Technical Debt and the Reliability of Enterprise Software Systems: A Competing Risks Analysis

Narayan Ramasubbu
Chris F. Kemerer

University of Pittsburgh
February 2015

forthcoming in *Management Science*, 2015

Technical Debt and the Reliability of Enterprise Software Systems:
A Competing Risks Analysis

Abstract

Enterprise software systems are required to be highly reliable as they are central to the business operations of most firms. However, configuring and maintaining these systems can be highly complex, making it challenging to achieve high reliability. Resource constrained software teams facing business pressures can be tempted to take design shortcuts in order to deliver business functionality more quickly. These design shortcuts and other maintenance activities contribute to the accumulation of *technical debt*, that is, a buildup of software maintenance obligations that need to be addressed in the future. We model and empirically analyze the impact of technical debt on system reliability by utilizing a longitudinal dataset spanning the 10 year lifecycle of a commercial enterprise system deployed at 48 different client firms. We use a *competing risks analysis* approach to discern the interdependency between client and vendor maintenance activities. This allows us to assess the effect of both problematic client modifications (client errors) and software errors present in the vendor-supplied platform (vendor errors) on system failures.

We also examine the relative effects of *modular* and *architectural* maintenance activities undertaken by clients in order to analyze the dynamics of technical debt reduction. The results of our analysis first establish that technical debt decreases the reliability of enterprise systems. Second, modular maintenance targeted to reduce technical debt was about 53% more effective than architectural maintenance in reducing the probability of a system failure due to client errors, but had the side-effect of increasing the chance of a system failure due to vendor errors by about 83% more than did architectural maintenance activities. Using our empirical results we illustrate how firms could evaluate their business risk exposure due to technical debt accumulation in their enterprise systems, and assess the estimated net effects, both positive and negative, of a range of software maintenance practices. Finally, we discuss implications for research in measuring and managing technical debt in enterprise systems.

Keywords: technical debt, enterprise systems, software reliability, architectural maintenance, modular maintenance, software maintenance, software product management, software package customization, competing risks modeling, Integrated Development Environment (IDE) toolkits, software risks, software complexity, Enterprise Resource Planning (ERP) systems, Commercial Off The Shelf (COTS) software.

Technical Debt and the Reliability of Enterprise Software Systems:
A Competing Risks Analysis

1. Introduction

Enterprise software systems, such as Enterprise Resource Planning (ERP) systems, are central to the business operations of a wide variety of firms, and constitute one of the biggest spending categories in their IT budgets (Hitt *et al.* 2002). Worldwide spending on enterprise software systems is estimated to exceed 300 billion U.S. Dollars, with a forecasted growth of about 6% in the next few years (Gartner 2013). Enterprise software systems are required to be highly reliable, as system failures could cause severe disruptions to the business processes that they support (Wu *et al.* 2013). However, prior research shows that achieving this high reliability is often difficult, in part due to the challenges involved in implementing and maintaining these systems over a long planning horizon amidst increasing business pressures that mandate shortened cycle times for software projects (Banker *et al*. 1998; Banker and Slaughter 2000; Masini and Wassenhove 2009).

Software teams facing such challenges may, in part, resort to design shortcuts to rapidly deliver the functionality demanded by business and thereby trade off the potential longer-term benefits of appropriate software design investments for these short-term business benefits (Austin 2001; Woodard *et al.* 2013)[1]. By taking design shortcuts and other maintenance activities software organizations incur what has been referred to as *technical debt*, that is, accumulated maintenance obligations that must be addressed in the future (Brown *et al.* 2010; Woodard *et al.* 2013; Ramasubbu and Kemerer 2014). For example, engineers facing time pressures to deliver functionality may hardcode text strings, saving the additional design efforts needed for properly externalizing text fragments from software code. However, such shortcuts, while expedient, tend to create long-term problems. For example, the well-known "Y2K" problem was the result of a coding practice that was adequate in the short run, but created a long-term maintenance obligation when all of the two digit date fields had to eventually be remediated to accommodate the change in the century. Enterprise systems laden with high levels of technical debt are not only prone to failures, but also create hindrances for the successful execution of business strategies (Woodard *et al.* 2013). Technical debt accumulation, combined with business pressures and limited resources, can place software teams in a downward cycle of taking design shortcuts, leading to an increase in system failures, leading to more expensive and difficult maintenance efforts, which may, in turn, lead to more design shortcuts. Thus, it could be expected that reducing the amount of technical debt

[1] Customizations at the client site may be based on requirements or functionality not available in the COTS software. These customizations may not always be possible to accommodate within the confines of the vendor prescribed design and development standards. These functional changes associated with customizations are likely to cause compatibility issues with the vendor's native software, irrespective of how these changes are made, including, but not limited to, design shortcuts.

over an enterprise system's evolution would confer business benefits by improving *system reliability*, that is, the extent to which a system is resilient to failures (Ramdas and Randall 2008; Guajardo *et al.* 2012).

Technical debt reduction in an enterprise software systems environment is difficult, as these applications are often built on an underlying vendor-supplied, commercial-off-the-shelf (COTS) platform which is customized by clients for their use (Basili and Boehm 2001; Chellapa *et al.* 2010; Subramanyam *et al.* 2011). Maintenance of such systems is especially challenging because of the interdependencies and potential for conflict between the underlying, vendor-supplied platform and the customizations done by individual clients (Masini and Wassenhove 2009; Ceccagnoli *et al.* 2012; Huang *et al.* 2013). For example, software update patches supplied by vendors as part of their maintenance plans could be incompatible with the customizations implemented by clients. Similarly, design shortcuts taken by clients could damage the overall software architecture of the underlying platform and disrupt maintenance plans offered by vendors (MacCormack and Verganti 2003; MacCormack *et al.* 2008). These interdependencies make it difficult to measure and assess the impact of technical debt on system reliability and therefore to plan the software maintenance activities necessary to reduce the debt. As a consequence of this complexity and the longitudinal nature of the required data there is only a very limited set of empirical studies of technical debt despite the business value in rigorously assessing the economic payoff from software maintenance activities in the presence of interdependencies between vendor-supplied COTS software and client-driven customizations.

This study addresses this need by analyzing a longitudinal dataset spanning the10 year lifecycle of a COTS-based enterprise software system deployed at 48 different client firms. We address the challenging research need of accounting for the interdependencies between client and vendor maintenance of COTS-based enterprise systems by distinguishing system failures due to software errors that originate in the modifications made by a client (*client errors*) from errors that are residual in the platform supplied by the vendor (*vendor errors*). Then, utilizing a *competing risks analysis* empirical approach, we assessed the impact of technical debt on system failures due to these client and vendor errors (Fine and Gray 1999; Pintilie 2007; Zhou *et al.* 2012; Kuk and Varadhan 2013). In our analysis an important focus was on the relative effects of *modular* and *architectural* design maintenance activities. Modular design maintenance activity focuses on improvements in individual modules that do not alter the interdependencies among the modules of a system. In contrast, maintenance activities targeting architectural design improvements focus on the interdependencies among the modules and the overall structure of a system (Henderson and Clark 1990; MacCormack *et al.* 2006, 2008). Utilizing the 10 year lifecycle data we empirically assess the impact of both technical debt and the modular and architectural maintenance activities undertaken by the 48 client firms on system reliability (i.e., the probability of a system failure).

The results of our analysis first establish that technical debt is associated with an increase in the probability of a system failure (a decrease in reliability). Next, we find that modular maintenance activities undertaken by the client firms were 53% more effective in reducing the chance of a system failure due to client errors than were architectural maintenance activities. However, modular maintenance undertaken by client firms also increased, by about four fold, the chance of a system failure due to vendor errors present in the underlying COTS platform. In contrast, architectural maintenance undertaken by clients, although less effective in improving the reliability of enterprise systems with respect to client errors, had only half the chance of causing a system failure due to vendor errors than did modular maintenance. These results establish the interdependency between client and vendor maintenance activities in COTS-based enterprise systems, and illustrate the dynamics of technical debt reduction and its effect on the reliability of these systems.

This study makes four primary research and practical contributions in advancing the software maintenance research literature on technical debt. First, most software maintenance research studies have focused on single site, bespoke systems implementations, and researchers have called for investigations of multi-site implementations of packaged enterprise software, given their increased practical importance (e.g., Williams and Pollock 2012). And to understand the full implications of enterprise solutions for organizations scholars highlight the need to examine software implementations over multiple use settings and time. Our study addresses these needs by analyzing the implementations of a COTS-based enterprise system at 48 different client sites over a ten year period. Second, rigorous empirical analysis of the reliability of enterprise software systems has generally been limited in prior research due to the lack of fine-grained and reliable data on software failures. Our ten year longitudinal dataset allows us to empirically measure the technical debt accumulated in real world enterprise software and to assess its dynamic impact on system reliability. Third, due in part to methodological challenges, prior software maintenance studies did not delineate risks arising from the different aspects of an enterprise software design and its implementation context, which prevented the accurate assessment of the impacts of technical debt on reliability of software systems. We address this challenge by utilizing a competing risks analysis approach that enables us to account for event-specific hazards that impact the failure of enterprise software systems. Finally, we utilize our empirical results to illustrate how firms could evaluate their business risks exposure due to technical debt accumulation in their enterprise systems and assess the likely effects, both positive and negative, of a range of software maintenance practices.

The rest of the paper is organized as follows. In §2 we provide a theoretical background for client-vendor interdependence in COTS-based software product development, and develop our hypotheses by utilizing relevant product development, technical debt, and software engineering economics research literatures. We also describe our conceptual model. In §3 we enumerate our data

collection procedures and explain the competing risks analysis methodology in §4. We present the results of our analysis in §5 and discuss implications for research and practice in §6. §7 provides a brief summary and conclusion.

2. Theoretical Background and Hypotheses

We begin by introducing the product development literature that explains how COTS vendors address client demand heterogeneity by providing a toolkit approach to software product development. Then, we elaborate how maintenance done by clients using the vendor-supplied toolkits could result in technical debt accumulation in COTS-based enterprise systems, which introduces interdependencies between the vendor and client product development tasks. Building on this background, and on empirical results of prior software maintenance studies, we develop our hypotheses relating technical debt, software maintenance, and the reliability of enterprise systems.

2.1. Demand Heterogeneity and Technical Debt in COTS Product Development

In a COTS-based software development model client firms purchase a software package that typically consists of a pre-configured version of the software and the necessary tools for further client-specific customization and development (Masini and Wassenhove 2009; Subramanyam *et al.* 2011; Ceccagnoli *et al.* 2012; Huang *et al.* 2013). Such a *toolkit* approach to product development has been attributed by von Hippel and Katz to a vendor strategy of addressing heterogeneous customer demands by shifting the costs and risks associated with specific needs-related product development to users (von Hippel and Katz 2002). They have proposed that the toolkit approach aims to partition product development and ongoing maintenance tasks into two different categories of subtasks, each owned independently by the COTS vendor and client firms respectively, and that such a partition empowers users to learn and innovate through iterative trial-and-error activities without the need for sole reliance on the vendor. Supporting this proposition, case studies of both manufacturing-related product development and open source software products have shown that the toolkit approach tends to lower overall costs and to improve both product development and maintenance productivity for users (von Hippel 1998; Franke and von Hippel 2003; Franke and Piller 2004).

However, there are two important aspects of the toolkit approach in COTS-based software development that warrant further investigation. First, it is not clear if ongoing product development and maintenance activities for a product-like COTS software that evolves continuously during its lifecycle can be fully partitioned into *independent* client and vendor subtasks (Barry *et al.* 2006). The evolution of the vendor-supplied software package and its associated toolkit has an impact on a client's development and maintenance activities by either constraining or expanding the design space available for specific needs-

related customizations (Baldwin *et al.* 2006). Similarly, needs-related customizations of a software package enacted by clients can also have an impact on the product development decisions of the vendor (MacCormack *et al.* 2001; Woodard *et al.* 2013). For example, a vendor might incorporate a popular modification enacted by several clients into its standard release of the software (Ethiraj *et al.* 2012). Or, perhaps more rarely, a vendor could also delete functionality from its standard release of the software package if it was incompatible with the existing infrastructure of several of its important clients (Deelstra *et al.* 2005; Ethiraj *et al.* 2012). Thus, the likely *interdependence* between the ongoing product maintenance activities of a vendor and its clients needs to be carefully considered while assessing the performance of COTS-based enterprise software systems.

Second, the manner in which a client uses the toolkit supplied by a vendor for ongoing product development and maintenance activities needs to be examined when assessing the performance outcomes of COTS-based enterprise systems. The toolkit supplied by the vendor enables clients to configure and customize the systems to suit their specific needs, as well as to modify the source code to add functionality that is not offered by the standard vendor product releases. However, it is well-documented that resource-constrained software teams tend to take design shortcuts in enacting these enterprise system modifications by violating vendor-specified standards, trading off longer-term benefits of appropriate design investments for short-term business benefits, such as quicker roll-out of the functionality (Austin 2001; Brown *et al.* 2010; Woodard *et al.* 2013). These design shortcuts tend to result in the accumulation of technical debt in enterprise systems. This technical debt, like financial debt, accumulates "interest" over time, which is to say that the costs of fixing design shortcuts to reduce enterprise systems technical debt increase with time (Brown *et al.* 2010; Curtis *et al.* 2012; Ramasubbu and Kemerer 2014). If not addressed through appropriate maintenance, technical debt severely impacts the evolution of an enterprise system by making it progressively harder for software teams to add new functionality and to rectify software errors (Ramasubbu and Kemerer 2014). Furthermore, in a COTS-based enterprise system the accumulation of technical debt by a client is generally not visible to a vendor, adding further complexity to the interdependence between the vendor and client tasks.

In this study we extend the examination of the toolkit approach to software product development by focusing on the interdependence between the vendor and client maintenance activities, and by accounting for the technical debt existing in enterprise software systems. Specifically, we distinguish system failures due to *client errors* and system failures due to *vendor errors*. Client errors are the software errors that originate from the client-induced modifications of a COTS-based enterprise system. Client errors occur because of problems associated with customizations, for example developing use-specific functionality through design shortcuts that violate vendor-specified programming standards (Ethiraj *et al.* 2012; Woodard *et al.* 2013). Such violations cannot be monitored and prevented by the vendor because the

toolkits through which the source code modifications are performed are locally installed and used by clients. In addition to client errors, the reliability of COTS-based enterprise systems is also affected by residual errors in the vendor-supplied software. It is well-documented that software vendors necessarily optimize software product testing and verification activities on a variety of factors which prevents them from achieving zero defects through quality control (Slaughter *et al.* 1998; Kemerer and Paulk 2009). As a result, COTS software products released by vendors often contain residual errors, which we term as *vendor errors*. By distinguishing systems failures due to client errors and those due to vendor errors we propose to study the distinct effects of technical debt and maintenance activities on the different types of systems failures. In the following sections we explicate the relationships among these constructs and develop the logic leading to our hypotheses.

2.2. Technical Debt and System Reliability

As explained above technical debt arises from design shortcuts and non-standard system changes undertaken by client software teams. Resource-constrained software teams under pressure to quickly implement functionality requested by business users may adopt expedient "quick and dirty" methods in lieu of methods recommended by vendors (Brown *et al.* 2010; Woodard *et al.* 2013). Such system changes increase the interdependencies between the different functions within a module and across the various modules of a system, and therefore increase the overall structural complexity of the system (Austin 2001; Ethiraj and Levinthal 2004). Software that is structurally more complex has been shown to be more error-prone, as higher levels of structural complexity hinder program comprehension and cause sub-optimal allocation of maintenance tasks in software teams (Banker and Slaughter 2000; Darcy *et al.* 2005; Kemerer and Darcy 2005, Ramasubbu *et al.* 2012). Higher levels of technical debt can also cause unpredictable ripple effects within a system, as errors propagate through the myriad interconnections between the system's various modules, often causing maintenance activities to be less effective relative to what they might have been (Kemerer 1995; Banker *et al.* 1998). Over time this contributes to software entropy, where the code "decays", that is, deteriorates in quality, causing unpredictable system failures (MacCormack *et al.* 2006; Eick *et al.* 2001).

Technical debt also increases the knowledge asymmetry between COTS vendors and their clients. Vendors possess knowledge unique to the development and maintenance of their toolkits, whereas clients possess knowledge related to their specific use cases (von Hippel and Katz 2002). Such unique knowledge possessed by both the vendors and the clients is often "sticky," that is, transfer of the knowledge between these parties is costly, which tends to contribute to knowledge asymmetry between vendors and clients (Ko *et al.* 2005; Esteves and Bohorquez 2007). Accumulation of technical debt by clients and the resulting increase in structural complexity increase the differences between client-modified

versions of the enterprise system and the standard configuration of the system supported by the vendor. Combined with the "stickiness" of vendor and client knowledge the resulting ineffective knowledge transfer over the lifecycle of a COTS-based enterprise system hampers the timely adoption of vendor-provided problem solutions, best practices, and design standards that could otherwise improve performance of the systems at client installations (Slaughter and Kirsch 2006; Woodard *et al.* 2013).

Thus, technical debt accumulation can cause software teams to be in a continuous downward cycle of taking design shortcuts, leading to increased structural complexity, leading to more expensive and difficult maintenance efforts, which may, in turn, lead to more technical debt accumulation and new design shortcuts. Therefore, all else being equal, we expect enterprise systems with higher levels of technical debt to be more error prone and less reliable, which leads to our first set of hypotheses:

Hypothesis 1: **Technical Debt and Enterprise System Failures**

H1a: A higher level of technical debt in an enterprise software system is associated with a higher probability of a system failure due to client errors.

H1b: A higher level of technical debt in an enterprise software system is associated with a higher probability of a system failure due to vendor errors.

2.3. Technical Debt Reduction through Modular and Architectural Software Maintenance

Software maintenance activities to reduce technical debt in enterprise systems focus on rolling back the shortcuts taken by software teams and reducing the resulting structural complexity (Marinescu 2012; Ramasubbu and Kemerer 2014). Such maintenance activities could be categorized as either addressing software design *within* the modules of a system (modular) or *across* several modules of the system (architectural). *Modular maintenance activities* to reduce technical debt include fixing flaws within a module (such as increasing cohesion among functions and methods within a module), altering complex string manipulation functions, and removing unreferenced methods, hard-coded business rules and direct usage of database tables (Curtis *et al.* 2012). In contrast, *architectural maintenance activities* to reduce technical debt focus on improving system-level modularity by simplifying the interconnections between the modules of a system (e.g., decreasing unnecessary coupling) and thereby minimizing the ripple effects due to modular errors (Banker *et al.* 1998; MacCormack *et al.* 2006).

While prior software maintenance studies have shown that maintenance activities undertaken to reduce structural complexity of a system improve software quality (Subramanyam and Krishnan 2003; Subramanyam *et al.* 2011), the relative efficacies of modular and architectural maintenance activities in the presence of technical debt remains an open empirical question. Previous research in product development (e.g., Henderson and Clark 1990) has examined modular and architectural product development activities to assess the performance of innovation efforts of organizations. Examining the relative efficacies of modular and architectural software maintenance activities in COTS-based enterprise system maintenance is needed as it would aid in furthering our understanding of the dynamics involved in

technical debt reduction, and could help firms to organize software maintenance activities more efficiently (Curtis *et al.* 2012). Client software maintenance activities specifically target client-induced modifications, and since clients possess the knowledge related to their specific use-cases they are well-positioned to identify erroneous business logic, conduct root-cause analysis, and perform preventive maintenance (Garg *et al.* 1998). However, client-induced system modifications, especially those enacted by taking design shortcuts, might also alter system architecture. Client software teams tend to have limited knowledge and resources to roll back such architectural changes without error (Woodard *et al.* 2013). This is because details about the architectural aspects of COTS systems, such as interfaces with database schemas, global views of cross-module method calls, and other system-level configurations, are typically not fully disclosed by vendors to clients, making it challenging for client software teams to reduce technical debt by revamping system architectures (Davenport 1998; Masini and Wassenhove 2005; Woodard *et al.* 2013). Therefore, we expect modular maintenance of clients to be more effective in reducing system failures due to client errors than will architectural maintenance, which leads to our second set of hypotheses:

Hypothesis 2: **Software Maintenance and Client Errors**

H2a: Modular maintenance will be associated with a lower probability of a system failure due to client errors.

H2b: Architectural maintenance will be associated with a lower probability of a system failure due to client errors.

H2c: Modular maintenance will be associated with a lower probability of a system failure due to client errors than will architectural maintenance.

Although both modular and architectural maintenance to reduce technical debt can be expected to help reduce client errors (H2a and H2b), in the presence of knowledge asymmetry there is an increased chance that these maintenance activities targeted at reducing client errors in a system could introduce *new* hazards by activating vendor errors that were dormant in the system (Parnas 1994; Reiss 2006; Kapur *et al.* 2011). Case studies of long-term enterprise systems usage have documented how client-driven maintenance activities could be in conflict with the vendor's planned product maintenance (Vogt 2002; Woodard *et al.* 2013). The chances of such conflicts increase when the enterprise system continues to evolve during its entire lifecycle and when the vendor and client planning horizons for the system are divergent (Barry *et al*. 2006). It is common for vendors of enterprise systems not to reveal development and maintenance roadmaps that cover the entire lifecycle of a product, but only to disclose activities pertaining to shorter time periods through, for example, annual release notes (Huang *et al.* 2013; Ceccagnoli *et al.* 2012). This gives enterprise software product vendors the necessary flexibility to adapt to new technological and market conditions, including extending the lifecycle of a product, or even abandoning a product in favor of a new technology (Greenstein and Wade 1998; MacCormack *et al.* 2001; Gjerde *et al.* 2002).

However, the limited release of information on the vendor's product development and maintenance roadmap often causes clients to address product limitations through their own self-driven activities (von Hippel 1998; Andersen and Morch 2009). Such client-initiated maintenance activities in the presence of knowledge asymmetry and technical debt increase the chance of conflicts with newer versions of components released by the vendor, such as new APIs, database schema alterations, and user interface specifications (Woodard *et al.* 2013). Therefore, we could expect both modular and architectural maintenance undertaken by clients in order to reduce technical debt to increase the probability of system failure due to vendor errors. While architectural maintenance may not be as effective as modular maintenance in reducing client-specific technical debt (hypothesis H2c), it is expected to have a lower chance of accidentally triggering any dormant vendor errors. This is because, unlike architectural maintenance, modular maintenance activities initiated by clients do not typically take into consideration system-level design parameters and interrelationships between and among the different modules of a system. Thus, the chance of conflicts between clients' modular technical debt reduction and vendor-driven platform-level changes is expected to be particularly high. Even though knowledge asymmetry between the vendor and client teams renders any client-driven maintenance activities to be less effective in preventing system failures due to vendor errors, architectural maintenance activities provide better opportunities for client teams to identify potential conflicts with vendor-driven platform changes. Thus, we expect client modular technical debt reduction to be associated with a higher chance of system failures due to vendor errors than will clients' architectural maintenance. The above effects are captured in the following set of hypotheses:

Hypothesis 3: **Software Maintenance and Vendor Errors**

H3a Modular maintenance will be associated with a higher probability of a system failure due to vendor errors.

H3b Architectural maintenance will be associated with a higher probability of a system failure due to vendor errors.

H3c: Modular maintenance will be associated with a higher probability of a system failure due to vendor errors than will architectural maintenance.

All of these hypothesized effects are summarized in Table 1.

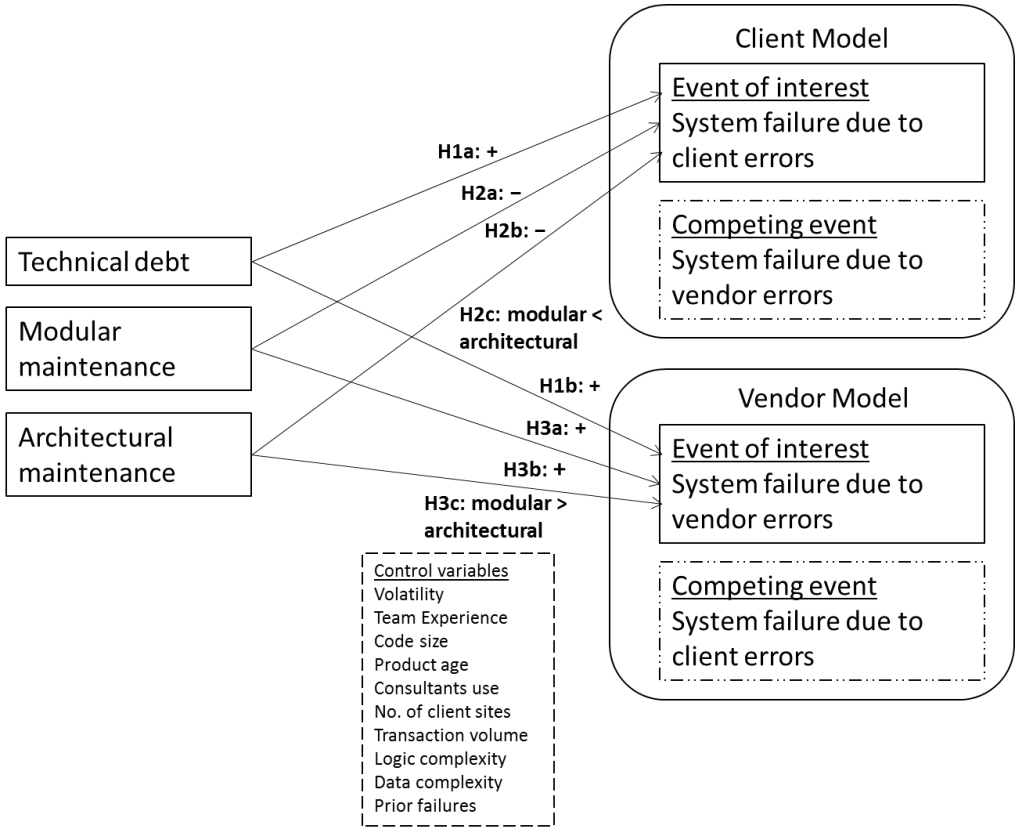
Table 1. Technical Debt Reduction and System Failures

	Probability of a system failure due to **Client** errors	Probability of a system failure due to **Vendor** errors
Technical debt	H1a: +	H1b: +
Modular maintenance	H2a: —	H3a: +
Architectural maintenance	H2b: —	H3b: +
Relative effects of modular vs. architectural maintenance	H2c: modular < architectural	H3c: modular > architectural

2.4. Conceptual Model

Our conceptual model of the competing risks analysis approach is shown in Figure 1. The Client Model in Figure 1 shows the conceptualized effects of technical debt, modular maintenance, and architectural maintenance on the probability of a system failure due to *client* errors, and portrays a system failure caused by vendor errors as a competing event. The Vendor Model in Figure 1 captures the effect of the independent variables on the chance of a system failure due to *vendor* errors as the primary focus, and treats a system failure due to client errors as a competing event. Figure 1 also shows that we control for covariates other than the main independent variables, and these covariates are described and discussed further in §3.4. Overall, our conceptualization enables the accounting of both event-specific hazards and the analysis of distinct effects of covariates on system failures.

Figure 1. Conceptual Model



3. Research Site and Data Collection

We collected data pertaining to the lifecycle of a COTS enterprise resource planning (ERP) software package from a leading multinational vendor and its clients. The package was first released by the vendor in 2002 and was in continuous operation until 2012 when it was permanently retired in favor of a new product. The package consisted of a preconfigured version of the application and an integrated

development environment (IDE), which was a toolkit that clients could use to modify the preconfigured version. The vendor-supplied IDE supported a proprietary programming language similar to C++ and a variety of relational databases. The vendor also maintained a client resource website, which had extensive training materials for learning the programming language supported by the toolkit. Throughout the ten year lifecycle of the product the vendor added functionality to both the preconfigured application and to the toolkit. The vendor also took responsibility for the maintenance of the preconfigured software package and toolkit, and provided regular updates to fix their residual errors.

We began our data collection efforts in cooperation with the product management division of the vendor. We collected archival data on the evolution of the product, including the source code pertaining to all versions of the product released during its ten year lifecycle. For each release of a new version of the product in the ten year time period we collected the corresponding release notes, which contained detailed information on the product roadmap, corrected vendor errors, and the anticipated rollout dates of the next generation of functionality.

The vendor maintained a database of all clients who had purchased the product and a separate team, called the installed-base support team, was in charge of post-sale customer support and maintenance activities. Clients (typically Fortune 500 firms) purchased licenses from the vendor to install the software package in their premises. Clients used the toolkit to view and edit source code of the ERP software package, alter certain system-level configurations, modify business logic embedded in the source code, and alter a section of the database tables. All modifications done by clients were system labeled using unique identifiers and timestamps. All client-modified modules were automatically disabled from receiving updates from the vendor and had to be manually patched by the clients. Typically, the regular maintenance updates from the vendor were communicated through a portal called the Customer Support Network (CSN), and designated client administrator personnel were automatically notified when new vendor-supplied software updates were available for installation.

Through the CSN channel we contacted the client firms to gather data on their specific installations of the product. Initially, out of the total universe of 117 firms in the vendor's client database, 75 client firms responded to our request to study their implementations. However, given the data demands of the research we were limited to gathering the full set of data for this study from 48 customer installations. We checked for any potential bias in client participation by comparing the participant and non-participant client firms for their size, employee count, product versions, and sales revenues, and did not record any significant differences. Similar to the archival data we collected from the vendor, we collected source code from all the 48 client firms and stored them in a version control database. We then contacted the individual project managers in the client firms to collect data on the software personnel who

worked on the client installations. In some cases the human resource managers of the client firms also provided additional data on the experience level of software teams in the firms.

Once we had the archival data in place we processed it in three stages. In the first stage we performed a series of sequence and phase analyses to track the evolution of the vendor-supplied product and the 48 variants at the client firms (Kemerer and Slaughter 1999). We broke down the ten year lifecycle of the product into 120 phases (12 months x 10 years) and tracked changes to the software within and across these phases. Through this we were able to map the growth of functionality in the vendor and client versions of the system for the entire ten year lifecycle. The sequence and phase analysis also helped us derive the software volatility metric from the changes we observed in the distinct phases of the product's lifecycle. We utilized software mining and version control tools[2] provided by the vendor to compare the source code pertaining to the different versions of the product at different time intervals. After the source code modifications were identified in client systems we assessed whether the client-initiated modifications detrimentally altered structural complexity, and thereby contributed to technical debt. Following the best practices prescribed in the literature (Curtis *et al.* 2012; Marinescu 2012), we identified these detrimental modifications by assessing whether the alterations violated vendor-prescribed design and development standards. For example, if a source code modification enacted direct database access or complex method calls without utilizing prescribed APIs, it was ascribed as contributing to the buildup of technical debt. Through the unique labels and timestamps associated with the source code modifications in client versions we were able to trace and cluster changes with the same change id and timestamp to identify the groups of modules and their interfaces which were impacted by a specific client-initiated change. This helped us to identify whether a particular client-initiated change was primarily modular maintenance or architectural maintenance[3].

In the second stage of data preparation we traced all system failures reported by client firms and mapped the communication logs present in the CSN system with the product version control logs supplied by the clients. For each system failure we marked the origin point of the error in the corresponding source code version, and categorized system failures as either client errors or vendor errors[4]. We further corroborated the identified cause of errors with the quality control and post-mortem reports that were logged in the CSN system.

---

[2] Additional details on the change sequence analysis and software code mining process are presented in §A1 of the online Appendix for the interested reader.

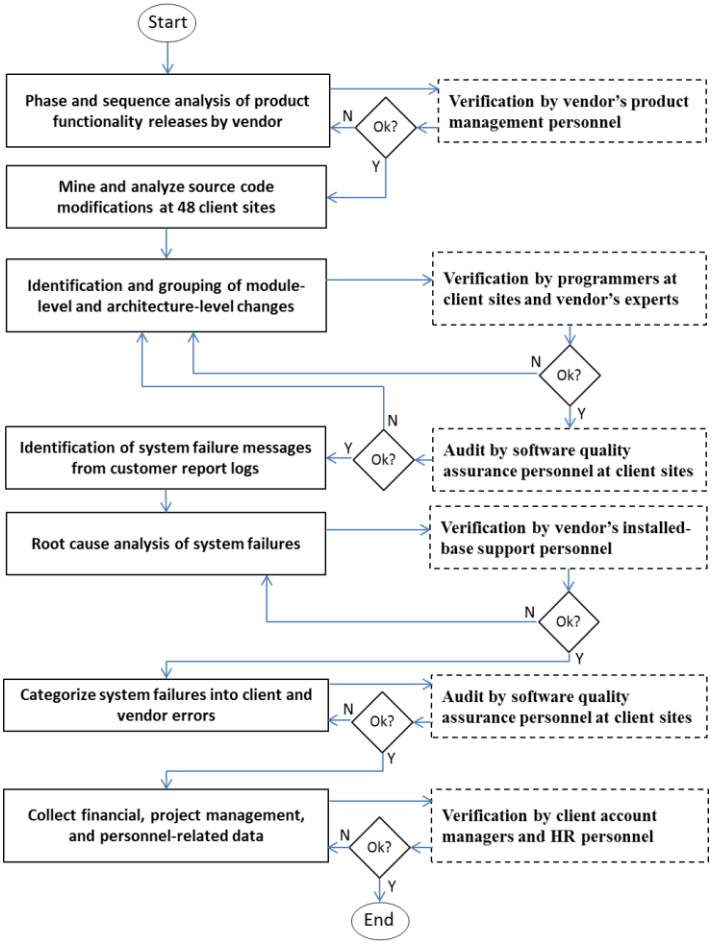
[3] We identified clusters of source code changes that happened at a given time (evolutionary coupling) and then compared evolutionary coupling sequences with the actual modular dependencies (structural coupling) to identify whether a particular source code change was primarily related to a modular or architectural maintenance activity. Additional details are presented in §A1 of the online Appendix. See also Figure 2 for an overview of the verification process.

[4] Additional details on the cause-effect analysis for categorizing client and vendor errors are presented in §A1 of the online Appendix. Also see Figure 2 for an overview of the verification process.

In the final stage of our data preparation we engaged with the product management and programming personnel at both the vendor and the client sites to verify our coding of the different facets of the archival data. Programmers helped us verify whether our coding of key customizations in source code was categorized correctly as adhering to the vendor-prescribed design standards or not. We utilized the client software quality assurance personnel to audit our archival data and to verify if the maintenance activities we identified for a given time period correctly matched with the internal project management records. Finally, product management personnel at the vendor site verified the mapping of sequence and phase analysis we derived from source code changes with the evolutionary milestones of the product as recorded in the internal product release notes.

The above data collection and verification steps are summarized in Figure 2. An overview of the dataset structure and a walkthrough example from the dataset illustrating the evolution of an application at a client site, technical debt accumulation in the system, the modular and architectural maintenance undertaken by the client, and the impact on system failures are presented in §A2 and §A3 of the online Appendix.

Figure 2. Data Collection and Verification Steps



3.1. Constructs and Measurement

3.2. Dependent Variable: System Reliability

The dependent variable of interest in our analysis is the reliability of an enterprise system, which is measured as the extent to which the system is resilient to failures[5]. Our dataset captures all of the system failure incidences at 48 client firms during the ten year observation period of this study. To assess a system's resilience to failure we calculate the time interval (hours) between subsequent incidences of system failures at a client site, and utilize this time-to-failure variable for survival analysis (Ramdas and Randall 2008; Guajardo *et al.* 2012). The specific competing risk analysis approach we utilize for modeling the failure of an enterprise system is further elaborated in §4.

3.3. Independent Variables

3.3.1. Technical Debt

We measure the technical debt accumulated in a client firm's enterprise system by tracking client-driven modifications of the COTS systems that violated design standards prescribed by the COTS vendor (Curtis *et al.* 2012; Woodard *et al*. 2013). Specifically, we tracked all source code modifications undertaken by a client in the ten year lifecycle of their system, and derived three variables to capture the accumulation of technical debt. First, we calculated the percentage of client-driven source code modifications that did not adhere to vendor-prescribed standards to capture the overall extent of problematic alterations undertaken by a client. Then, we categorized these customizations into modifications that altered business logic and those that altered the vendor-provided data schemas. Finally, for all newly client-added source code, we examined the functional calls and database access statements in the source code to identify the incidences of non-use of application programming interfaces (APIs) prescribed by a vendor, and derived the total number of API violations variable. The three variables, the percentage of problematic customizations pertaining to business logic, percentage of data schema modifications that violated vendor's master data standards, and the number of API violations, together capture the overall extent of technical debt in a client's system by accounting for design shortcuts in newly added source code as well as alterations done to existing source code (Curtis *et al.* 2012; Marinescu 2012). As shown in Figure 2, all of these research measurements (and the ones described below) were validated by vendor and client technical staff.

[5] The term "system failure" here refers to the "critical" category of errors identified by the clients and vendor, resolutions of which were well documented and governed as per the client-vendor maintenance contracts. These failures caused business process disruptions which were considered critical by the clients. Therefore, we only consider what other software defect analyses might refer to as "highly serious" or "critical errors".

3.3.2. Modular Maintenance

To capture the extent of modular maintenance undertaken by a client in a time period we derived the percentage of total source code in modules that were examined by client teams during maintenance activities in that time period. This is similar to the code coverage software metric utilized during software verification that captures the ratio of total software test requirements satisfied by a given verification procedure (Amman and Offutt 2008).

3.3.3. Architectural Maintenance

To capture the extent of architectural maintenance activities undertaken by a client in a time period we derived the percentage of interfaces between modules that were examined during the maintenance period. This measure is similar to the interface coverage software metric utilized during module integration testing procedures that verify if the interfaces between modules in a given system communicate correctly (Jin and Offutt 2001).

3.4. Control Factors

In examining the hypothesized relationships between technical debt, maintenance activities, and enterprise system reliability we consider the following factors that have been identified in the literature as potential covariates: (1) source code size; (2) volatility; (3) team experience; (4) use of consultants as intermediaries; (5) product age, (6) number of client sites, (7) transaction volume, (8) logic and data complexity, and (10) number of prior failures.

3.4.1. Source code size

Source code size has been shown to be an important predictor of project outcomes such as productivity and quality, indicating the presence of scale economies in software development and maintenance projects (Banker and Kemerer 1989; Banker and Slaughter 1997). All else being equal software packages with a larger source code size tend to be more complex and have been reported to be associated with an increased number of errors (Kemerer 1995; Subramanyam and Krishnan 2003). Therefore, we include the source code size of the client's version of the software package as a control variable in our analysis.

3.4.2. Volatility

Prior software product development research has shown that volatility in environmental conditions significantly influences the rate of changes in systems, as well as the way those changes are enacted (Barry *et al.* 2006). From a vendor's perspective higher environmental volatility would likely increase heterogeneity in customer demands (Adner and Levinthal 2001). This increased demand heterogeneity would prompt a vendor to keep its standard release of functionality relatively minimized and allow clients to utilize the supplied toolkits to satisfy their unique demands. Thus, from a client perspective an increase in environmental volatility would tend to lead to an increase in software volatility,

that is, the magnitude and rate of changes enacted to the software system[6] (Barry *et al.* 2006). Since environmental volatility might not equally impact all clients of an enterprise system, it is possible that clients experience different levels of software volatility. The level of software volatility faced by a client is known to influence the extent to which the client uses designs that do not violate vendor-prescribed standards (Barry *et al.* 2006). At the same time, higher levels of software volatility have also been associated with lower levels of software quality, and have been identified as a key source of critical system failures (Banker and Slaughter 2000; Agarwal and Chari 2007). Therefore, it is important to account for software volatility while examining the effect of technical debt-reducing maintenance activities on enterprise system reliability. Furthermore, since the software volatility variable accounts for the magnitude of all system changes, we can be confident that the key independent variables measuring problematic client-initiated modifications (customizations and API violations) in the empirical models are appropriately capturing the effects of technical debt accumulated in a client's enterprise system.

3.4.3. Team experience

Empirical evidence from prior research shows that software teams that are stable and consisting of more experienced members in their respective roles perform better, all else being equal (Boh *et al.* 2007; Huckman *et al.* 2009). In particular, generational experience, that is, experience with multiple releases of the product over its lifecycle, has been associated with higher software productivity and quality (MacCormack *et al.* 2001). Furthermore, more experienced software teams have been found to be able to handle software volatility in a superior way by developing mature software processes that suit their environmental context (Harter *et al.* 2000; Barry *et al.* 2006). Thus, we anticipate that, all else being equal, client software teams with higher levels of experience will be associated with fewer enterprise system failures.

3.4.4. Use of consultants as intermediaries

It is a common practice for clients to use external consultants to implement and maintain their enterprise systems (Akkermans and van Helden 2002; Sarker and Lee 2003; Ramasubbu *et al.* 2008). Such external consultants, typically from firms that offer system integration services, play an intermediary role between the client software teams and the system vendor. The effects of using external consultants for enterprise system projects have been reported as mixed in the literature (Gable 1996). On one hand, external consultants have been noted as being able to bring specialized design knowledge that helps software teams in their problem solving activities (Nah *et al.* 2001; Akkermans and van Helden 2002). On the other hand, conflicts between client teams and external consultants in their problem solving approach and project priorities, and coordination challenges between these parties have been

[6] We calculated client and vendor software volatility variables separately by taking into account client-initiated and vendor-initiated changes respectively. However, the two variables were highly correlated ($p<0.001$ level) so we combined them into a single volatility variable to measure the total software volatility of a client's enterprise system.

identified as critical factors in impacting enterprise system failure rates (Holland and Light 1999; Finney and Corbett 2007). Since the presence of external consultants could potentially alter project dynamics and the flow of knowledge between the vendor and client teams, we control for this in our empirical models (Sarker and Lee 2003; Ko *et al.* 2005).

3.4.5. Product age

It has been documented that entropy in software products tends to increase with system age, causing an increase in system failure rates (MacCormack *et al*. 2006; Eick *et al.* 2001). To control for the overall effects of product aging, we include the time in months since the enterprise system was installed and became operational at a client system in our regression models.

3.4.6. Number of client sites

Firms that operate in multiple locations have been documented as partitioning their enterprise implementations across these sites (Davenport 1998; Umble *et al.* 2003). Case studies of enterprise system operations have noted differences in software maintenance practices between multi-site enterprise system operations and single-site operations, including the way systems are configured and the manner in which resources are allocated to software teams (Holland and Light 1999; Markus *et al.* 2000). Also, multi-site enterprise system operations have been reported to be complex and more prone to failures than single-site implementations (Finney and Corbett 2007). To account for these potential effects we include the number of client sites as a control variable in our models.

3.4.7. Transaction volume

The extent of customizations and maintenance of enterprise systems undertaken by clients and the eventual performance of these systems might be influenced by the extent to which clients use these systems (Devaraj and Kohli 2003). To account for the possible correlations between systems usage and maintenance, and the variance in system usage levels across the different clients, we included transaction volume as a control variable in our models[7]. A transaction in this context is defined by the vendor as the execution of program modules that accomplish specific business functionality. For example, "create invoice", "reconcile stock levels", and "create compliance report" are some commonly-used transactions in the software package. Similar in spirit to Function Points, the transactions metric captures the customer-focused functionality offered by the software package (Ramasubbu and Kemerer 2014). Transaction volume refers to the cumulative number of transactions executed in a client's system in an observation time period.

[7] We thank an anonymous referee for this suggestion.

3.4.8. Logic and Data complexity

System failures and the effectiveness of client-driven maintenance activities could be impacted by system module characteristics beyond those captured by source code size[8]. Prior research studies have shown that logic and data complexity of source code could have a significant impact on the effectiveness of software maintenance undertaken by clients (Kemerer 1995; Banker and Slaughter 2000). Accordingly, we include two control variables in our models to account specifically for the logic and data complexity of the client systems. Logic complexity was measured using McCabe's cyclomatic complexity metric (McCabe 1976) and data complexity was measured using the extended McCabe metric that quantifies the complexity of a module's structure as it relates to data-related variables, capturing the number of independent paths through data logic of a module (McCabe and Butler 1989).

3.4.9. Prior failures

Systems that have failed in the past might be particularly error prone in the future. To account for the effects of past failures of client systems, we included the cumulative number of prior system failures due to client errors and the cumulative number of prior system failures due to vendor errors as additional control variables in our regression models.

3.4.10. Summary of control factors

The above constructs and their measurement are summarized in Table 2. We found that total team experience and generational experience of the software team were highly correlated (Pearson's correlation of 0.93). Hence, we utilized only the total team experience measure in the empirical models. Similarly, the three collected measures of software size (KLOC, Function Points, and vendor-defined transactions) were highly correlated (Pearson's correlation of 0.87). Therefore, we included only the KLOC measure of code size in the regression models[9]. Furthermore, we checked if there was a need to include module-level dummy variables in the empirical models to account for module-level use variations across the different clients. We found that there were no significant differences across the 48 clients at the module-level granularity, but that there were significant differences at the level of transactions in terms of transaction volume and transactions variety (i.e., the number of unique transactions). As might be expected, transactions variety was highly correlated with code size, and hence we included transaction volume and code size as controls in the regression models[10]. The summary statistics and correlation between the variables are presented in Table 3.

[8] We thank the department editor and an anonymous referee for this suggestion.
[9] Regression results did not significantly vary with the choice of any of the alternative software size and team experience measures.
[10] We thank an anonymous referee for this suggestion.

Table 2. Constructs and Variables in Dataset

Variable	Measurement
Technical debt: Client customizations violating vendor standards (business logic)	% total source code modifications by a client that altered vendor-provided business logic and did not adhere to vendor-prescribed standards
Technical debt: Client customizations violating vendor standards (data schema)	% data schema modifications by a client that altered vendor-provided master data schema and did not adhere to vendor-prescribed standards
Technical debt: Client customizations that have API violations	The total number of violations for the non-use of application programming interfaces where applicable
Modular maintenance	% of source code in a module examined by client teams during the maintenance period (code coverage)
Architectural maintenance	% of interfaces between modules examined during the maintenance period
Volatility	% of the overall source code of the enterprise system that changed during the maintenance time period
Team experience	Average experience of the team in years
Code size	Size of the source code base measured in thousands of Lines Of Code (KLOC)
Product age	Time since system installation (in months)
Consultants use	Indicator variable coded =1 if consultants were ever used as an intermediary between the vendor and a client
Number of client sites	The number of geographical sites of a client firm in which distinct implementations of the enterprise system existed
Transaction volume	Total number of transactions completed in a client system during the observation period
Logic complexity	McCabe's cyclomatic complexity metric
Data complexity	McCabe's extended data complexity metric
Prior client error failures	Total number of system failures due to client errors before the current observation period
Prior vendor error failures	Total number of system failures due to vendor errors before the current observation period

4. Empirical Model

The event of interest in our analysis is a failure of enterprise systems at 48 firms in a 10 year observation period as a measure of their relative reliability. The failure data is right censored because systems at the 48 client firms evolved at different rates; for example, systems at the different client sites had different failure rates and maintenance schedules, and were eventually migrated or retired in different ways at different time periods[11]. Typically, such failures in censored data have been examined using time-to-event analysis through Cox's proportional hazard models (Ramdas and Randall 2008; Li *et al.* 2010). In such time-to-event analysis we would look at the time to failure (t), the risk set, the hazard function *h(t)*, and the cumulative incidence function (CIF) to assess the factors that contribute to the failure. However, the standard single event time model is limited in its ability to analyze COTS-based enterprise software system failures because the failure of such systems could be caused by two types of events, namely, (1) failures due to client errors, and (2) failures due to vendor errors. In other words, client errors and vendor errors *compete* for system failure. Thus, the standard survival analysis model needs to be extended for this competing risks scenario.

[11] An overview of the censored dataset structure and a walkthrough example from the dataset are presented in §A2 and §A3 of the online Appendix for the interested reader.

Table 3. Summary Statistics and Correlations

Variable		Mean	Std. Dev	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Customization business logic (%)	1	56.43	25.91	1.00															
Customization data schema (%)	2	53.29	24.2	0.43	1.00														
API violation (#)	3	43.39	24.02	0.25	0.32	1.00													
Module maintenance (%)	4	34.77	47.35	0.23	0.16	-0.13	1.00												
Architecture maintenance (%)	5	32.97	47.07	0.11	0.21	0.17	-0.15	1.00											
Volatility (%)	6	56.87	22.83	0.07	0.02	0.05	0.02	0.05	1.00										
Total team experience (yrs.)	7	9.70	2.16	-0.33	-0.32	-0.31	0.06	0.10	-0.26	1.00									
Code size (KLOC)	8	866.11	259.33	0.27	0.35	0.34	-0.09	-0.06	0.35	-0.29	1.00								
Product Age (months.)	9	55.85	33.45	0.31	0.23	0.28	-0.09	-0.07	0.25	-0.19	0.24	1.00							
Consultants (0/1)	10	0.33	0.48	0.29	0.28	0.21	-0.23	-0.09	0.14	-0.38	0.01	0.42	1.00						
Client sites (#)	11	2.73	1.14	0.11	0.06	0.04	0.15	-0.02	0.19	0.01	0.06	0.11	0.34	1.00					
Transaction volume (# 1000s)	12	62.14	20.53	0.33	0.27	0.31	0.23	0.24	0.16	0.42	0.15	0.11	0.29	0.17	1.00				
Logic complexity (#)	13	11.35	5.41	0.42	0.32	0.31	0.22	0.31	0.28	0.27	0.33	0.19	0.07	0.31	0.18	1.00			
Data complexity (#)	14	6.71	4.32	0.13	0.53	0.25	0.31	0.23	0.30	0.10	0.13	0.57	0.03	0.16	0.22	0.26	1.00		
Prior client error failures (#)	15	5.07	2.13	0.29	0.34	0.18	0.21	0.27	0.23	-0.18	0.12	0.26	-0.06	0.12	0.19	0.36	0.31	1.00	
Prior vendor error failures (#)	16	4.29	2.35	0.26	0.31	0.18	0.18	0.17	0.23	-0.21	0.32	0.28	-0.04	0.15	0.25	0.26	0.36	0.66	1.00

Note: For sample size n=3641, correlation coefficients > .03 or < -.03 are statistically significant at p < 0.05

One way to model the competing risks situation is to use separate cause-specific hazards to represent the enterprise system failure rate from vendor errors and client errors, each following a Cox proportional hazard model (Cox 1972). However, since an analysis of a cumulative incidence of system failures is a function of both of the cause-specific hazards (client errors and vendor errors), in such an approach it would not be possible to directly evaluate the effects of individual covariates, such as code size, on the *cumulative incidence* of system failures. To overcome this limitation Fine and Gray (1999) proposed a semiparametric approach to directly model the cumulative incidence function of competing risks using a subdistribution hazard function, which can be intuitively thought of as the hazard for a system that fails from a specific cause or does not, and in the latter case has an indefinite survival time from the specific cause. The Fine-Gray model takes into account competing events and does not make assumptions about their independence or censoring distribution. It facilitates testing for covariate effects on subdistribution hazards, and issues pertaining to model selection and efficient regression predictions could be easily addressed. Therefore, in line with the recommended best practice, in addition to the cause-specific hazard analysis (Cox model), we also use the Fine-Gray model to empirically assess the system failures in our data (Pintilie 2007; Lau *et al.* 2009).

Formally, each system in our data is associated with a tuple (T, D), where T is the time-to-failure and D is the type of event that captures the possible causes (k=1, 2), namely, client error-related failure

(k=1) and vendor error-related failure (k=2). The cause-specific hazard function for the competing risks is the hazard of failing from a given cause in the presence of the other competing event:

$$h_k(t) = \lim_{\Delta t \to 0} \{ \frac{P(t \leq T < t + \Delta t, D = k | T \geq t)}{\Delta t} \} \text{ with } D = 1, 2$$

The regression model on the cause-specific hazards (Cox model) with the inclusion of covariates represented by the vector Z is given as $h_k(t; Z) = h_{0k}(t)\, e^{\beta Z}$. The total hazard can be calculated as the sum of all k hazards, and the survival probability for the k^{th} risk, $S_k(t; Z)$, is interpreted as the survival probability for the k^{th} risk if all other risks were hypothetically removed. The cause-specific coefficient estimates for the covariates obtained from this regression model can be interpreted in the same way as hazard ratios in the absence of competing risks. However, as mentioned above, a limitation is that the covariate effects do not pertain to the cumulative incidence of system failures. The Fine-Gray model specification below overcomes this limitation.

In the Fine-Gray regression model the hazard associated with the Cumulative Incidence Function (CIF) is defined as $I_k(t;Z)$, which captures the covariate effects directly on the CIF through the subdistribution hazard function $h_k^*(t;Z)$. This subdistribution hazard is the probability of system failure due to a cause given that the system has survived up to time t without any failure, or has had a failure due to a competing cause prior to time t. This means that while analyzing a specific event of interest, the model does not censure failures that happen due to competing causes.

$$I_k(t;Z) = 1 - exp\{-\int_0^t h_k^*(u, Z) du\}$$

$$h_k^*(t;Z) = h_{0k}^*(t)\, e^{\beta Z}$$

$$h_k^*(t;Z) = \lim_{\Delta t \to 0} \{ \frac{P(t \leq T < t + \Delta t, D = k | T \geq t \cup (T < t \cap D \neq k)}{\Delta t} \}$$

Since $h_k^*(t;Z)$ is no longer the cause-specific hazard in the Fine-Gray model, the CIF for cause k depends not only on the hazard of a specific k, but also on the hazard for the other possible causes. Unlike the Cox model, the regression estimates of covariates from the Fine-Gray model directly relate to the CIF. The regression model specification with the entire list of covariates for our model is given as:

$$\begin{aligned} h_k^*(t; Z) = h_{0k}^*(t) * \exp\{ & \beta_1 (customization\ of\ business\ logic) \\ & + \beta_2 (customization\ of\ data\ schema) + \beta_3 (API\ violation) \\ & + \beta_4 (modular\ maintenance) + \beta_5 (architectural\ maintenance) \\ & + \beta_6 (volatility) + \beta_7 (team\ experience) + \beta_8 (code\ size) \\ & + \beta_9 (product\ age) + \beta_{10} (consultants\ use) \\ & + \beta_{11} (client\ sites) + \beta_{12} (transactions\ volume) + \beta_{13} (logic\ complexity) \\ & + \beta_{14} (data\ complexity) + \beta_{15} (prior\ client\ failures) \\ & + \beta_{16} (prior\ vendor\ failures)\} \end{aligned}$$

While estimating the above competing risks regression model we explicitly consider the potential dependence across the different versions of a system at a client site. We treat the different versions of a system at a client site as siblings in a family cluster that might be correlated due to unobserved shared factors. We utilize a recent extension of the Fine-Gray proportional hazard model to derive the estimators of the regression parameters specific to clustered data (Zhou *et al.* 2012). Furthermore, as an alternate approach for comparison, we also estimate the model by treating failures of a system at a client's site as ordered and repeated events using variance-correction and frailty models that specifically correct standard error of regression estimates for non-independence caused by repeated events (Box-Steffensmeier and Zorn 2002)[12].

5. Results

Tables 4 and 5 present the regression results obtained using the Fine-Gray model for clustered data and the different Cox models with adjusted standard errors using variance correction and frailty approaches. Estimates of models 1 and 2 presented in Table 4 are derived using the competing risks estimation approach, treating the 48 client firms as individual clusters. Models 3 and 4 in Table 5 are estimated using the latent survivor time approach, which involves estimating two different models for the client and vendor failures by resetting the survival data for each of these ways of system failures. In models 5 and 6 in Table 5, the estimates from the standard Cox model are adjusted for variance-correction to account for repeated events. These estimates were derived using the Prentice-Williams-Peterson (PWP) conditional risk-sets method estimated in gap time (Prentice *et al.* 1981). In the PWP models, an individual observation is not at risk for a later event until all prior events have occurred. Accordingly, the sample size for the estimation varies from the Fine-Gray and other Cox models because the risk set at time t for the k^{th} occurrence of a repeated event is limited to those observations that have already experienced k-1 events of that type. Models 7 and 8 in Table 5 are estimated using a frailty approach, which treats repeated events in a cluster as a special case of unit-level heterogeneity, specifically as random draws from a gamma distribution (Box-Steffensmeier *et al.* 2007).

In interpreting the hazard ratio estimates of the various models note that higher system reliability is associated with a lower hazard ratio (coefficient < 1) and lower system reliability is associated with a higher hazard ratio (coefficient > 1) since the actual measurements are of system failures, i.e., the lack of reliability. Referring to Tables 4 and 5 we see that the chi-square statistics for all models are significant at $p < 0.01$, indicating strong statistical support for the overall model selection. Tests of the proportional

[12] We thank an anonymous reviewer for suggesting this approach. We note that the estimation procedures for ordered, recurrent events under a competing risk model are complex and not readily available (Taylor and Pena 2013; Dauxois and Sencey 2009). The use of the Fine-Gray model for clustered data is appropriate for our data because membership in each of the clusters is dependent on the different versions of the software implemented at each client site over time (subject to censoring). For an individual member within each of the 48 clusters (i.e., a specific version of a client software), we do not observe repeated failures in the data.

hazard assumption of the regression models verified that the hazard ratios were constant over time, and therefore, stratification, or time-varying interaction terms, were not needed. We verified that the results were not influenced by outliers in the data by examining the distribution of martingale and deviance residuals of the regressions (Rogers 1994). Coefficient estimates from the various Cox models and the Fine-Gray models are in the same direction, increasing the confidence of our empirical results. Since the Fine-Gray model estimates directly relate to CIF, for ease of exposition we utilize the Table 4 Fine-Gray model estimates to explain the hypothesis-testing results.

Table 4. Clustered Data Fine-Gray Model Regression Estimates

		(1)	(2)
		Event of interest: System failure due to **client** errors Competing event: System failure due to **vendor** errors	Event of interest: System failure due to **vendor** errors Competing event: System failure due to **client** errors
Customization-business logic	β_1	1.14***	1.62***
Customization-data schema	β_2	1.12***	1.41***
API Violations	β_3	1.23***	1.98***
Modular maintenance	β_4	0.22***	3.98***
Architectural maintenance	β_5	0.47**	2.17***
Volatility	β_6	1.02***	1.08***
Team experience	β_7	0.62***	0.88***
Code size	β_8	1.42***	1.45***
Product age	β_9	1.05***	1.07***
Consultants use	β_{10}	0.96*	0.99*
Client sites	β_{11}	1.04	1.01
Transaction volume	β_{12}	1.01*	1.08*
Logic complexity	β_{13}	1.14***	1.15***
Data complexity	β_{14}	1.09**	1.11***
Prior client error failures	β_{15}	1.05***	1.04*
Prior vendor error failures	β_{16}	1.04***	2.19***
Model Chi-squared statistic		450.98***	448.35***
Variance explained (Pseudo R-squared)		0.38	0.28
Observations		3641	3641

Note: ***, **,* Hazard Ratios significant at $p < 0.01$, $p < 0.05$, $p < 0.1$ respectively

Table 5. Regression Estimates for Cox Models with Adjusted Standard Errors

		Cox latent failure models with robust standard errors		PWP repeated failures gap time models		Cox frailty models with robust standard errors	
		(3)	(4)	(5)	(6)	(7)	(8)
		System failure due to **client** errors	System failure due to **vendor** errors	System failure due to **client** errors	System failure due to **vendor** errors	System failure due to **client** errors	System failure due to **vendor** errors
Customization-business logic	β_1	1.21^{***}	1.78^{***}	1.46^{***}	1.63^{***}	1.11^{***}	1.98^{***}
Customization-data schema	β_2	1.19^{***}	1.19^{***}	1.24^{***}	1.52^{***}	1.05^{***}	1.50^{***}
API Violations	β_3	1.32^{***}	1.95^{***}	1.14^{***}	1.62^{***}	1.02^{***}	1.31^{***}
Modular maintenance	β_4	0.19^{***}	4.11^{***}	0.21^{***}	2.97^{***}	0.19^{***}	4.53^{***}
Architectural maintenance	β_5	0.38^{***}	2.63^{***}	0.42^{***}	1.36^{***}	0.39^{***}	2.38^{***}
Volatility	β_6	1.05^{***}	1.13^{***}	1.17^{***}	1.07^{***}	1.10^{***}	1.06^{***}
Team experience	β_7	0.51^{***}	0.85^{***}	0.48^{***}	0.89^{***}	0.43^{***}	0.80^{**}
Code size	β_8	1.51^{***}	1.29^{***}	1.30^{***}	1.05^{***}	1.49^{***}	1.06^{**}
Product age	β_9	1.02^{**}	1.08^{***}	1.01^{*}	1.11^{**}	1.01^{**}	1.07^{**}
Consultants use	β_{10}	0.96^{*}	0.99^{*}	0.95^{**}	0.97^{**}	0.84	0.97^{*}
Client sites	β_{11}	1.04	1.07	1.03^{*}	1.09	1.02^{*}	1.00
Transaction volume	β_{12}	1.01	1.03	1.03	1.02^{**}	1.01^{*}	1.01^{*}
Logic complexity	β_{13}	1.13^{***}	1.12^{***}	1.16^{***}	1.12^{***}	1.18^{***}	1.21^{***}
Data complexity	β_{14}	1.03^{**}	1.16^{***}	1.23^{***}	1.30^{***}	1.03^{**}	1.07^{**}
Prior client error failures	β_{15}	1.06^{**}	1.03^{*}	1.03^{**}	1.02^{*}	1.03^{**}	1.06^{**}
Prior vendor error failures	β_{16}	1.07^{***}	2.66^{***}	1.14^{***}	1.72^{**}	1.04^{**}	2.73^{**}
Model Chi-squared statistic		490.27^{***}	469.55^{***}	78.91^{***}	62.56^{***}	219.54^{***}	198.81^{***}
Frailty parameter						0.25^{***}	0.31^{***}
Variance explained (Pseudo R-squared)		38%	28%	31%	29%	37%	26%
Observations		3641	3641	386	382	3641	3641

Note: ***, **,* Hazard Ratios significant at $p < 0.01$, $p < 0.05$, $p < 0.1$ respectively

Referring to Table 4 we see that a 1% increase in non-standard customization of business logic increases the probability of a system failure due to client errors by 14% (Column 1, $\beta_1 = 1.14$; 14% above the baseline hazard of 1). Similarly, data schema customizations increase the chance of client-error system failures by 12%. Also, we see that a 1% increase in non-standard customizations significantly increases the probability of a system failure due to vendor errors (Column 2, $\beta_1 = 1.62$ for business logic and $\beta_2 = 1.41$ for data schema). Similarly, a 1% increase in API violations increases the chance of a system failure due to client errors by 23% (Column 3, $\beta_3 = 1.23$), and also increases the chance of a system failure due to vendor errors by about two times (Column 4, $\beta_3 = 1.98$). Taken together, these results show strong support for Hypotheses 1a and 1b that predicted technical debt to increase the

probability of a system failure due to client and vendor errors respectively. This result supports our postulation that technical debt results in poor reliability of systems because the source code modifications and API violations implemented by clients tend to be error-prone and to accentuate defects present in the vendor-supplied package by creating conflicts between client and vendor maintenance activities.

Hypothesis 2a predicted that modular maintenance would reduce the chance of a system failure due to client errors. Referring to Table 4, Column 1, we see that modular maintenance is associated with a 78% reduction in the probability of a system failure due to client errors ($\beta_4 = 0.22$, 78% lower than baseline hazard ratio of 1), lending strong support for Hypotheses 2a. We also find full support for Hypothesis 3a, which predicted modular maintenance to be associated with a higher probability of a system failure due to vendor errors. Our results show that a 1% increase in code coverage during modular maintenance to reduce technical debt increases the probability of a system failure due to vendor errors by about four times (Table 4, Column 2: $\beta_4 = 3.98$).

Our results indicate that architectural maintenance reduces the chance of a system failure due to client errors by about 53% (Table 4, Column 1: $\beta_5 = 0.47$, 53% lower than baseline hazard ratio of 1), which supports Hypothesis 2b. Also, as predicted by Hypothesis 3b, architectural maintenance increases the probability of a system failure due to vendor errors by about two times (Table 4, Column 2: $\beta_5 = 2.17$). These results are consistent with our postulation that knowledge asymmetries between the clients and vendor of a COTS-based enterprise system may lead to conflicts between the maintenance activities performed by the clients and the vendor, leading to poor system reliability.

Hypothesis 2c posited that modular maintenance would be more effective in reducing the probability of a system failure due to client errors than would architectural maintenance. The predicted hazard ratios of modular maintenance and architectural maintenance variables were significantly different (Table 4 Column 1: $\beta_4 = 0.22$ and $\beta_5 = 0.47$; test Chi-squared = 6.65, $p < 0.01$). We see that modular maintenance is 25% more effective in reducing the chance of a system failure due to client errors than is architectural maintenance. Thus, Hypothesis 2c is fully supported.

Finally, Hypothesis 3c predicted that modular maintenance would cause more conflicts with the vendor platform than would architectural maintenance, and would be associated with a higher probability of a system failure due to vendor errors. Referring to Table 4 Column 2, we see that the hazard ratio for modular maintenance ($\beta_4 = 3.98$) is about 83% higher than that of architectural maintenance ($\beta_5 = 2.17$). These predicted hazard ratios were statistically significant, lending support to Hypothesis 5 (test Chi-squared = 11.40, $p < 0.01$). This result shows that the chance of a system failure due to vendor errors increases by about two times for every percentage increase in the number of interfaces between modules that were altered during architectural maintenance. However, when technical debt reduction is focused only at the module-level without rectifying the interfaces between modules, every percentage of source

code modified within modules is associated with a four-fold increase in the chance of a system failure due to vendor errors.

Figures 3 and 4 visually present the CIF for system failure due to client and vendor errors respectively, on a scale from 0 to 1. As the graphs show, modular maintenance is more effective in reducing the CIF of system failure due to client errors, but is associated with an increased CIF of system failure due to vendor errors. In contrast, architectural maintenance is less effective in reducing the CIF of system failure due to client errors, but is associated with a lower level of CIF of system failure due to vendor errors. This illustrates the tradeoffs involved in performing software maintenance of enterprise systems in the presence of technical debt.

Figure 3. Cumulative Incidence of System Failure due to Client Errors

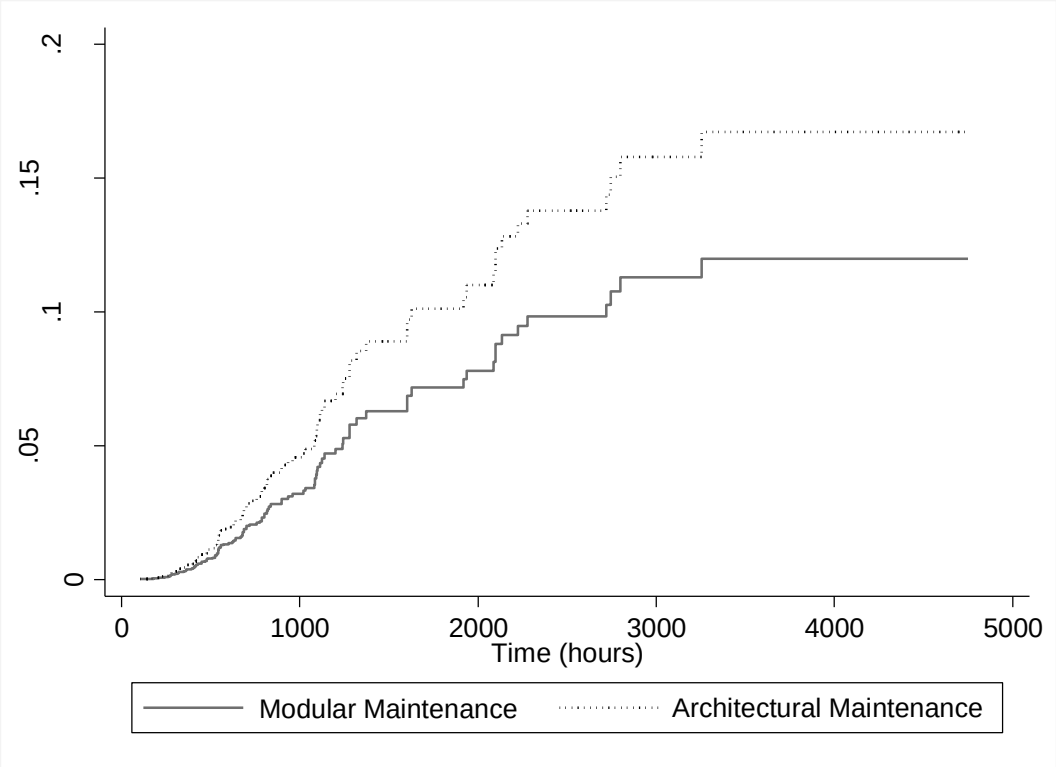
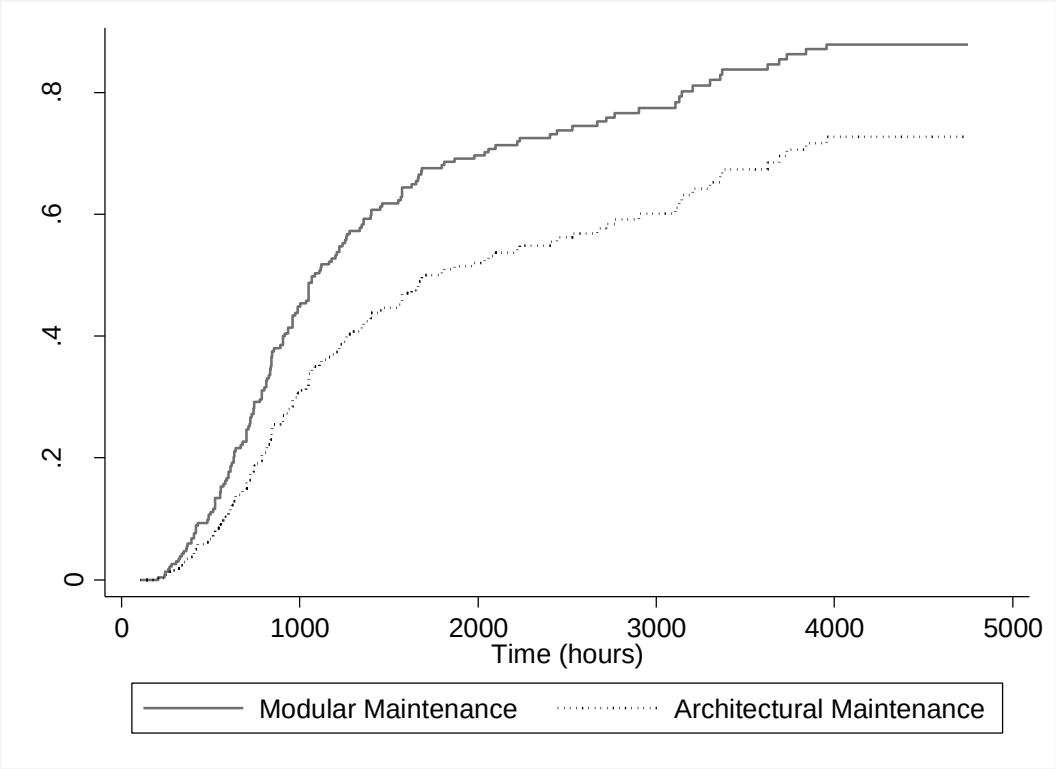


Figure 4. Cumulative Incidence of System Failure due to Vendor Errors

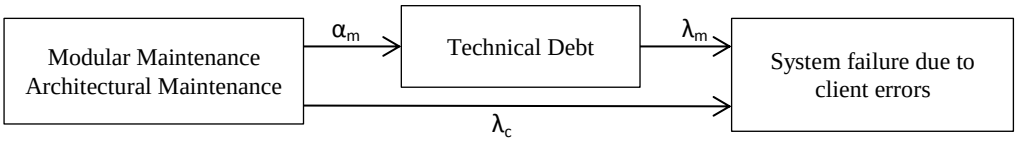


Results for the control variables in the regression models are in the expected directions. We find that volatility slightly increased the chance of a system failure due to both client errors (Table 4, Column 1: $\beta_6 = 1.02$) and vendor errors (Table 4, Column 2: $\beta_6 = 1.08$). More experienced teams were associated with a reduction in the probability of a system failure due to client errors by 38% (Table 4, Column 1: $\beta_7 = 0.62$) and in the probability of a system failure due to vendor errors by about 12% (Table 4, Column 2: $\beta_7 = 0.88$). As expected, code size had a significant detrimental effect on system reliability. Our results show that for every KLOC increase in code size, the probability of a system failure due to client errors increased by 42% (Table 4, Column 1: $\beta_8 = 1.42$) and the chance of a system failure due to vendor errors increased by 45% (Table 4, Column 2: $\beta_8 = 1.45$). The probability of a system failure due to client errors increased with product age (Table 4, Column 1: $\beta_9 = 1.05$) and aging systems were also prone to a higher incidence of system failures due to vendor errors (Table 4, Column 2: $\beta_9 = 1.07$). The overall effect of the use of consultants was to help reduce the chance of a system failure to a small extent (Table 4, Column 1: $\beta_{10} = 0.96$, Column 2: $\beta_{10} = 0.99$). The number of client sites at which the enterprise system was operated did not have a statistically significant effect on the probability of a system failure. The estimates for the transaction volume variable show that heavier usage is associated with a marginal increase in both client-error and vendor-error failures (Table 4, Column 1: $\beta_{12} = 1.01$, Column 2: $\beta_{12} = 1.08$). As expected, we also see a higher probability of system failures associated with logic and data complexity as well as with

the number of previous system failures. Prior vendor failures had a significant and large impact on future system failures due to vendor errors (Table 4, Column 2: $\beta_{16} = 2.19$), which indicates that vendor errors might have a long-term effect on impacting reliability throughout a system's lifecycle.

To test the potential mediation effect[13] of technical debt in relating client-level modular maintenance and architectural maintenance with client-error related system failures, we used a relatively newly developed method for mediation analysis in survival data setting based on the Aalen additive hazard model (Lange and Hansen 2011). The schematic representation of the mediation analysis is shown in Figure 5. α_m represents the change in technical debt as a result of one unit change in modular maintenance as modeled by a linear least squares regression. λ_c and λ_m represent the regression coefficients for Aalen's additive hazard model when the survival status is regressed on maintenance and technical debt respectively.

Figure 5. Mediation Analysis



$$Technical\ debt = \propto_0 + \propto_m maintenance + \varepsilon$$
$$\propto (t) = \beta_0 + \lambda_c maintenance + \lambda_m technical\ debt$$
$$\propto (t) = \beta_0 + \lambda_m \propto_0 + (\propto_m \lambda_m + \lambda_c) technical\ debt$$

The mediation analysis involved two steps. First, we estimated the effects of maintenance on the mediator (technical debt) by a linear regression model. Second, we estimated the effects of both maintenance and technical debt on system failure due to client errors by fitting an additive hazard model adjusted for other covariates. The total effects (TE) on system failure were given by the sum of direct effects (DE) and the indirect effects (IE). The indirect effect through technical debt was given by the product of the parameter estimates for the regression of technical debt on modular maintenance and the parameter estimate of the effect of the technical debt on system failures from the additive hazard models. The mediated proportion is IE/TE. The schematic representation of the mediation analysis is shown in Figure 3. α_m represents the change in technical debt as a result of one unit change in modular maintenance as modeled by a linear least squares regression. λ_c and λ_m represent the regression coefficients for Aalen's additive hazard model when the survival status is regressed on maintenance and technical debt respectively. The results of the mediation analysis are presented in Tables 6 and 7. The results show that there is a significant mediation effect due to technical debt. We find that about 35% of the effect of

[13] Mediation analysis helps to test the structural relationship between client-level maintenance activities, technical debt, and system failures due to client errors, and complements our reasoning associated with hypotheses 2a and 2b. We thank an anonymous reviewer for suggesting this analysis.

modular maintenance on system failure reduction is mediated through technical debt reduction. Similarly, 20% of the effect of architectural maintenance on system failure reduction is mediated through technical debt reduction. These results show that technical debt in an enterprise system accumulated over a period of time by a client could be reduced to a certain degree by targeted client-level maintenance activities. Furthermore, it confirms that modular maintenance undertaken by a client is more effective than architectural maintenance in reducing both the accumulated technical debt and system failures arising from the technical debt.

Table 6: Effect of Maintenance on Technical Debt

	Customization-business logic	Customization-data schema	API Violations
Modular maintenance	-0.81***	-0.38***	-0.51***
Architectural maintenance	-0.19***	-0.25***	-0.18***

Note: Estimates are adjusted for volatility, team experience, code size, logic complexity, data complexity, product age, client sites, consultants use, prior failures, and transaction volume; *** significant at $p < 0.01$

Table 7: Mediation Effects through Technical Debt Reduction

	DE (λ_c)	IE ($\propto_m \lambda_m$)	TE ($\lambda_c + (\propto_m \lambda_m)$)	IE/TE
Customization-business logic				
Modular maintenance	-1.53***	-0.87***	-2.40***	0.36
Architectural maintenance	-0.92***	-0.23***	-1.15***	0.20
Customization-data schema				
Modular maintenance	-1.49***	-0.84***	-2.33***	0.36
Architectural maintenance	-0.90***	-0.19***	-1.09***	0.17
API Violations				
Modular maintenance	-1.61***	-0.87***	-2.48***	0.35
Architectural maintenance	-0.99***	-0.25***	-1.24***	0.20

Note: Estimates are adjusted for volatility, team experience, code size, logic complexity, data complexity, product age, client sites, consultants use, prior failures, and transaction volume; *** significant at $p < 0.01$

6. Discussion

In this section we first use our empirical results to quantitatively evaluate a typical client's business risk exposure due to technical debt accumulation in their enterprise system. Then, we illustrate how managers could perform cost-benefit analysis of maintenance projects that aim to reduce the business risk exposure arising from technical debt. Following this we present the best practices reported to us during our field research to manage technical debt. Finally, we conclude by discussing implications for research.

6.1. Managing Technical Debt in Enterprise Systems

6.1.1. Managerial Evaluation of Business Risk Exposure from Technical Debt

The results presented in Section 5 show that technical debt is associated with a significant increased probability of enterprise system failures. We next utilize the estimated probabilities of system failures from our empirical models to illustrate how software managers at client sites can evaluate their business risk exposures due to technical debt and derive risk management policies accordingly.

Risk exposure is conventionally defined as the potential loss in value due to an unexpected event (Boehm 1991; Lyytinen *et al.* 1998). The unexpected event of focus in this study is a system failure, and the risk exposure of a system failure is derived as:

$$Risk\ exposure = Probability\ of\ system\ failure * Loss\ due\ to\ system\ failure$$

We collected data on the financial consequences of business process disruptions due to system failures through follow-up field interviews with the account managers at the client firms[14]. We collected data pertaining to two of the most commonly used business processes supported by the enterprise software, the warehouse management business process and the accounts payable business process, and derived the risk exposures due to hazards in these two example processes.

Table 8 presents the results of this analysis. Column (1) in Table 8 lists the relevant modules of the enterprise system. The corresponding list of key hazards in the warehouse management and accounts payable business processes at the client firms is shown in Column (2). The extent of technical debt accumulated in the modules, measured in terms of API violations and the business logic and data schema customizations that violated vendor-provided programming and design standards, is listed in Column (3). Applying our empirical model and the cumulative incidence function estimates from Section 5 we derived the estimated probabilities of system failures due to software errors in the modules and present them in Column (4) of Table 8. Multiplying these estimated probabilities of system failures with the average financial loss due to the business process hazards (Table 8, Column (5)) gives the business risk exposure per system failure (Column (6)). The sum of these values for all modules yields a total estimated business risk exposure value of about $0.9 million per firm per year due to technical debt for a typical client in our dataset.

Technical debt build-up in enterprise systems is often not readily visible to business managers (Kruchten *et al.* 2012; Tom *et al.* 2013). The derivation of estimated business risk exposure, as illustrated above, could highlight the potential impact of technical debt and help managers to assess whether the levels of technical debt accumulated in their enterprise systems are appropriate. If the business risk

[14] As shown in Figure 2 in §3 we collected financial, project management, and personnel-related data for each of the system failures at client sites to derive the business risk exposures due to technical debt accumulation in the client systems.

exposure stemming from technical debt is deemed inappropriate, then managers can plan targeted maintenance activities that reduce technical debt in the system and thus help reduce risk exposure in the key business processes supported by the system. We discuss the evaluation of such enterprise system maintenance decisions in the next section.

Table 8. Evaluating Business Risk Exposure due to Technical Debt

(1)	(2)	(3)			(4)	(5)	(6)
Module	Hazard in business process	Technical debt			Estimated probability of system failure	Average business process hazard loss (thousand $)	Business risk exposure (thousand $)
		Customizations: business logic	Customizations: data schema	API Violations			
Invoice management	Customer invoices not generated	24.20	6.05	12.00	0.44	717.16	315.55
Payments management	Incorrect billing amounts collected	11.40	2.85	4.00	0.21	179.58	37.71
Customer order management	Customer orders delivered to wrong locations	3.50	0.88	0.00	0.04	1253.84	50.15
Customer services	Customer-reported issues not recorded	7.80	1.95	2.00	0.11	46.64	5.13
Managerial reports	Regulatory reports not filed	31.90	7.98	13.00	0.54	744.91	402.25
Warehouse management	Wrong stock levels held	6.70	1.68	2.00	0.13	1494.06	194.23
Estimated typical client total annual business risk exposure due to technical debt ≈ $ 924,385							

6.1.2. Managerial Evaluation of Enterprise System Maintenance Decisions

Having demonstrated how managers could utilize the models developed in this study to assess business risk exposures due to system failures, we extend our discussion to illustrate how our results could be utilized to evaluate system maintenance decisions.

Let us consider a scenario where a client firm plans to reduce the estimated $0.9 million business risk exposure discussed above and proposes to conduct modular and architectural maintenance. We illustrate how our empirical model and results could be used to perform a cost benefit analysis to evaluate the above proposal. Following our discussion of competing risks, it is important to note that any project benefits, such as a reduction in business risks of technical debt, need to be adjusted for competing project risks, for example, the unexpected increase in system failures due to vendor errors.

We derived nine hypothetical maintenance project scenarios by choosing values for client maintenance investments at the mean level, and at approximately one standard deviation above and below the mean level for the two types of maintenance, modular and architectural. These nine scenarios are shown in Table 9. The resulting code coverage and architectural interfaces examined in the maintenance projects covered a range from 0% to 70%.

Table 9. Nine Hypothetical Maintenance Project Scenarios (S1-S9)

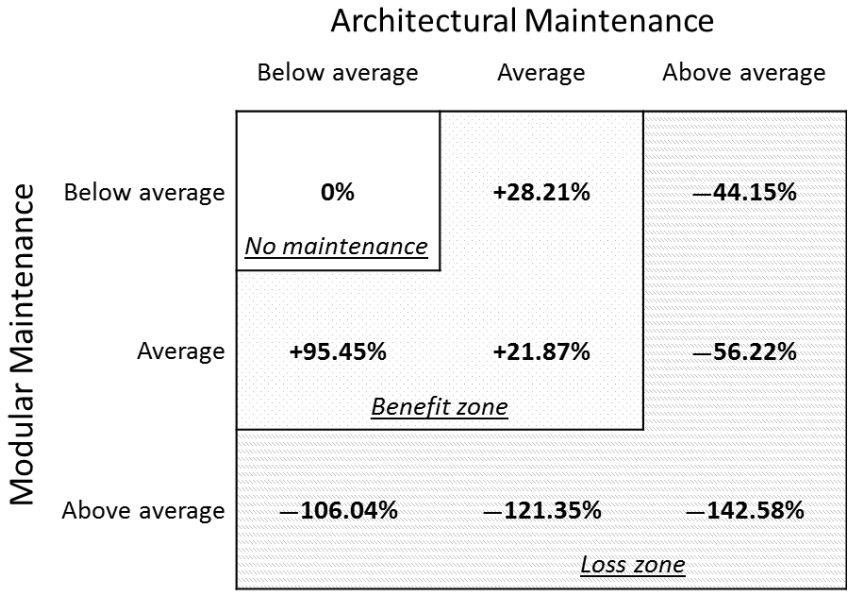
		Architectural maintenance		
		Below average	Average	Above average
Modular maintenance	Below average	S1: (0%, 0%)	S2: (0%, 30%)	S3: (0%, 70%)
	Average	S4: (30%, 0%)	S5: (30%, 30%)	S6: (30%, 70%)
	Above average	S7: (70%, 0%)	S8: (70%, 30%)	S9: (70%, 70%)

For each of the business process hazards in Column (2) of Table 8 we analyze the nine hypothetical scenarios (Table 9) for the proposed maintenance projects that were within the ranges of activity at the client firms in our dataset. For each of these scenarios we collected actual maintenance costs from projects conducted at our client sites. As described in §6.1.1 we derived risk exposure values for each business process hazard under the nine maintenance scenarios. For each of the scenarios we calculated the reduction in risk exposure levels by utilizing the regression estimates of the hazard of system failures due to client errors (Table 4). The modular and architectural maintenance activities in the proposed project would also introduce new hazards arising from conflicts with the errors in the vendor supplied platform. We estimated the increase in business risk exposure due to this new hazard by utilizing the regression results presented in Table 4, and then calculated the overall return on investments for each of the maintenance scenarios.

The pattern of return on investments for different maintenance scenarios is summarized in Figure 6[15]. As can be seen from Figure 6, not all maintenance scenarios yield net positive benefits. The estimated return on investment of Scenario 4 (30% modular coverage and no architectural coverage) is the highest at 95.45%. The project with the highest level of maintenance investments, Scenario 9, which involves 70% modular coverage and 70% architectural coverage, actually results in a net loss with a return of investments of -142.58%, according to this analysis.

[15]Detailed module-level cost-benefit estimates are presented in Table A1 of the online Appendix (§A4). The return on investments patterns are consistent whether we use cumulative project values or average the values over the six modules considered in the illustration.

Figure 6: Estimated Return on Investments from Maintenance Projects



Note: These Figure 6 results are intended only to show the estimated impact of technical debt on maintenance costs and risk exposure under various scenarios. These results should not be interpreted as the effectiveness or ineffectiveness of maintenance policies at the client sites. This analysis does not include potential opportunity costs, business benefits other than maintenance cost savings, and other competitive advantages that a client site might accrue based on the firm's specific technical debt-handling strategy.

These results illustrate our belief that technical debt reduction in enterprise systems is not straightforward because of the interdependencies between client-related and vendor-related tasks and the knowledge asymmetry between the clients and the vendor. Therefore, more investments in client-level maintenance activities may not necessarily translate into business benefits. In the presence of technical debt, higher levels of client maintenance activities are not only expensive, but also have a higher chance of introducing new hazards due to conflicts with vendor's maintenance activities, and thereby causing project losses. On the other hand, projects that aim for targeted and incremental technical debt reduction, which is synchronized with the vendor's roadmap for the product lifecycle, could yield benefits by reducing the business risk exposure of clients. Our empirical models and results, along with a cost-benefit analysis like the one illustrated above, could help managers to evaluate their portfolio of maintenance project proposals in a judicious manner and invest appropriately in the most beneficial portfolios.

6.1.2.1. Industry Best practices

In addition to the statistical analysis this research required an extensive amount of field research and interactions with technical staff and management at the 48 client sites, as summarized in Figure 2.

Through this more qualitative work we observed two apparent key best practices, both of which corroborate well with our empirical results. First, client firms who participated more actively in vendor-initiated activities, such as trade conferences and product roadshows, seemed to be more capable of identifying use-specific configurations and modifications that could potentially be in conflict with the future evolution of the product. Such teams had established good networks with specialists from the vendor firm, which helped reduce their knowledge asymmetry with the vendor. By participating in the vendor-initiated conferences clients had to reveal their firm-specific practices, which had the risk of competitive advantage erosion due to imitation by competitors. However, the benefits of participation may outweigh these perceived risks because they could help reduce the knowledge asymmetry with the vendor, and thereby avoid conflicts with the vendor's product evolution and the subsequent system failures that tend to create significant business losses, as estimated above.

The second best practice we observed involved choosing the appropriate modules of the enterprise system to customize and modify. It is a common practice for clients of COTS systems to aggressively modify vendor-supplied modules that are perceived as "weak", for example, modules that lack functionality when they are released (Davenport 1998). However, counter-intuitively, client firms who had the highest levels of system reliability, more often than not, seemed to avoid modifying the weaker modules supplied by the vendor. This strategy makes sense in light of our empirical results related to competing risks in the COTS enterprise system context. Over the lifecycle of a COTS enterprise system product the vendor typically tends to allocate resources to the modules that were perceived as weak during the initial release and attempts to improve them over the product's evolution. A client who engages in aggressive source code modifications early on in an attempt to rectify the vendor-supplied module weaknesses tends to accumulate technical debt, all else being equal. Even though such client customizations might yield short-term business benefits, as our results show the accumulated technical debt has the potential to cause severe system failures later in the system's lifecycle. Completely eradicating the accumulated technical debt is challenging, and therefore the business risk exposure values of these clients increase with time. In contrast, clients who adopt a "wait and watch" strategy benefit from vendor-initiated improvements in enterprise systems with long lifecycles. Therefore, we recommend judicious selection of modules for pursuing use-specific modifications and holding back such customization projects until the vendor's standard release has reasonably matured.

6.2. Implications for Research

In addition to the above managerial implications we highlight three important implications for research. First, this study has empirically quantified the negative impact of technical debt on enterprise system reliability and the presence of competing risks that influence enterprise system failures. Therefore,

in the enterprise systems context, software maintenance decision models need to account for the patterns of technical debt accumulation and reduction over the lifespan of the systems while deriving optimal maintenance policies. Furthermore, construction of models that account for costs of software quality need to be done in a way to accommodate the competing risks that accompany technical debt reduction. The presence of technical debt often makes software maintenance imperfect, and omitting the new hazards introduced during a maintenance activity would lead to biased calculations of returns on investments of software maintenance activities and business risk exposures valuations.

Second, this study has established the presence of interdependencies between vendor and client tasks in the toolkit approach to enterprise software product development, which significantly influences the reliability of the systems and consequently the business risk exposure of clients. There is a need for research that sheds light on client and vendor strategies that could mitigate the effects of this interdependence, possibly through a reduction of knowledge asymmetry between the vendor and clients. Furthermore, exploring the co-evolution of the enterprise system variants at client firms and the vendor's standard product releases could reveal how the interdependence between client and vendor versions of the product varies over the lifecycle of the enterprise system, revealing clues as to how the transitions to new product versions could be effectively managed by clients.

Finally, our examination of the relative effects of modular and architectural maintenance activities to reduce technical debt revealed a challenging tradeoff. We found that modular maintenance activities were more effective in reducing system failures due to software errors from client modifications than architectural maintenance. However, modular maintenance activities increase the probability of system failures due to residual software errors that are found in the vendor-supplied platform by as much as two times more than architectural maintenance. Because of this tradeoff, as illustrated in §6, calculating the benefits of software maintenance investments targeted at reducing technical debt is not straightforward. Therefore, research studies that focus on deriving maintenance policies and designing software maintenance project portfolios need to carefully account for the accumulated levels of technical debt in enterprise systems, as well as the distribution of resources across modular and architectural maintenance.

Future research could expand the contributions of this study in a number of ways. First, replicating the study using different COTS products from multiple vendors would help to assess the degree of generality of the results. Second, and more expansive, would be an investigation of the impact of technical debt in product platform ecosystems, where distinct product families are based on a single core software platform (Meyer *et al.* 1997; Ceccagnoli *et al.* 2012). The competing risks approach advocated in this research could be particularly useful in understanding how technical debt in one product family propagates across the ecosystem in which it is situated. Another research direction would be to

examine how vendors and clients of enterprise systems manage technical debt during periods of technological transitions (Iansiti 2000). Investigating the circumstances under which technical debt hastens or hinders transitions to new technological paradigms would help further our understanding of the diffusion of new innovations in the software product development industry. There is also an opportunity to expand the cost-benefit analysis of technical debt reduction reported in this paper. For example, future research could extend the analysis by attempting to specifically account for business opportunity costs that arise from a technical debt-avoiding maintenance policy as well as fully accounting for possible heterogeneity in client site-level maintenance policies.[16] Finally, examining the relationship between the software process maturity of product development teams and technical debt in their projects could help to establish important process-related antecedents of technical debt accumulation in enterprise software products.

7. Summary and Conclusion

This research investigated the reliability of COTS-based enterprise systems, which are critical business platforms for the operations of modern organizations. We specifically addressed the research challenges that stem from the interdependencies between the vendor-supplied enterprise system platform and the customizations done by individual clients through a competing risks analysis approach. By empirically examining a longitudinal dataset spanning the 10 year lifecycle of a COTS-based enterprise software system deployed at 48 different client firms, we advanced a method to measure technical debt in an enterprise system and assess its impact on the reliability of the system. The competing risks approach we deployed for our empirical analysis enabled us to accurately estimate the impact of technical debt by accounting for event-specific hazards that impact the reliability of an enterprise software system due to the specific actions taken by both the vendor and the client firms throughout the evolution of the system. Moreover, we examined the economic payoff from modular and architectural software maintenance aimed at reducing the technical debt of an enterprise system, and demonstrated how the results of the study could aid managerial decision-making related to controlling business risk exposures due to enterprise system failures.

[16] We thank an anonymous reviewer for this suggestion.

References

Adner, R., and Levinthal, D., 2001. "Demand Heterogeneity and Technology Evolution: Implications for Product and Process Innovation," *Management Science* 47(5), pp. 611-628.

Agarwal, M., and Chari, K., 2007. "Software Effort, Quality, and Cycle Time: A Study of CMM Level 5 Projects," *IEEE Transactions on Software Engineering* 33(3), pp. 145-156.

Akkermans, H., and van Helden, K., 2002. "Vicious and Virtuous Cycles in ERP Implementation: A Case Study of Interrelations between Critical Success Factors," *European Journal of Information Systems* 11(1), pp. 35-46.

Ammann, P., and Offutt, J., 2008. *Introduction to Software Testing*. Cambridge University Press: New York, NY.

Andersen, R., and Morch, A., 2009. "Mutual Development: A Case Study in Customer-Initiated Software Product Development," *Lecture Notes in Computer Science*, volume 5435, 2009, pp 31-49.

Austin, R.D., 2001. "The Effects of Time Pressure on Quality in Software Development: An Agency Model," *Information Systems Research* 12(2), pp. 195-207.

Baldwin, C., Hienerth, C., and von Hippel, E., 2006. "How User Innovations Become Commercial Products: A Theoretical Investigation and Case Study," *Research Policy* 35(9), pp. 1291-1313.

Banker, R.D., Datar, S.M., Kemerer, C.F., 1991. "A Model to Evaluate Variables Impacting the Productivity of Software Maintenance Projects," *Management Science* 37(1), 1991, pp. 1-18.

Banker, R.D., Davis, G.B., Slaughter, S.A., 1998. "Software Development Practices, Software Complexity, and Software Maintenance Performance: A Field Study," *Management Science*, 44(4), pp. 433-450.

Banker, R.D. and Kemerer, C.F., 1989. "Scale Economies in New Software Development," *IEEE Transactions on Software Engineering* 15(10), pp. 1199-1205.

Banker, R.D. and Slaughter, S.A., 1997. "A Field Study of Scale Economies in Software Maintenance," *Management Science* 43(12), pp 1709-1725.

Banker, R.D., and Slaughter, S.A., 2000. "The Moderating Effects of Structure on Volatility and Complexity in Software Enhancement," *Information Systems Research* 11(3), pp. 219-240.

Barry, E., Kemerer, C.F., and Slaughter, S.A., 2006. "Environmental Volatility, Development Decisions and Software Volatility: A Longitudinal Analysis", *Management Science*, 52(3), pp. 448-464.

Basili, V.R., and Boehm, B.W., 2002. "COTS-based Systems Top 10 list," *IEEE Computer* 34(5), pp. 91-95.

Boehm, B.W., 1991. "Software Risk Management: Principles and Practices," *IEEE Software* 8(1), pp. 32-41.

Boh, W.F., Slaughter, S.A., and Espinosa, J.A., 2007. "Learning from Experience in Software Development: A Multilevel Analysis," *Management Science* 53 (8), pp. 1315-1331.

Box-Steffensmeier, J.M., and Zorn, C., 2002. "Duration models for repeated events," *Journal of Politics* 64(4), pp. 1069-1094.

Box-Steffensmeier, J.M., De Boef, S., and Joyce, K. A., 2007. "Event Dependence and Heterogeneity in Duration models: The Conditional Frailty Model," *Political Analysis* 15(3), pp. 237-256.

Brown, N., Cai, Y., Guo, Y., Kazman, R., *et al*., 2010. "Managing Technical Debt in Software-reliant Systems," *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, pp. 47-52.

Ceccagnoli, M., Forman, C., Huang, P., and Wu D.J., 2012. "Co-creation of Value in a Platform Ecosystem: The Case of Enterprise Software," *MIS Quarterly* 36(1), pp. 263-290.

Chellappa, R., Sambamurthy, V., and Saraf, N., 2010. "Competing in Crowded Markets: Multimarket Contact and the Nature of Competition in the Enterprise Systems Software Industry," *Information Systems Research*, 21(3), pp. 614-630.

Cox, D.R., 1972. "Regression Models and Life-tables," *Journal of the Royal Statistical Society*, Series B 34(2), pp. 187-220.

Curtis, B., Jay, S., Szynkarski, A., 2012. “Estimating the Principal of an Application’s Technical Debt,” *IEEE Software* 29(6), pp. 34-42.
Darcy, D.P., Kemerer, C.F., Slaughter, S.A., and Tomayko, J.E., 2005. "The Structural Complexity of Software: An Experimental Test," *IEEE Transactions on Software Engineering* 31(11), pp. 982-995.
Dauxois, J-Y., and Sencey, S., 2009. “Non-parametric tests for recurrent events under competing risks,” *Scandinavian Journal of Statistics*, 36(4), pp. 649-670.
Davenport, T., 1998. “Putting the Enterprise into the Enterprise System,” *Harvard Business Review* 76(4), pp. 121-131.
Deelstra, S., Sinnema, M., and Bosch, J., 2005. “Product Derivation in Software Product Families: A Case Study,” *Journal of Systems and Software* 74(2), pp.173-194.
Devaraj, S., and Kohli, R., 2003. “Performance Impacts of Information Technology: Is Actual Usage the Missing Link,” *Management Science* (49:3), pp. 273-289.
Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., and Mockus, A., "Does Code Decay? Assessing the Evidence from Change Management Data," *IEEE Transactions on Software Engineering* 27(1), pp.1-12.
Esteves, J., and Bohorquez, V., 2007. “An Updated ERP Systems: Annotated Bibliography 2001-2005,” *Communications of the AIS* 19(18), pp. 386-446.
Ethiraj, S.K., and Levinthal D., 2004. "Modularity and Innovation in Complex Systems," *Management Science* 50(2), pp. 159-173.
Ethiraj, S.K., Ramasubbu, N., and Krishnan, M.S., 2012. “Does Complexity Deter Customer-Focus?,” *Strategic Management Journal* 33(2), pp. 137-161.
Franke, N. and von Hippel, E., 2003. “Satisfying Heterogeneous User Needs via Innovation Toolkits: The Case of Apache Security Software,” *Research Policy* 32(7), pp. 1199-1215.
Franke, N. and Piller, F., 2004. “Value Creation by Toolkits for User Innovation and Design: The case of the watch market,” *Journal of Product Innovation Management* 21(6), pp. 401-415.
Fine, J.P. and Gray, R.J., 1999. “A Proportional Hazards Model for the Subdistribution of a Competing Risk,” *Journal of American Statistical Association* 94 (446), pp. 496-509.
Finney, S., and Corbett, M. 2007. “ERP Implementation: A Compilation and Analysis of Critical Success Factors,” *Business Process Management Journal* 13(3), pp. 329-347.
Gable, G.G., 1996. “A Multidimensional Model of Client Success When Engaging External Consultants,” *Management Science* 42(8), pp. 1175-1198.
Garg, S., Puliafito, A., Telek, M., and Trivedi, K., 1998. “Analysis of Preventive Maintenance in Transaction Based Software Systems,” *IEEE Transactions on Software Engineering* 47(1), pp. 96-107.
Gartner 2013, http://www.gartner.com/newsroom/id/2537815.
Gjerde., K.A.P., Slotnick, S.A., and Sobel, M.J., 2002. "New Product Innovation with Multiple Features and Technology Constraints," *Management Science* 48(10), pp. 1268-1284.
Greenstein, S., and Wade, J.B. 1998. "The Product Lifecycle in the Commercial Mainframe Computer Market, 1968-1982," *Rand Journal of Economics*, 29(4), pp. 772-789
Grover, V., Fiedler, K., and Teng, J. 1997. “Empirical Evidence on Swanson’s Tri-core Model of Information Systems Innovation,” *Information Systems Research* 8(3), pp.273-287.
Jin, Z., and Offutt, J., 2001. “Deriving Tests from Software Architectures,” *Proceedings of the 12th International Symposium on Software Reliability Engineering*, pp. 308-313.
Harter, D.E., Krishnan, M.S., and Slaughter, S.A., 2000. “Effects of Process Maturity on Quality, Cycle Time, and Effort in Software Product Development,” *Management Science* 46(4), pp. 451-466.
Henderson, R.M, and Clark, K. 1990. “Architectural Innovation: The Reconfiguration of Existing Product Technologies and the Failure of Established Firms,” *Administrative Science Quarterly* 35(1), pp. 9-30.
Hitt, L., Wu, D.J., and Zhou, X., 2002. "Investment in Enterprise Resource Planning: Business Impact and Productivity Measures," *Journal of Management Information Systems* 19(1), pp. 71-98.

Holland, C.P., and Light, B., 1999. “A Critical Success Factors Model for ERP Implementation,” *IEEE Software* 16(3), pp. 30-36.

Huang, P., Ceccagnoli, M., Forman, C., and Wu, D.J., 2013. “Appropriability Mechanisms and the Platform Partnership Decision: Evidence from Enterprise Software,” *Management Science* 59(1), pp. 102-121.

Huckman, R.S., Staats, B.R., and Upton, D.M., 2009. "Team Familiarity, Role Experience, and Performance: Evidence from Indian Software Services," *Management Science* 55(1), pp. 85-100.

Iansiti, M., 2000. “How the Incumbent Can Win: Managing Technological Transitions in the Semiconductor Industry,” *Management Science* 46(2), pp. 169-185.

Ji, Y., Kumar, S., Mookerjee, V., Sethi, S.P., and Yeh, D., 2011, “Optimal enhancement and lifetime of software systems: A control theoretic analysis,” *Production and Operations Management* 20(6), 889-904

Kapur, P.K., Pham, H., Anand, S., and Yadav, K., 2011. “A Unified Approach for Developing Software Reliability Growth Models in the Presence of Imperfect Debugging and Error Generation,” *IEEE Transactions on Reliability* 60(1), pp. 331-340.

Kemerer, C.F., 1995. “Software Complexity and Software Maintenance: A Survey of Empirical Research,” *Annals of Software Engineering* 1(1), pp. 1-22.

Kemerer, C. and Darcy, D. 2005. “OO Metrics in Practice”, *IEEE Software*, 22(6), pp. 17-19.

Kemerer, C. and Paulk, M. 2009. "The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data", *IEEE Transactions on Software Engineering*, 35(4), pp. 534-550.

Ko, D-G., Kirsch, L.J., and King, W.R., 2005. “Antecedents of Knowledge Transfer from Consultants to Clients in Enterprise System Implementations,” *MIS Quarterly* 29(1), pp. 59-85.

Kruchten, P., Nord, R.L., and Ozkaya, I., 2012. “Technical Debt: From Metaphor to Theory and Practice,” *IEEE Software* 29(6), pp. 18-21.

Kuk, D., and Varadhan, R., 2013. “Model Selection in Competing Risks Regression,” *Statistics in Medicine* 32(18), pp. 3077-3088.

Lau, B., Cole S.R., and Gange, S.J., 2009. “Competing Risk Regression Models for Epidemiologic Data,” *American Journal of Epidemiology* 170(2), pp. 244-256.

Li, S., Shang, J., and Slaughter, S.A., 2010. “Why do software firms fail? Capabilities, Competitive Actions, and Firm Survival in the Software Industry from 1995 to 2007,” *Information Systems Research* 21(3), pp. 631-654.

Lyytinen, K., Mathiassen, L., and Ropponen, J., 1998. “Attention Shaping and Software Risk-A Categorical Analysis of Four Classical Risk Management Approaches,” *Information Systems Research* 9(3), pp. 233-255.

MacCormack, A., Verganti, R., and Iansiti, M., 2001. “Developing Products on “Internet Time”: The Anatomy of a Flexible Development Process,” *Management Science*, 47(1), pp. 133-150.

MacCormack, A., and Verganti, R., 2003. “Managing the Sources of Uncertainty: Matching the Process and Context in Software Development,” *Journal of Product Innovation Management*, 20(3), pp. 217-232.

MacCormack, A., Rusnak, J., and Baldwin, C.Y., 2006. “Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code,” *Management Science*, 52(7), pp. 1015-1030.

MacCormack, A., LaMantia, M.J., Cai, Y., and Rusnak, J., 2008. “Analyzing the Evolution of Large-Scale Software Systems using Design Structure Matrices and Design Rule Theory: Two Exploratory Cases,” *Proceedings of the 7th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, Vancouver, Canada, pp 83-92.

Markus, M.L., Tanis, C., and van Fenema, P.C., 2000. “Enterprise Resource Planning: Multisite ERP Implementations,” *Communications of the ACM* 43(4), pp. 42-46.

McCabe, T.J., 1976. “A Complexity Measure,” *IEEE Transactions on Software Engineering* (SE-2:4), pp. 308-320.

McCabe, T.J., and Butler, C.W., 1989. "Design Complexity Measurement and Testing," *Communications of the ACM* (32:12), pp. 1415-1423.

Mens, T., and Tourwe, T., 2004. "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering*, 30(2), pp. 126-139.

Meyer, M.H., Tertzakian, P, and Utterback, J.M., 1997. "Metrics for Managing Research and Development in the Context of the Product Family," *Management Science* 43(1), pp. 88-111.

Marinescu, R., 2012. "Assessing Technical Debt by Identifying Design Flaws in Software Systems," *IBM Journal of Research and Development* 56(5), pp. 9:1-9:13.

Masini, A., and Wassenhove, L.N., 2009. "ERP Competence-Building Mechanisms: An Exploratory Investigation of Configurations of ERP Adopters in the European and U.S. Manufacturing Sectors," *Manufacturing and Service Operations Management* 11(2), pp. 274-298.

Nah, F. F-H., Lau, J. L-S., Kuang, J., 2001. "Critical Success Factors for Successful Implementation of Enterprise Systems," *Business Process Management Journal* 7(3), pp. 285-296.

Parnas, D.L., 1994. "Software Aging," *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 16-21, 1994, pp. 279-287.

Pintilie, M., 2007. "Analysing and Interpreting Competing Risk Data," *Statistics in Medicine* 26(6), pp. 1360-1367.

Ramdas, K., and Randall, T., 2008. "Does Component Sharing Help or Hurt Reliability? An Empirical Study in the Automotive Industry," *Management Science* 54(5), pp. 922-938.

Ramasubbu, N., Mithas, S., and Krishnan, M.S., 2008. "High Tech, High Touch: The Effect of Employee Skills and Customer Heterogeneity on Customer Satisfaction with Enterprise System Support Services," *Decision Support Systems* 44(2), pp. 509-523.

Ramasubbu, N., Kemerer, C.F., and Hong, J., 2012. "Structural Complexity and Programmer Team Strategy: An Experimental Test," *IEEE Transactions on Software Engineering* 38(5), pp. 1054-1068.

Ramasubbu, N. and Kemerer, C.F., 2014. "Managing Technical Debt in Enterprise Software Packages", *IEEE Transactions on Software Engineering* 40(8), pp. 758-772.

Ranganathan, C., and Brown, C.V., 2006. "ERP Investments and the Market Value of Firms: Toward an Understanding of Influential ERP Project Variables," *Information Systems Research* 17(2), pp. 145-161.

Reiss, S.P., 2006. "Incremental Maintenance of Software Artifacts," *IEEE Transactions of Software Engineering* 32(9), pp. 682-697.

Rogers, W.H., 1994. "ssa4: Ex post tests and diagnostics for a proportional hazards model," *Stata Technical Bulletin 19*, College Station, TX: Stata Press, pp. 23-27.

Sarker, S., and Lee, A.S., 2003. "Using a Case Study to Test the Role of Three Key Social Enablers in ERP Implementation," *Information and Management* 40(8), pp. 813-829.

Subramanayam, R., and Krishnan, M.S., 2003. "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects," *IEEE Transactions on Software Engineering* 29(4), pp. 297-310.

Subramanyam, R., Ramasubbu, N., and Krishnan, M.S., 2012. "In Search of Efficient Flexibility: Effects of Software Component Granularity on Development Effort, Defects, and Customization Effort," *Information Systems Research*, 23(3), pp. 787-803

Slaughter, S.A., Harter, D.E., and Krishnan, M.S., 1998. "Evaluating the Cost of Software Quality," *Communications of the ACM* 41(8), pp. 67-73.

Slaughter, S.A., and Kirsch, L.J., 2006. "The Effectiveness of Knowledge Transfer Portfolios in Software Process Improvement: A Field Study," *Information Systems Research* 17(3), pp. 201-320.

Taylor, L.L., and Pena, E.A., 2013. "Nonparametric estimation with recurrent competing risks data," *Lifetime Data Analysis*, pp. 1-24.

Tom, E., Aurum, A., and Vidgen, R., 2013. "An Exploration of Technical Debt," *Journal of Systems and Software* 86(6), pp. 1498-1516.

Umble, E.J., Haft, R.R., and Umble, M.M., 2003. “Enterprise Resource Planning: Implementation Procedures and Critical Success Factors,” *European Journal of Operations Research* 146(2), pp. 241-257.

Vogt, C., 2002. “Intractable ERP: A Comprehensive Analysis of Failed Enterprise-Resource-Planning Projects,” *ACM SIGSOFT Software Engineering Notes* 27(2), pp. 62-68.

von Hippel, E., 1998. "Economics of Product Development by Users: The Impact of “Sticky” Local Information," *Management Science* 44(5), pp. 629-644.

von Hippel, E., and Katz, R., 2002. "Shifting Innovation to Users via Toolkits," *Management Science* 48(7), pp. 821-833.

Williams, R., and Pollock, N., 2012. “Moving Beyond the Single Site Implementation Study: How (and Why) We Should Study the Biography of Packaged Enterprise Solutions,” *Information Systems Research*, 23(1), pp. 1-22.

Woodard, C.J., Ramasubbu, N., Tschang, T., and Sambamurthy, V., 2013. “Design capital and design moves: the logic of business strategy,” *MIS Quarterly* 37(2), pp. 537-564.

Wu, D.J., Ding, M. and Hitt, L., 2013. “IT Implementation Contract Design: Analytical and Experimental Investigation of IT Value, Learning, and Contract Structure” *Information Systems Research* 24 (3), pp. 787–801.

Zhou, B., Fine, J., Latouche, A., and Labopin, M., 2012. “Competing risks regression for clustered data,” Biostatistics 13(3), pp. 371-383.

Online Appendix

A1. Mining the Software Repository and Identifying Software Changes

A1.1. Mining the Software Repository

At the vendor and client sites of this research study the software production process involved a robust software configuration management system called Perforce version control (Perforce 2014). This system was the central repository that stored the different versions of the software and associated documents during the ten year lifecycle of the enterprise systems we studied. Data in this central repository was organized in terms of projects. Each project consisted of system files at the granularity of revisions done to the system files at every hour during the ten year period. Overall, there were about 6100 projects, 45 million system files, and about 1.2 billion file revisions done by 4133 developers accounting to a total size of about 2TB of software archival data.

Although the individual software production processes at the 48 client firms varied, they all followed a similar structure composed of development and maintenance phases, and the data in the repository reflected this distinction between the different phases, and project data was organized into different *trunks* and *branches* of a product release. When the vendor released a maintenance patch (called a service pack), clients created a separate branch of a project in which they assembled, tested, and deployed the service pack provided by the vendor. All client-driven changes were also maintained in the same way in separate project trunks and branches. This organization helped us parse the software repositories and identify all the individual changes to a client system, and classify them into vendor or client-driven changes. Similar to the experience of other software researchers, we were able to identify which software changes happened, and where they happened in the source code for the systems at all our client firms (Holschuh *et al.* 2009).

A1.2. Identifying and Sequencing Changes

To build an empirical dataset from the software repository we utilized a proprietary tool supplied by the vendor to parse the software code in the repository. Utilizing this tool, we were able to parse the different versions of the system files in the repository (version histories) to identify the components of an enterprise system that changed together at a given time, which is termed *evolutionary coupling* (Zimmerman *et al.* 2005). Using the vendor-supplied tool we could identify evolutionary coupling at different granularities, such as at the levels of hourly time intervals, system files, and even at the entities level, for example at the level of classes and methods in object-oriented program code. We utilized these evolutionary couplings to perform the phase and sequence analyses to identify and sequence the changes that happened to the enterprise systems at our client sites over the ten year period. Through trial and error, and iteratively refining the evolutionary coupling entities at different time intervals, we determined that a monthly aggregation of changes was optimal for our empirical data construction. This aligned well with the monthly product release history documents that we collected from the product management division of the firm. Therefore, our dataset is organized according to 120 evolutionary phases (12 months x 10 years) during which we observe system failures.

A1.3. Cause-Effect Analysis

For each incidence of a system failure we performed a cause-effect analysis utilizing the fault tree analysis (FTA) procedures established in the software engineering literature (Nakajo and Kume 1991). During FTA we utilized pre-defined classification methods using the failure causes already known to the clients and the vendor, as well as emergent classification schemes based on clustering of similar data from the historical software repositories. The emergent classification schemes were particularly helpful to identify root causes of system failures when a particular system change involved both client and vendor-driven changes. Whenever there was ambiguity in the root causes classification, we collaborated with the client and vendor system administrators and quality assurance personnel to resolve the ambiguity. In the

final dataset all system failures were clearly marked as caused by either client errors *or* vendor errors, and there were no overlap between these classifications.

A2. Data Structure

We tracked the evolution of the enterprise systems at 48 clients over a ten year period. Figure A1 shows the cumulative time period in months when the system was active in production at each of the 48 client sites and the total number of client error and vendor error failures at those sites. We can see from Figure A1 that the data is right-censored as the systems at different client sites experienced different failure rates, maintenance schedules, and were eventually retired at different time periods. In a given time period, a system at a client site could fail because of client errors or due to vendor errors. That is, client and vendor errors *compete* for system failure. A system failure typically evokes a comprehensive maintenance response including a systematic root-cause analysis and bug fixing effort. After the maintenance following a failure, a client system is typically denominated with a different version number because it would have been 'patched' with the latest bug and stabilization fixes released by the vendor. To employ the competing risks modeling approach, we treat the different versions of a system at a client site as siblings in a family and cluster them (using client id) to account for correlations due to unobserved shared factors. Overall, the dataset includes a total of 3641 observations with 193 system failures due to client errors and 191 system failures due to vendor errors.

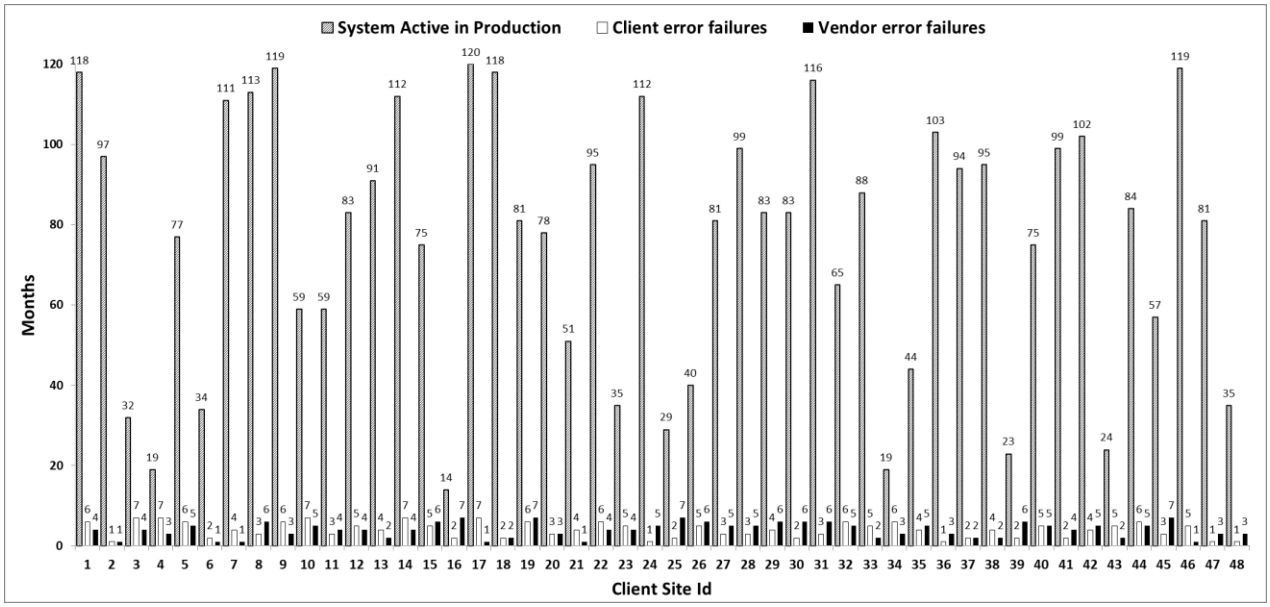


Figure A1. Histogram of Observations

A3. Technical Debt, Modular and Architectural Maintenance: An Example from the Dataset

In this example we focus on events specific to the warehouse management business process of client #23 in our dataset. At time t=1 the warehouse management system went live in production at the client site. Throughout the first year (until t=12) the client used the vendor-provided system without any modifications. At time t=13, the client modified the source code and database schema of the system to implement a modified way of allocating warehouse space to items. This involved redefining the barcode generation scheme, as well as the corresponding storage procedures in the database. In implementing this custom business logic, the client violated 3 vendor-provided APIs (barcode generator, warehouse job scheduler, database monitoring job). Furthermore, 30% of the modified code and 25% of the new database schema also violated the vendor-prescribed design and programming standards. While

implementing the new system changes the client team also undertook maintenance activities to integrate the new source code and database schema with other parts of the system. These maintenance activities involved examining 10% of the overall source code of the system and 5% of the interconnections between the different modules involved in the system integration.

At the start of the next time period (t=14), the modified system went live in production at the client site. For the next three time periods no system failures were reported. However, during this period the vendor released service packs to update its base platform. Normally (as seen in t=2 to t=12), these vendor-provided updates were automatically patched to the system without any additional effort required from the client. However, since the source code was modified at t=13, automatic updating of vendor service packs was disabled. To implement the vendor service packs, the client team had to manually inspect and test the system. We observed that the maintenance effort to implement each of the vendor update packs was similar to the effort spent to integrate the business logic and database change made by the client at t=13 (modular maintenance: 10% and architectural maintenance:5%).

At t=18 the system failed. The root cause analysis revealed that the failure was due to a client error that stemmed from the system modifications done by the client at t=13. The system failed because it could not resolve a conflict when two different items with the same barcodes were competing for warehouse shelf space at the same time. Because of faulty business logic in the barcode generation function implemented by the client, two different items were allocated the same barcode at the same time. Furthermore, the vendor-provided database schema and API for periodic database monitoring were circumvented by the client. Thus, the vendor-provided database monitoring job could not alert about duplicate barcodes in the database in a timely fashion to prevent the eventual conflict and system failure. To resolve the system failure issue the client teams undertook maintenance activities that involved modification of the barcode generation and warehouse space allocation functions. The database was also purged of all duplicate barcode records for different warehouse items. The maintenance also contributed to a decrease in technical debt in the system as the client teams partially reverted back to the original vendor-provided libraries for the barcode generation, warehouse space allocation, and database monitoring functions. One of the API violations pertaining to the database monitoring job was also fully resolved. Overall, this corrective maintenance effort involved modular maintenance activities that involved examination of about 75% of all module-level functions in the system and architectural maintenance that involved examining about 20% of the interfaces between the modules.

Between t=19 and t=23 there were no system failures at the client site. However, as seen earlier (t=14 to t=17), vendor-provided updates were not automatically patched to the system. To implement the vendor service packs, the client team had to manually inspect and test the system (modular maintenance: 50% and architectural maintenance: 10%). We observed that the maintenance effort to implement each of the vendor update packs was higher than before at t=13 and t=14. More of the maintenance effort was spent on modular maintenance, presumably to avoid system failures similar to the one that happened at t=18.

At t=24 the system failed again. This time the root cause analysis was a lengthy and complex process as the client and vendor teams had to go back and forth to localize the error and allocate responsibilities to fix the problem. Eventually, the root cause analysis revealed that the failure was due to a dormant vendor error related to the cache sizes allocated to barcodes. The original vendor design for file caching had not anticipated more than a million items in the warehouse. Even though the vendor's

database scheme could accommodate more than a billion barcodes, the file caching systems implemented by the vendor could not handle barcodes that were more than 7 digits. The client's modified business logic and the subsequent maintenance to the barcode generation scheme to avoid duplicates (t=18) had contributed to a deluge of barcodes in the client's system. At t=24 the newly generated barcodes exceeded 7 digits and triggered the error in the file caching function that caused the system failure. Thus, the client modification of the barcode generation scheme and the subsequent client-driven maintenance activities to prevent duplicate records had triggered a dormant vendor error.

The vendor issued a service pack to rectify the file caching problem at no cost to the client. However, to implement the service pack the client had to undertake additional maintenance activities as automatic updating of vendor service packs was not possible because of the source code modifications. This maintenance effort was mostly spent on architectural maintenance including the verification of file caching interfaces across the different modules, the corresponding API parameters, and the modified database schemas. It is important to note that at the end of t=24 when the system was reinstituted after the resolution of the file caching error, technical debt level of the client's system had not altered. This is because the maintenance activities pertaining to the resolution of the file caching error did not involve rolling back the modifications to the business logic or database changes enacted at t=13 and subsequently at t=18.

A4. Return on Investments for Various Maintenance Scenarios

Table A1 presents detailed module-level results of the cost benefits analysis for various maintenance scenarios, which were summarized in Figure 6 in §6.1.2 of the paper. For each of the business process hazards in Column (2) of Table A1 we analyze the nine hypothetical scenarios (Column (3)) for the proposed maintenance projects that were within the ranges of activity at the client firms in our dataset. For each of these scenarios we collected actual maintenance costs from projects conducted at our client sites. The average values of these maintenance costs for the respective modules are reported in Column (4) of Table A1. As described in §6.1.1 of the paper, we derived risk exposure values for each business process hazard under the nine maintenance scenarios. For each of the scenarios we calculated the reduction in risk exposure levels by utilizing the regression estimates of the hazard of system failures due to client errors (§5, Table 4). These values are presented in Column (5) of Table A1. The modular and architectural maintenance activities in the proposed project might also introduce new hazards arising from conflicts with the errors in the vendor supplied platform. We estimated the increase in business risk exposure due to this new hazard by utilizing the regression results presented in the paper (§5, Table 4), and report them in Column (6) of Table A1. Column (7) presents the module-level net benefits or losses estimated for each of the maintenance scenarios, and the corresponding return on investments is shown in Column (8) of Table A1.

Table A1. Evaluating Enterprise System Maintenance Decisions

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
Module	Hazard in business process	Project Scenario (modular%, architectural%)	Maintenance project cost ($)	Reduction in risk exposure (thousand $)	Increase in risk exposure (thousand $)	Benefits (Losses) of maintenance project ($)	Return on Investments (%)
Invoice management	Customer invoices not generated	S1: (0%, 0%)	$0.00	$0.00	$0.00	$0.00	0.00
		S2: (0%, 30%)	$42,210.07	$78.89	$15.78	$20,899.93	49.51
		S3: (0%, 70%)	$98,490.15	$94.67	$25.24	-$29,069.15	-29.51
		S4: (30%, 0%)	$28,140.04	$126.22	$59.95	$38,125.46	135.48
		S5: (30%, 30%)	$70,350.11	$205.11	$63.11	$71,647.39	101.84
		S6: (30%, 70%)	$126,630.20	$220.89	$94.67	-$410.20	-0.32
		S7: (70%, 0%)	$65,660.10	$189.33	$126.22	-$2,550.10	-3.88
		S8: (70%, 30%)	$107,870.17	$268.22	$189.33	-$28,982.67	-26.87
		S9: (70%, 70%)	$164,150.26	$315.55	$220.89	-$69,485.26	-42.33
Payments management	Incorrect billing amounts collected	S1: (0%, 0%)	$0.00	$0.00	$0.00	$0.00	0.00
		S2: (0%, 30%)	$25,751.38	$29.79	$2.75	$1,291.81	5.02
		S3: (0%, 70%)	$66,753.21	$31.37	$3.15	-$38,524.15	-57.71
		S4: (30%, 0%)	$15,500.92	$32.17	$3.98	$12,686.26	81.84
		S5: (30%, 30%)	$26,252.30	$32.96	$2.01	$4,696.24	17.89
		S6: (30%, 70%)	$87,254.13	$35.82	$6.06	-$57,493.55	-65.89
		S7: (70%, 0%)	$42,835.48	$34.54	$6.73	-$15,020.28	-35.07
		S8: (70%, 30%)	$73,586.85	$35.33	$6.99	-$45,245.04	-61.49
		S9: (70%, 70%)	$114,588.69	$36.13	$7.26	-$85,720.26	-74.81
Customer order management	Customer orders delivered to wrong locations	S1: (0%, 0%)	$0.00	$0.00	$0.00	$0.00	0.00
		S2: (0%, 30%)	$5,527.79	$7.03	$1.34	$1,500.87	27.15
		S3: (0%, 70%)	$12,731.51	$7.83	$3.14	-$4,904.65	-38.52
		S4: (30%, 0%)	$3,951.86	$8.23	$6.09	$4,272.05	108.10
		S5: (30%, 30%)	$8,479.65	$9.64	$8.69	$1,147.66	13.53
		S6: (30%, 70%)	$15,083.38	$10.04	$11.03	-$5,054.41	-33.51
		S7: (70%, 0%)	$10,921.01	$11.34	$17.76	$401.23	3.67
		S8: (70%, 30%)	$12,848.80	$12.85	$18.88	-$19.68	-0.15
		S9: (70%, 70%)	$19,552.52	$14.05	$20.14	-$5,520.16	-28.23
Customer services	Customer-reported issues not recorded	S1: (0%, 0%)	$0.00	$0.00	$0.00	$0.00	0.00
		S2: (0%, 30%)	$36,129.95	$1.54	$0.73	-$35,322.80	-97.77
		S3: (0%, 70%)	$84,303.21	$2.05	$1.47	-$83,718.92	-99.31
		S4: (30%, 0%)	$24,086.63	$2.93	$2.13	-$23,281.92	-96.66
		S5: (30%, 30%)	$60,216.58	$3.59	$1.61	-$58,236.58	-96.71
		S6: (30%, 70%)	$108,389.84	$4.09	$4.40	-$108,696.98	-100.28
		S7: (70%, 0%)	$56,202.14	$4.88	$8.06	-$59,383.57	-105.66
		S8: (70%, 30%)	$92,332.08	$5.36	$9.45	-$96,425.94	-104.43
		S9: (70%, 70%)	$140,505.34	$6.13	$9.53	-$143,902.49	-102.42
Managerial reports	Regulatory reports not filed	S1: (0%, 0%)	$0.00	$0.00	$0.00	$0.00	0.00
		S2: (0%, 30%)	$34,259.14	$100.56	$13.41	$52,895.03	154.40
		S3: (0%, 70%)	$79,937.99	$120.68	$35.58	$5,160.23	6.46
		S4: (30%, 0%)	$22,839.43	$160.90	$91.03	$47,025.74	205.90
		S5: (30%, 30%)	$57,098.57	$261.46	$156.81	$47,557.79	83.29
		S6: (30%, 70%)	$102,777.42	$281.58	$268.62	-$89,819.80	-87.39
		S7: (70%, 0%)	$53,291.99	$241.35	$425.15	-$237,095.09	-444.90
		S8: (70%, 30%)	$87,551.13	$341.91	$629.25	-$374,884.50	-428.19
		S9: (70%, 70%)	$133,229.99	$402.25	$897.43	-$628,414.77	-471.68
Warehouse management	Wrong stock levels held	S1: (0%, 0%	$0.00	$0.00	$0.00	$0.00	0.00
		S2: (0%, 30%)	$33,373.56	$48.56	$4.86	$10,328.19	30.95
		S3: (0%, 70%)	$147,871.64	$92.26	$12.82	-$68,431.57	-46.28
		S4: (30%, 0%)	$42,249.04	$133.05	$32.49	$58,309.66	138.01
		S5: (30%, 30%)	$105,622.60	$172.82	$55.16	$12,032.80	11.39
		S6: (30%, 70%)	$190,120.68	$187.81	$92.57	-$94,884.19	-49.91
		S7: (70%, 0%)	$98,581.10	$191.66	$142.76	-$49,679.99	-50.40
		S8: (70%, 30%)	$161,954.66	$193.84	$205.10	-$173,212.35	-106.95
		S9: (70%, 70%)	$246,452.74	$194.23	$282.97	-$335,194.77	-136.01

References cited in the Appendix

Holschuh, T., Pauser, M., Herzig, K., Zimmerman, T., Premraj, R., and Zeller, A., 2009. "Predicting Defects in SAP JAVA Code: An Experience Report," *Proceedings of the 31st International Conference on Software Engineering (Companion Volume)*, pp. 172-181.

Nakajo, T., and Kume, H., "A Case History Analysis of Software Error Cause-Effect Relationships," *IEEE Transactions on Software Engineering* 17(8), pp. 830-838.

Perforce 2014, Perforce Version Control, http://www.perforce.com/product/components/version-control Accessed on 5/2/2014

Zimmerman, T., Weißgerber, P., Diehl, S., Zeller, A., 2005. "Mining Version Histories to Guide Software Changes," *IEEE Transactions on Software Engineering* 31(6), pp. 429-445.