. The Verhulst-Pearl equation [9] results in a sigmoid curve that explains some of the behavior of third-party software aging. The model exponentially increases the cost of migrating to the new version as a function of

Table 1: Monte-Carol Simulation Parameters

Variable	Low	High
Years	0	10
Deprecation	0%	100%
Size	100K	5,000K

time asymptotically approaching the cost of replacing the entire component. The model equation is expressed as follows:

$$f(x) = \frac{Lk}{1 + e^{(x_0 - x)}} \quad (1)$$

Where:

k = degree of deprecation for the third-party component's API

x = age in years elapsed since the release date of the version of third-party code used in the project

x_0 = sentinel year for the age of components (recommended value of 5)

L = size (LOC) of the software that interfaces with the third-party component

Estimating effort for upgrading third-party software involves estimating the Lines of Code (LOC) that are affected by the changes to the API of the third-party code. The model takes this into account by considering the percentage of API methods used that are deprecated in the newer version of the third-party code. The exponential factor in the model considers how much time elapsed between the release of the version of third-party code currently in the subject component and the version that is upgraded to. Age, measured in years, reflects several ideas such as gaps in knowledge/training that may have developed in the team, the aging nature of the third-party code itself which may have reached the end of life from a support perspective, and the time elapsed for other, better, solutions to have entered the marketplace. The model uses this exponential factor to consider that at some age it may be expedient to reevaluate the competitors to the third-party code and determine whether it is more cost effective to remove and replace the chosen solution with a better competitive product.

Monte Carlo simulation was used to determine the potential range of responses to the input space. The ranges of values used in the simulation are independently and randomly generated for each variable in the model. For the degree of deprecation (k), values are simulated between 0 and 100%. The age in number of years (x) is bracketed between 0 and 10 years. For Lines of Code (L), we simulated values between 100K and 5,000K lines of code as shown in Table 1. For these simulations, we selected a value for sentinel year (x_0) of 5. The value of (x_0) reflects the pivot point where the growth in cost switches from increasing growth to decreasing growth.

Results of the Monte-Carlo simulation in Figure 1 show the plot for all size factors of the model. Each color represents the distribution of 400 samples of each size factor. Because of the exponential relationship with time, longer number of elapsed years indicates increased size in the number of affected lines of code. This estimated value makes sense when we consider longer time horizons are more likely to have API changes and compatibility issues for the component. As duration reaches the maximum, it is more likely to require replacement of the third-party component entirely due to newer technologies becoming available.

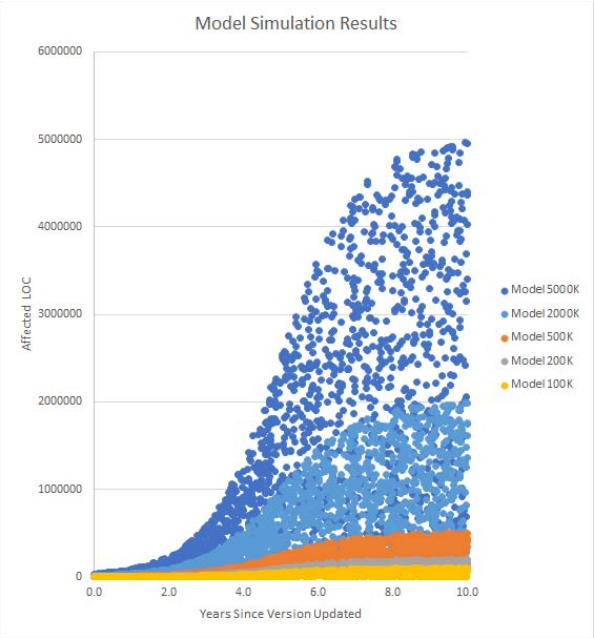

Figure 1: Monte Carlo Simulation

The 3D graphs allow exploration of two of the key input factors simultaneously. We fix the size factor (L) at 100K LOC, then vary the number of years since release and degree of deprecation according to the values shown in Table 1. Figure 2 shows a 3D plot of the size as a result of varying Age and Deprecation. The S-shaped curve can be seen along the x-axis where age has a prominent effect on the size estimate.

Figure 3 shows a 3D plot of the size as a result of varying Age and Deprecation. The effect of deprecation can be seen more clearly when it is displayed on the x-axis. As deprecation varies between 0 and 100% the size impacted increases in a linear fashion depending on the age of the included third-party code.

5 DISCUSSION AND FUTURE WORK

Modeling the cost of deferring upgrades to third-party software helps developers because it communicates the trade-off decisions being made to management at the time when a lower cost alternative can be pursued. Often developers are faced with resource-constrained decisions about what to include in the backlog for a release. Issues of upgrading third-party components can seem like a low-impact and receive a low-priority during backlog planning. By expressing the impact through this model, we propose that the impact of deferring upgrades is actually much higher in the long-run.

The environment that supported the development this model is based on a software product that has been in production use for many years. The product release cycle for new feature releases is on the order of one per year, therefore the product has a long

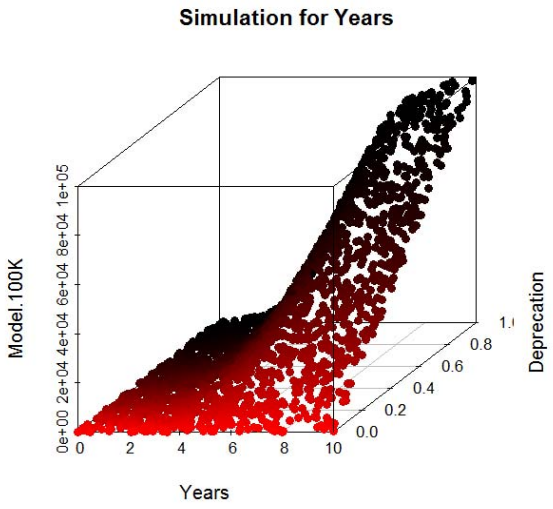

Figure 2: Monte Carlo Simulation for Age

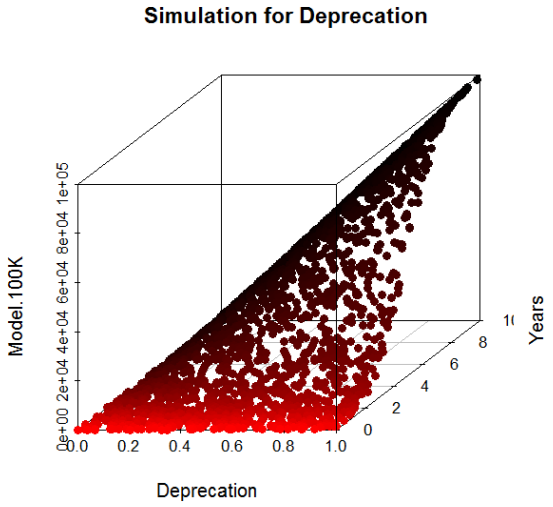

Figure 3: Monte Carlo Simulation for Deprecation

development cycle. The product is a fairly stable code base, but it is modernizing its architecture to current standard practices in the industry including adopting open source components to replace aging proprietary third-party components. Based on experiences with this product, we applied this model for estimating the size impact of third-party component upgrades. Products with different release cycle timing and other characteristics may have different results.

Other models that could be considered for the cost of deferring upgrades to third-party software include the inverse power law, which would eliminate the grace period given through a sigmoid curve and have an immediate exponential increase in the cost to upgrade in later releases of the project.

Due to the limited amount of data available, this model has not been validated. The model was reviewed by the development organization. The opinion of the subject matter experts is that the model is representative of the cost of deferring software updates for third-party dependencies.

Future work will focus on data collection around the cost of deferred and not-deferred upgrade strategies in product planning. Key future research questions include: When organizations adhere to the strategy of upgrading to the latest release early do they save effort in the long run as compared to deferring upgrades until absolutely necessary? Does the grace period exist for deferring upgrades as displayed by the sigmoid model or is an inverse power law model a better fit for the cost to upgrade? Is the conversion between size estimate and cost for upgrading third-party software already understood by estimation models such as COCOMO [4]? What other factors in the cost of upgrading third-party software may be important in a cost model?

6 CONCLUSIONS

We have proposed a model for estimating the size impact of upgrading third-party components on the host system. The model is a sigmoid curve that provides a grace period where the impact remains low, followed by an exponential increase that approaches the cost of replacing the entire affected component in the host system. The model uses key factors of size, the degree of deprecation, and the number of years since the last update. We discuss the results and opportunities for future work to explore the cost of upgrading third-party components in a project.

REFERENCES

[1] Maria Carmela Annosi, Massimiliano Di Penta, and Genny Tortora. 2012. Managing and assessing the risk of component upgrades. In *Proceedings of the Third International Workshop on Product LinE Approaches in Software Engineering*. IEEE Press, 9–12.

[2] Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman. 2016. Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). *Dagstuhl Reports* 6, 4 (2016), 110–138. https://doi.org/10.4230/DagRep.6.4.110

[3] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the Apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering* 20, 5 (2015), 1275–1317.

[4] Barry Boehm, Bradford Clark, Ellis Horowitz, Chris Westland, Ray Madachy, and Richard Selby. 1995. Cost models for future software life cycle processes: COCOMO 2.0. *Annals of Software Engineering* 1, 1 (16 Dec. 1995), 57–94. https://doi.org/10.1007/bf02249046

[5] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. 2012. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software* 29, 6 (2012), 18–21.

[6] Raula Gaikovina Kula, Daniel M German, Takashi Ishio, and Katsuro Inoue. 2015. Trusting a library: A study of the latency to adopt the latest maven release. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 520–524.

[7] Raula Gaikovina Kula, Daniel M German, Takashi Ishio, Ali Ouni, and Katsuro Inoue. 2017. An exploratory study on library aging by monitoring client usage in a software ecosystem. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 407–411.

[8] David Lorge Parnas. 1994. Software aging. In *Proceedings of the 16th international conference on Software engineering*. IEEE Computer Society Press, 279–287.

[9] Raymond Pearl and Lowell J Reed. 1920. On the rate of growth of the population of the United States since 1790 and its mathematical representation. *Proceedings of the national academy of sciences* 6, 6 (1920), 275–288.