Technical Debt from the Stakeholder Perspective

Ted Theodoropoulos
Acrowire, LLC
235 Lincoln Street
Charlotte, NC 28203
011 +1 (704) 900 1601
ted@acrowire.com

Mark Hofberg, CISA, CRISC
1530-K S. Church Street
Charlotte, NC 28203
011 +1 (704) 281 1976
markhofberg@gmail.com

Daniel Kern, PhD
PO Box 49193
Charlotte, NC 28277
011 +1 (704) 975 6248
info@dkern.com

ABSTRACT

The concept of technical debt provides an excellent tool for describing technology gaps in terms any stakeholder can understand. The technical debt metaphor was pioneered by the software development community and describes technical challenges in that context very well. However, establishing a definitional framework which describes issues affecting quality more broadly will better align to stakeholder perspectives. Building on the existing concept in this way will enable technology stakeholders by providing a centralized technical debt model. The metaphor can then be used to consistently describe quality challenges anywhere within the technical environment. This paper lays the foundation for this conceptual model by proposing a definitional framework that describes how technology gaps affect all aspects of quality.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management - life cycle, productivity.

General Terms

Management, Measurement, Design, Economics, Reliability, Standardization.

Keywords

Technical debt, risk management, software quality, governance

1. INTRODUCTION

The technical debt metaphor was coined by Ward Cunningham in 1992 to provide a basis for describing the implications associated with shortcuts taken during the software development process [1] and the inherent need for refactoring as applications evolve. Cunningham's description of technical debt remains the most commonly referenced and is defined as follows.

"Technical Debt includes those internal things that you choose not to do now, but which will impede future development if left undone. This includes deferred refactoring.

Technical Debt doesn't include deferred functionality, except possibly in edge cases where delivered functionality is 'good enough' for the customer, but doesn't satisfy some standard (e.g., a UI element that isn't fully compliant with some UI standard)."

This metaphor allows us to leverage our existing understanding of debt mechanics and provides an excellent tool for describing technology gaps in financial terms. The ability to use business-friendly terms and concepts facilitates communications between IT departments and other stakeholders.

The software development community conceived the concept and has championed its use over the last two decades. This has resulted in a definitional framework heavily focused on factors influencing intrinsic quality. Understandably, pioneers of the concept used technical debt to describe quality issues within their immediate concern and control. This limited perspective may have impaired the concept's ability to gain traction with other stakeholders.

The purpose of this paper is to propose a definitional framework that will bridge several gaps that exist today. First, the definition of technical debt we propose will better align to stakeholder concerns. Second, it will account for the fact that compromises made in one area of software quality often impacts others. Lastly, it will promote better collaboration between product owners and technology teams when managing quality. It is not the goal of this paper to provide a complete conceptual model of technical debt. Rather, it is to establish a broad definitional framework from which a conceptual model could be built.

2. STAKEHOLDER PERSPECTIVE

Technology stakeholders are those individuals or groups who are affected by an organization's technical environment and have a vested interest in its well-being. These stakeholders can be internal to the organization such as business executives, technology leaders, risk managers and end users. They can also include external parties such as regulators, shareholders and customers. Generally, these stakeholders are equally concerned if "the right system was built" and whether or not "the system was built right." Therefore, it matters little whether interest is being paid on a code base in need of refactoring or on missing features that are impacting efficiency.

Managing technology gaps consistently across the organization provides many stakeholder benefits. Consistency allows business executives to allocate resources more effectively by providing a common economic basis to prioritize initiatives. This approach also paves the way for centralized governance mechanisms. Centralized models provide better transparency around operational challenges thereby improving risk management processes. Transparency and effective risk management decreases the risk profile of the firm which encourages external investment and lowers the cost of capital.

3. QUALITY INTERDEPENDENCE

In an influential paper evaluating different perspectives on quality, David Garvin describes several different lenses through which quality is typically evaluated [2]. Technical debt's current definitional framework takes on a perspective similar to what Garvin describes as the "manufacturing view." This view

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

emphasizes impacts to quality that are internal to system development. This includes specifications compliance, design effectiveness, defect rates and maintenance challenges. If we highlight the ISO quality attributes [3] emphasized by technical debt's current definitional framework it might look something like Figure 1.

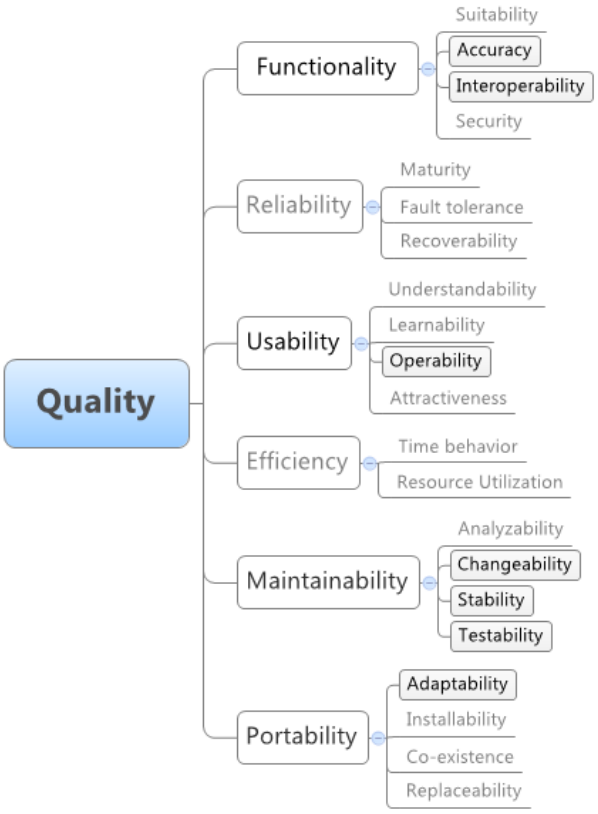

Figure 1: ISO 9126 Quality Model [3]

Areas of technology infrastructure are interrelated to form a structure similar to a biological ecosystem. Each area of the ecosystem has inherent dependencies on the others. Without accounting for these dependencies, organizations could pay off technical debt in one area and unknowingly accumulate debt in another area with a higher interest rate. We will now explore several examples that demonstrate the interdependent nature of software quality.

3.1 Deferred Functionality

When features are deferred, the Operability of that system is impacted. This can lead to undesirable user interactions and ultimately compromise the Accuracy of the system. One example is when users "cram" data in unintended places from missing attributes. In this scenario, Accuracy would be impacted by compromises made in Operability. This is depicted in Figure 2 with a pencil icon next to Operability which changed and a red arrow next to Accuracy which was unintentionally impacted.

Figure 2: Deferred functionality's effect on overall quality

To illustrate, consider the example of an entity whose ability to be deactivated is missing or poorly implemented in the user interface. To differentiate between active and inactive records, users might concatenate strings such as "(Inactive)" in the entity's name field. Another common example is when there are first and last name fields for a customer entity, but none for middle name or suffix. Users will sometimes append the middle initial to the first name and the suffix to the last name. This behavior has a denormalizing effect on the database which will impact downstream processes.

3.2 User Experience

Poor user experience can drive the use of what process experts call hidden factories. Missing or poorly implemented features encourage users to create workarounds resulting in hidden factories that decrease the quality and security of the data. The Operability shortcomings in an application can unintentionally bring about a decrease in Accuracy and Security as shown in Figure 3.

Figure 3: User experience's effect on overall quality

An example of this is when data is extracted from an application into spreadsheets for processing or reporting. These hidden factories appear when features are missing from an application or the functionality is more difficult to use than the workaround. The logic contained in these spreadsheets is usually not sanctioned or controlled, and leads to inconsistency and several "versions of the truth." Additionally, once the data leaves the confines of the system, any controls designed to protect it are taken out of play.

3.3 Control Weaknesses

Unintentional debt can also result from missing or inadequate controls. Validation control weaknesses decrease data quality by allowing invalid data to be entered into the system intentionally (e.g. cramming) or unintentionally (e.g. duplicates, typos, etc.). Access control deficiencies also present risk to data quality and Security as shown in Figure 4.

Figure 4: Control weaknesses' effect on overall quality

Access controls keep unauthorized users from viewing or editing certain data. Users are not permitted access to certain data because they do not have the training or clearance to interact with it. Data integrity will suffer when the controls that protect that data break down.

3.4 Data Quality

If we think of the technical environment as an ecosystem, data would be analogous to the water flowing through a river system. A polluted river impacts downstream areas in much the same way as the toxic effects of defective data ripple across a technology infrastructure. Bad data will impose cascading interest on any process or system that interacts with it. Inaccurate or incomplete

data impairs a system's ability to provide reliable intelligence and integrate with other systems.

Figure 5: Data quality's effect on overall quality

Data quality issues also introduce new technical debt by requiring workarounds in data movement logic. These workarounds increase complexity which makes enhancements and testing more challenging.

4. QUALITY COLLABORATION

When prioritizing a product backlog, business partners need to understand the quality implications of their decisions. To avoid the accumulation of unintentional technical debt, technology teams should provide feedback on the impact to overall quality associated with prioritization strategies. When stepping through the backlog there are three high level alternatives for each feature.

A. Do it right – no debt
B. Do it quick – take shortcuts and accrue debt
C. Do it later – defer features and accrue debt

Because Option C has been considered to primarily impact extrinsic quality, it falls outside of the current scope of technical debt. This creates a blind spot in technical debt's description of issues affecting software quality. As we have seen from the examples in Section 3, gaps in extrinsic quality can affect intrinsic quality.

When development teams focus exclusively on intrinsic quality, it is left to business partners to manage extrinsic quality. Technology teams should play a central role in managing all aspects of software quality because they have unique insights. Designers understand the relationship between user experience and adoption, database developers know what to expect if proper validation controls are not present, and systems administrators are aware of the risks associated with legacy platforms. This expertise must be leveraged to ensure business partners do not have a blind spot in their perspective on technical debt.

5. PROPOSED DEFINITION

In order to align technical debt's definitional framework to stakeholder objectives, it should transition from what David Garvin describes as a manufacturing view to a value-based view of quality [2]. The value-based view equates quality to what the customer is willing to pay for and encourages everyone to consider trade-offs between cost and quality [4]. To capture this value-based perspective, the definitional framework should be revised to capture all technical gaps that present risk to required levels of quality. Our proposed definition of technical debt reads as follows.

"Technical debt is any gap within the technology infrastructure or its implementation which has a material impact on the required level of quality."

The "required level" of quality will vary from one environment to the next based on the processes it supports. An application that has no sensitive data might not require certain controls. The fact that such an application has no access controls would not be considered technical debt as we have defined it because that functionality is not required. However, if the application has access controls that are not needed but cost money to maintain, it would qualify as debt. This might occur if the controls decrease productivity or staff is needed to manage them. Technical debt represents the cost associated with gaps between an application and its quality requirements.

5.1 Definition Scope

The term "technical" debt implies that the issue driving the cost relates to something in the technology infrastructure. If the principal amount of the debt relates to an investment needed in the technical environment it should be considered a technical debt. The interest may be incurred outside of the technical environment but the principal must represent a technology expense. It follows that if the investment needed to pay back the principal would be made outside the technical environment it should not be considered technical debt.

Therefore, gaps in supporting processes should not be considered technical debt. Only the technology required to support defined processes should be considered debt. For example, the absence of a business continuity plan would not qualify as debt but the technology necessary to support such a plan would. Without a defined process, requirements would not exist and calculating the principal of the debt is not possible. While process gaps are not themselves debt, they can and often do lead to technical debt. Inadequate application lifecycle management processes are such an example. Failure to employ proper quality assurance methods or development standards will result in technical debt in the form of decreased code quality.

There are gray areas that will require consideration, however. An example would be supporting documentation. We propose including supporting items in this gray area if they are considered a tangible extension of the technical asset. If a software package were to be purchased from a vendor, one would expect system documentation to be a part of that product package. Therefore, gaps in system documentation could be considered debt.

6. CONCLUSION

Regardless of how accurate the perception might be, technology departments are viewed as being out of touch with the rest of the business in many organizations. Part of the reason behind this is because technology and business departments have historically viewed the world through different lenses. Unsurprisingly, challenges and potential solutions often appear different from various vantage points. Issues surrounding software quality are no exception. Business and technology professionals alike must understand and react to issues affecting both intrinsic and extrinsic quality. As we discussed in Section 3, software quality components are hopelessly intertwined.

The definitional framework proposed in this paper accounts for this interdependence by describing issues affecting quality with equal emphasis. Tying the definition to required levels of quality creates a value-based view which is well aligned with stakeholder perspectives. Enabling stakeholders to evaluate and manage quality related challenges consistently across the enterprise will allow more effective collaboration towards common goals. This is the first step towards a centralized governance model where controls like credit limits could be established to better align the risk/reward decisions with organizational standards. Coalescing the expertise of business, technology and risk management resources maximizes effectiveness by creating a united front in the battle against technical debt.

REFERENCES

[1] Cunningham, W. 1992. The WyCash Portfolio Management System. *OOPSLA' 92 Experience Report*.

[2] Garvin, D. 1984. *What Does "Product Quality" Really Mean?*. Sloan Management Review, Fall 1984, 25-45.

[3] ISO. *ISO/IEL 9126-1:2001: Software Engineering – Product Quality. Part 1: Quality Model.* Geneva, Switzerland: International Organization for Standardization, 2001.

[4] Kitchenham, B. 1996. Software Quality: The Elusive Target. IEEE Software, January 1996, 12-21.