INSEAD

The Business School for the World®

Faculty & Research Working Paper

Product Architecture and Quality: A Study of Open-Source Software Development

Manuel SOSA
Jürgen MIHM
Tyson BROWNING
2010/53/TOM
(Revised version of 2009/45/TOM)

Product Architecture and Quality: A Study of Open-Source Software Development

We examine how the architecture of products relates to their quality. Using an architectural representation that accounts for both the hierarchical and dependency relationships between modules and components, we define a new construct, *system cyclicality*. System cyclicality recognizes component loops, which are akin to design iterations—a concept typically associated with models of product development processes rather than the products themselves. System cyclicality captures the fraction of mutually interdependent components in a system. Through a multilevel analysis of several open-source, Java-based applications developed by the Apache Software Foundation, we study the relationship between system cyclicality and the generation of *bugs*. At the system level, we examine 122 releases representing 19 applications and find that system cyclicality is positively associated with the number of bugs in a system. At the component level, we examine 28,395 components and find that components involved in loops are likely to be affected by a larger number of bugs. We find that, in order to identify the set of product components involved in cyclical dependencies that are detrimental to product quality, it is imperative to remove from consideration the architectural decisions by which components are assigned into modules: it is necessary to focus only on the patterns of dependencies among product components *without* considering how components are grouped together into modules. Our results suggest that new product development managers are likely to benefit from proactively examining the architecture of the system they develop and monitoring its cyclicality as one of their strategies to reduce defects.

Keywords: Product architecture; Conformance quality; Open-source software development; Design iterations; Iterative problem-solving; Defect proneness

1 Introduction

Previous research has studied the implications of product architecture decisions for various aspects of the firm (e.g., Baldwin and Clark 2000, Henderson and Clark 1990, Ulrich 1995). However, little attention has been devoted to understanding the relationship between a product's architecture and its performance.[1] How does the architecture of a product influence its quality? More specifically, to which features of the product architecture should managers attend during the development process if they want to minimize the number of defects? We address these questions by examining the architecture–quality relationship of Java-based open-source software products developed by the Apache Software Foundation.

We focus on the development of software applications for several reasons: they are complex systems; they exhibit fast rates of change (like fruit flies in studies of biological evolution); and they offer (through their source code) an efficient, reliable, and standardized means by which to capture the architecture of their design. Moreover, software applications typically have centralized repositories that reliably track the quality issues associated with each release.

System defects (called *bugs* in software applications) are identified when the system does not perform as specified. As part of a small subset of literature in technology management that has addressed the link between architecture and quality, Terwiesch and Loch (1999) documented a case that suggests integrated and "tight" architecture drives engineering change orders. In subsequent years, some researchers have studied the architecture of complex products to explore how the direct and indirect dependencies among components influence the propagation of design changes. This line of research suggests that change propagation can cause rework and prolong development time owing to the unpredictable nature of design changes propagating between directly and indirectly connected components (Clarkson et al. 2004, Eckert et al. 2004, Sosa et al. 2007b). More recently, Gokpinar et al. (2010) found that the connectivity of an automobile's subsystems—and the extent to which their interfaces are managed—significantly affect the subsystems' conformance quality. We contribute to this line of research by showing empirically, for the first time, that quality issues are more specifically associated with an architectural measure of the dependencies causing cyclicality. In this context we demonstrate that, contrary to common assumptions, not all dependencies among components are detrimental to quality.

The literature on computer science and information systems has examined the structure of software systems and related it to performance issues such as the time required to implement changes (Cataldo et al. 2006) and the factors that lead to refactoring of the source code (for a review, see Mens and Tourwé 2004). Moreover, the emergence of open-source software products has sparked both theoretical and empirical research on open-source development (von Krogh and von Hippel 2006).

[1] We use the term "product" in a broad sense to refer both to hardware and software systems.

MacCormack et al. (2006, 2008a), in particular, explored architectural differences between open-source and closed-source development of large systems. Also, MacCormack et al. (2008b) examined connectivity patterns of a successful commercial software product to study how product designs evolve over time; they found evidence suggesting that tightly connected components are harder to remove, to maintain, and to change. Other researchers (e.g., Briand et al. 2002, Fenton and Neil 1999) have investigated various determinants of defect proneness in software applications. Koru and Liu (2007) found evidence that the distribution of defects across software components follows the Pareto law (more than 80% of the defects affect less than 20% of the components). A significant amount of research in this community has focused on the relationship between the number of defects and the size of the software components (Basili and Perricone 1984, Fenton and Ohlsson 2000, Koru and Tian 2004, Koru et al. 2008). For instance, Koru et al. (2008) showed that the number of defects in a system increases at a slower rate than the size of software components. Finally, a stream of research in the information system (IS) community has focused on developing metrics to assess various performance aspects of the product, with special attention given to product complexity (Briand et al. 1998, Briand et al. 1999, Chidamber and Kemerer 1994, Henry and Selig 1990, McCabe 1976). This line of research suggests various approaches to measuring such complexity either as an internal property of the components that form the product (Briand et al. 1998, McCabe 1976, Stein et al. 2005) or as a function of the way components are coupled to each other within the product (Briand et al. 1999). The underlying rationale behind such research is that product complexity is a proxy for the cognitive complexity faced by the development team, which in turn can affect external attributes of the product such as reusability, maintainability, or defect proneness (Card and Glass 1990, Henry and Selig 1990, Martin 2002). Although these studies provide empirical evidence that the likelihood of defect proneness can be predicted by examining the source code, they do not examine how the architectural arrangements of software components influence the generation of defects. Yet Fenton and Neil (1999) and Briand et al. (2002) explicitly recognized the need to improve our understanding of architectural factors as a predictor for software quality, which is the intention of this paper.

Thus we focus on the link between a product's architectural properties and its quality. By integrating the methods that are used in new product development (NPD) and engineering design to analyze product and process architectures and to study problem-solving dynamics with the literature on defect proneness in information systems, we uncover a novel architectural property that we call *system cyclicality*. System cyclicality is associated with the risk of generating defects in software products. In this paper, we argue that it is not the *amount* of dependency between product components but rather the *fraction of components* involved in cyclical dependencies that is positively associated with the generation of bugs; the reason is that cyclicality critically influences the risk of developers generating defects as they design, build, and test systems in an iterative fashion. This finding not only offers an important contribution to the academic literature on new product development (on design iterations) and information systems (on defect proneness) but also has important managerial

implications. Our results inform managers about the need to identify, (re)structure, and manage the group of product components affected by cyclical dependencies. We especially advocate building an understanding of the actual architecture as it emerges from the product itself, rather than the architecture as intended by the design team, because it is the former architecture within which bugs flourish.

We test our core proposition by examining empirically the relationship between system cyclicality and bug generation in a sample of open-source applications developed by the Apache Software Foundation. We conduct our analysis at both system and component levels. At the system level, we examined 122 releases representing multiple generations of 19 distinct Java-based applications. At the component level, we examined more than 28,000 components of a large subset of our system-level data corresponding to 111 releases of 17 applications.

2 Linking Product Architecture and Quality

We start by discussing in this section the literature on the architecture of complex systems and how the notion of architecture applies to software products. Then we review the various representations used to capture the architecture of complex products and discuss our choice to represent the architectures of software applications. Next we argue that, by taking an information-processing view of software architecture—a view that is typically associated with problem solving in new product development—we are able to uncover an important architectural property of software applications: system cyclicality. Finally, we develop our arguments to hypothesize a positive relationship between system cyclicality and the number of product quality issues (bugs).

2.1 The Architecture of Complex Systems

The architecture of a designed system is determined during its design process through both problem decomposition and solution integration (Alexander 1964, Simon 1996). The question of what design principles should guide the development of "good" system architectures has been at the source of a long stream of research on management (Simon 1996) and design (Stevens et al. 1974). Simon suggested that complex systems (whether physical or not) should be designed as hierarchical structures consisting of "nearly decomposable [sub]systems" (1996, pp. 197f), with strong interfaces within subsystems and weak interfaces across subsystems. This criterion is consistent with the notion of modularity, which suggests that modular designs create options for the organization to enable the evolution of designs (Baldwin and Clark 2000). More importantly, the architecture of products—as defined by the way components interact both within and across subsystems—is intimately related to how problem solving evolves in NPD organizations (Mihm et al. 2003, Smith and Eppinger 1997b) and to how development actors interact when designing new products (Cataldo et al. 2006, Henderson and Clark 1990, Sosa et al. 2004, Sosa 2008). However, we are still learning the specific mechanisms by which architectural patterns affect an organization's ability to produce high-quality products (Gokpinar et al. 2010).

Previous research in engineering design has developed methods of formally analyzing the architecture of complex products by studying how their components interact to provide system functionality. More specifically, this line of research has modeled products as collections of interdependent components, developed methods to cluster components with similar dependencies into subsystems or modules (Browning 2001, Lai and Gershenson 2006, Pimmler and Eppinger 1994), and analyzed how patterns of component connectivity relate to design decisions (Sosa et al. 2003, Sosa et al. 2007b). This stream of work established the importance of modeling products—in early phases of the development process—as collections of networked components.

Analogously to the case hardware products, the architecture of a software application is the scheme by which its functional elements are codified into objects in the source code and the way in which these objects interact and are grouped into subsystems and layers (Parnas 1972, Parnas 1979, Shaw and Garlan 1996). In order to analyze the architecture of a software system, we examine its *source code* because it codifies the system's design (MacCormack et al. 2006): the dependency structure of the source code (i.e., the way in which its components exchange information) precisely specifies the system's functionality. In this sense, the source code captures the "process" (or "recipe") that determines how the system works. In addition, the source code's structure is a good proxy for the dependencies among design activities associated with development of the components that constitute the system. That is, if component *X* depends on component *Y* then the designing, building, and testing of component *X* is conditioned by the design of component *Y* (Gokpinar et al. 2010, Smith and Eppinger 1997b, Sosa et al. 2004). It is important to recognize that the set of software components codifies a *process* because this recognition facilitates our departure from the traditional methods used to analyze product architectures and leads us to use a novel approach derived from analyzing process architectures. We therefore analyze software architectures via techniques traditionally used to analyze iterative problem solving in new product development (for a review, see Browning 2001).

2.2 Architectural Representations of Software Products

The source code of a software application consists of a collection of connected components organized into subsystems, which in turn are grouped into levels and layers (Sangal et al. 2005, Shaw and Garlan 1996). For instance, the source code of Java-based, object-oriented software applications (such as those analyzed here) is typically viewed as a collection of components called *Java classes* that are grouped into modules, which in turn are arranged in a hierarchical manner (Martin 2002). Two basic concepts characterize a "good" architecture (design structure): cohesion and coupling (Stevens et al. 1974). *Cohesion* refers to the internal consistency of each software component or module, whereas *coupling* pertains to the strength of the dependencies among components or modules. In general, good source-code design maximizes cohesion and minimizes coupling (Chidamber and Kemerer 1994, Martin 2002). Such a general principle of software development is consistent with Simon's (1996) nearly decomposable view of system design. However, we argue that exploring the features and effects of a system's architecture requires that we understand how components (and the modules into

which they are arranged) interact. To make our discussion less abstract, we refer to the example of Ant version 1.4 (hereafter Ant 1.4), one of the applications we studied.

In a manner that is consistent with the general approach to the design of complex systems, software designers typically organize the source code of their applications into hierarchically arranged modules (Martin 2002, Sangal et al. 2005, Shaw and Garlan 1996). For instance, Ant 1.4 may be decomposed into two levels of nested modules. At the first level of decomposition, Ant 1.4 consists of five modules—two of which are further divided into a few lower-level modules. In addition, software products are often designed in layers to provide a coherent "command and control" structure such that components in higher layers can "call" (depend on) components in lower layers but preferably not vice versa. That is, modules located at the bottom of the decomposition serve as platforms for the modules built on top. These *layers* are defined by the system architect's design rules (Baldwin and Clark 2000, Sangal et al. 2005).

This view of the hierarchical arrangement of components into nested modules is the dominant one of system architects (Shaw and Garlan 1996, Simon 1996). One disadvantage of such a hierarchical perspective is that it overlooks the role of dependencies among components. Yet as we will discuss later, determining the architectural properties that influence the generation of bugs makes it advisable to consider not only the components' hierarchical arrangement but also their interdependencies. Dependencies among software components are formed by the "calls" made by one component to another. To represent dependencies, both within and across modules and layers, we use a design structure matrix (DSM) representation (Browning 2001). A DSM is a square matrix whose diagonal cells represent N components and whose off-diagonal cells indicate their dependencies. Several researchers have used the DSM representation to capture the architecture of complex products, both hardware (Sharman and Yassine 2004, Sosa et al. 2003, Sosa et al. 2007b) and software (MacCormack et al. 2006, Sangal et al. 2005, Sosa 2008, Sosa et al. 2007a, Sullivan et al. 2001). However, in contrast to previous work, we analyze our DSMs by treating the product components like activities in a process.

Note that the modeler chooses where to end the lowest level of decomposition (i.e., the component level); we stop at the "class" level,[2] although we could decompose our analysis to the level of methods and data members and even to lines of code. Three main arguments led us to model software architecture at the class level. First, classes tend to provide a set of common functionality (e.g., a set of low-level mathematical functions) that is maintained as one cohesive piece of software, often in a single source file by a single author. Second, the main attributes of the architecture are apparent at the class level, making further decomposition unnecessary for our purposes. Third, this level of decomposition is consistent with previous work focused on representing software architectures (e.g., MacCormack et al. 2006, Sangal et al. 2005). Thus, for the purposes of our

[2] Our data set contains only Java applications, wherein files and classes are typically the same except for "inner classes" (classes within classes), which we do not consider explicitly.

analysis, we treat each Java class as an ”atomic” component of the software architecture.

Figure 1 shows a “flat” DSM representation (i.e., temporarily ignoring hierarchical levels) of Ant 1.4, which has 160 components with 676 dependencies among them.[3] We use the convention whereby an off-diagonal mark in cell (i, j) indicates that the Java class in column j depends on the Java class in row i. Combining the hierarchical and the traditional DSM view yields the “hierarchical” DSM (Figure 2), which consists of the flat DSM overlaid with the membership of components in modules and layers. This arrangement determines the sequencing of components in the DSM: components that belong to the same module are grouped together (e.g., the 16 components that form the “types” module are adjacent). Within each module, components may be sequenced in any arbitrary way. (In the absence of a predefined criterion, components within modules are ordered alphabetically using the names of the Java classes.) Because the flat and hierarchical DSMs are sequenced identically, their patterns of dependencies are the same. The only difference is that the hierarchical representation allows us to distinguish inter- and intramodule dependencies.[4]

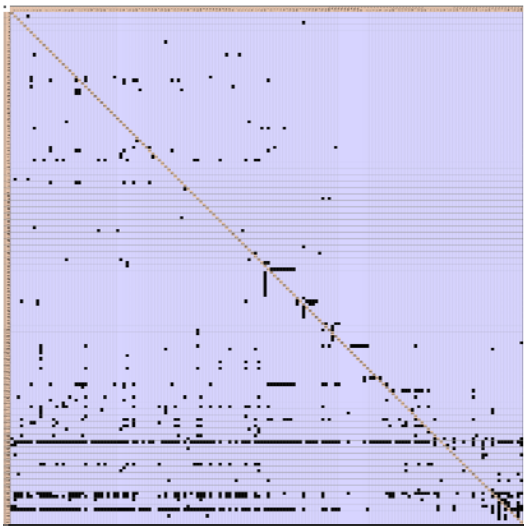

Figure 1: Flat DSM representation of Ant 1.4

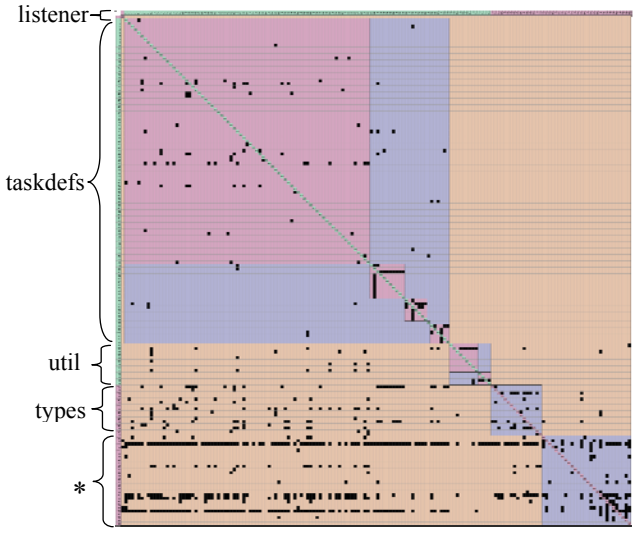

Figure 2: Hierarchical DSM representation of Ant 1.4

2.3 Identifying Component Loops in Software Architectures

Because the notion of component loops is central to our methods and hypothesis development, we discuss here our approach to identifying them in complex systems. A key strength of either DSM is its ability to highlight cycles or loops. We define a *component loop* as a subset of components whose dependencies form a complete circuit. Now consider the various patterns of dependencies that can

[3] Specifically, we include the following types of dependencies: invocations (static, virtual, and interface), inheritances (extensions and implementations), data member references, and constructs (both with and without arguments). We include these dependencies because they are deliberately created by the developer; the vast majority of such dependencies are integral to the design of the system.

[4] We build flat and hierarchical DSM representations of each version of all the Java-based applications included in our sample by using a commercially available tool developed to build DSM representations of software applications based on their source code (see www.lattix.com for details). For Java-based applications this tool uses precompiled (“prebuilt”) code in JAR (Java ARchive) files. JAR files contain all the Java class specifications (including the dependencies among them and the subdirectory structure to which each Java class is assigned) for a given software application.

exist among several components in a system. Figure 3 shows four cases.

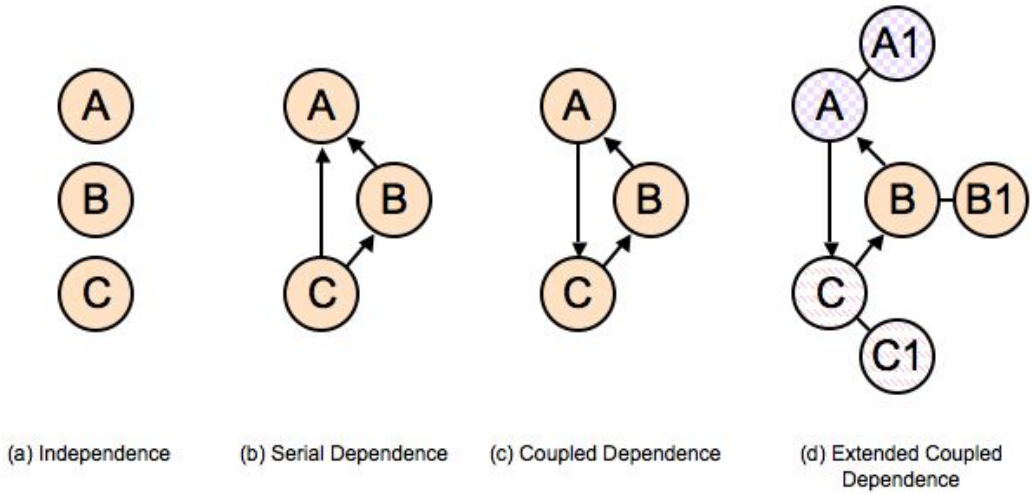

Figure 3: Four types of component relationship patterns

In case (a), the three components are independent and so data processing by any one of the components does not affect the others (barring resource constraints). This means that developers designing these components could work independently of each other. In case (b), component C provides inputs to components A and B, and component B provides data to component A. As a result, there is a serial order (C, B, and A) in which these three components must be executed and that determines the sequence in which the components should be built and tested. From a problem-solving viewpoint, developers responsible for the design of component A would probably need design data produced by the developers designing components B and C. In cases (c) and (d), components A, B, and C are involved in a component *loop* because they depend on each other in a cyclical manner. Procedures of component A depend on data processing performed by component B, which depends on data provided by component C, which in turn depends on data provided by component A. Such a cyclical dependency pattern suggests that developers designing components A, B, and C must work iteratively until they converge to a solution (Mihm et al. 2003, Smith and Eppinger 1997a, Smith and Eppinger 1997b). Intuitively, managing such cycles during the design process is complicated because testing or evaluating design changes to any of the three components will involve coordinating the impact of such changes with the other two components (since they all depend on each other). From an information-processing view of the system, cases (a), (b), and (c) represent the three fundamental patterns of dependencies (Eppinger et al. 1994, Thompson 1967). These three cases, however, assume that the components all belong to the same organizational group. Case (d) represents module membership—in addition to dependence—by showing components A, B, and C as well as (the newly added) A1, B1, and C1, which form three distinct two-component modules (that are shaded differently for visual distinction).

In the development of complex products, managers typically group together components that provide certain functionality in order to facilitate problem solving. Such groupings are formalized in

the product domain by defining product subsystems or modules (e.g., the fan subsystem of an aircraft engine, the input–output module of a software application). The managerial decision to create a module has organizational and operational implications because developers assigned to design a module's components must consider other components within the module during the design process—in other words, a module must be designed, built, and tested as an integrated unit rather than as a collection of isolated components (Martin 2002). We argue that, since the design of components A, B, and C (shown in Figure 3(d)) are considered in conjunction with other components contained in the same module (A1, B1, and C1, respectively), it follows that any time there is an update or change in the design of either A, B, or C then there is a significant risk of needing to update or change the design of other components in the same module.

In this paper we show that the presence of component loops, such as the one exhibited by A, B, and C in Figure 3(c), is positively associated with product quality issues. In addition, we explore whether components arranged into modules, as shown in Figure 3(d), result in additional negative side effects (on product quality) due to the presence of additional components being considered in conjunction with the components that are more directly involved in coupled dependence.

The concept of loops or cycles (also called iterations) is not new in the process analysis literature, where DSMs have been used to identify subsets of problem-solving activities that drive iterations (Meier et al. 2007, Smith and Eppinger 1997a, Smith and Eppinger 1997b, Steward 1981). However, our conceptualization of *component loops* is new in two ways. First, we define *system cyclicality*—a structural property—as the fraction of the system's components that are embedded in loops. Second, we identify hierarchical component loops in the presence of the constraints imposed by the architecture of components arranged into hierarchical modules.

In order to identify component loops, we start with a flat DSM representation such as the one shown in Figure 1. A mark (i, j) below the diagonal indicates a *feed-forward* dependency, where component i provides data to component j ($i < j$); similarly, a mark above the diagonal ("superdiagonal") indicates a *feedback* dependency, where component j provides data to component i. Because feedback dependencies spawn loops, feedback marks are generally undesirable in process architectures. A basic sequencing algorithm can order the DSM to minimize the number of subsets of components involved in loops. We use an algorithm that first minimizes the number of superdiagonal marks and then minimizes their distance from the diagonal (Warfield 1973).[5] Applying this to the flat DSM in Figure 1 identifies the three component loops in Ant 1.4; these are highlighted in Figure 4, where the coupled components are grouped adjacently and their dependencies are clustered by three blocks along the diagonal. Twelve feedback marks establish the three loops, which contain 6, 11, and 18 interdependent components, respectively. We call these loops *intrinsic (component) loops* because they represent the fundamental sets of coupled dependencies intrinsic to the system, analogously to

[5] This is the standard "first-cut" DSM partitioning approach used in most of the literature, however other objectives may also be used (for a review, see Meier et al. 2007).

the case of Figure 3(c). Because 35 of Ant 1.4's 160 components are involved in intrinsic loops, there is an 0.22 probability that a randomly chosen component is involved in an intrinsic loop. As we describe later, Ant 1.4 thus has an *intrinsic* (*system) cyclicality* measure of 22%.

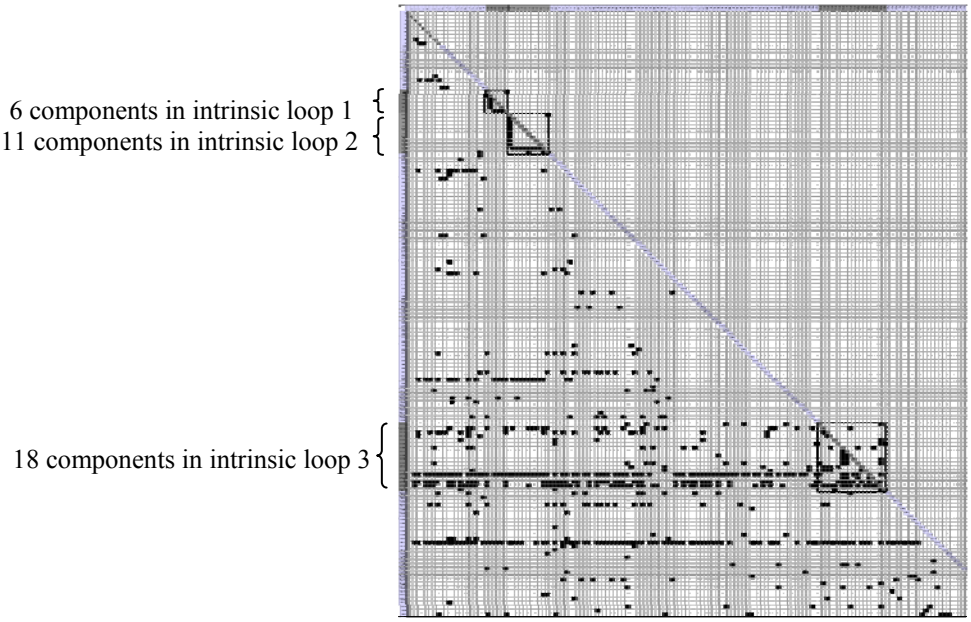

Figure 4: Sequenced flat DSM of Ant 1.4

Our use of sequencing to identify component loops distinguishes this approach from previous work in both the hardware and software product domains (e.g., MacCormack et al. 2006, Pimmler and Eppinger 1994), which has used clustering algorithms to group highly interdependent components without distinguishing between feed-forward and feedback interactions (Browning 2001). In contrast, we analyze software architecture DSMs by considering the process-like nature of the source code they represent; this analysis is analogous to DSM-based models of development processes, which identify subsets of activities involved in design iterations (Meier et al. 2007, Smith and Eppinger 1997b, Steward 1981).

		1	2	3	4	5
listener	1	.				
taskdefs	2		.			
util	3		21	.	1	2
types	4		89	1	.	5
ant.*	5	4	281	7	30	.

(a) First-level DSM

			1	2	3	4	5	6	7	8	9	10
listener	Log4jListener	1	.									
taskdefs	optional	2		.								
	compilers	3			.		2					
	rmic	4				.	2					
	*	5		1	5	2	.					
	condition	6					4	.				
util	*	7				1	20		.		1	2
	regexp	8							2	.		
types		9			15	6	68		1		.	5
ant.*		10	4		19	10	248	4	4	3	30	.

(b) Second-level DSM

Figure 5: First- and second-level sequenced hierarchical DSM of Ant 1.4

Intrinsic loops are determined from a flat DSM without any regard for their hierarchical arrangement within nested modules. To take into account a component's assigned place in the developer's organization of the code, we must require that the resequencing occur within the modules shown in the *hierarchical* DSM (Figure 2). Our goal is to minimize the number of marks above the diagonal (and their distance from it) subject to the constraint that all components within a module must be sequenced adjacent to each other. That is, we must recursively sequence the modules internally at each level, from the top (root) level down. Figure 5(a) shows the sequenced DSM of the five modules comprised by Ant 1.4 at the first level of decomposition. The off-diagonal cells in this DSM indicate the number of dependencies between the components of the modules labeling this DSM. The DSM shows that modules "util", "types", and "*" are involved in coupled dependence (i.e., these three modules form a loop). Because the modules "listener", "taskdefs", and "util" contain modules inside them, we can draw a second-level DSM (see Figure 5(b)) to show explicitly the dependency between the second-level modules. Figure 5(b) shows two loops of modules. First, observe that three of the five modules within "taskdefs" form a three-module loop; second, the three-module loop first identified on the root level in Figure 5(a) still forms a loop. In order to identify the components involved in the loops shown in Figure 5(b), we further explode the DSM to the component level and sequence components within each module (see Figure 6). The sequenced hierarchical DSM on this level continues to show the two hierarchical component loops identified at higher levels. (The algorithm used to determine a loop in a sequenced hierarchical DSM is described in Appendix A.) This approach highlights any feedback dependencies that traverse the modules and layers of decomposition laid out by the system architects. Note that, because the sequencing algorithm on the hierarchical DSM is constrained by the modules as defined by the architects, the set containing components that form a hierarchical loop may include components *additional* to those that form an intrinsic loop. As a result, the number of components involved in hierarchical loops will never be less than the number of components involved in intrinsic loops. We define a *hierarchical (component) loop* as the set of components that form the modules spanned by an intrinsic loop, analogously to the components of Figure 3(d). Hence, *hierarchical (system) cyclicality* is defined as the fraction of

system components that are involved in hierarchical component loops.

Figure 6 shows a sequenced hierarchical DSM of Ant 1.4. By examining the blocks formed along the diagonal when all the components involved in the two hierarchical loops are included, we find that they contain 151 components. The first loop includes 94 components across three modules ("compilers", "mic", and "*") that are contained within the high-level module "taskdefs". The second loop contains 57 components across four modules (the two modules that constitute "util" and the high-level modules "types" and "*"). Because 151 out of 160 total components are involved in hierarchical loops, the probability is 0.94 that a randomly chosen component is involved in a hierarchical loop (and so the system has a hierarchical cyclicality of 94%).

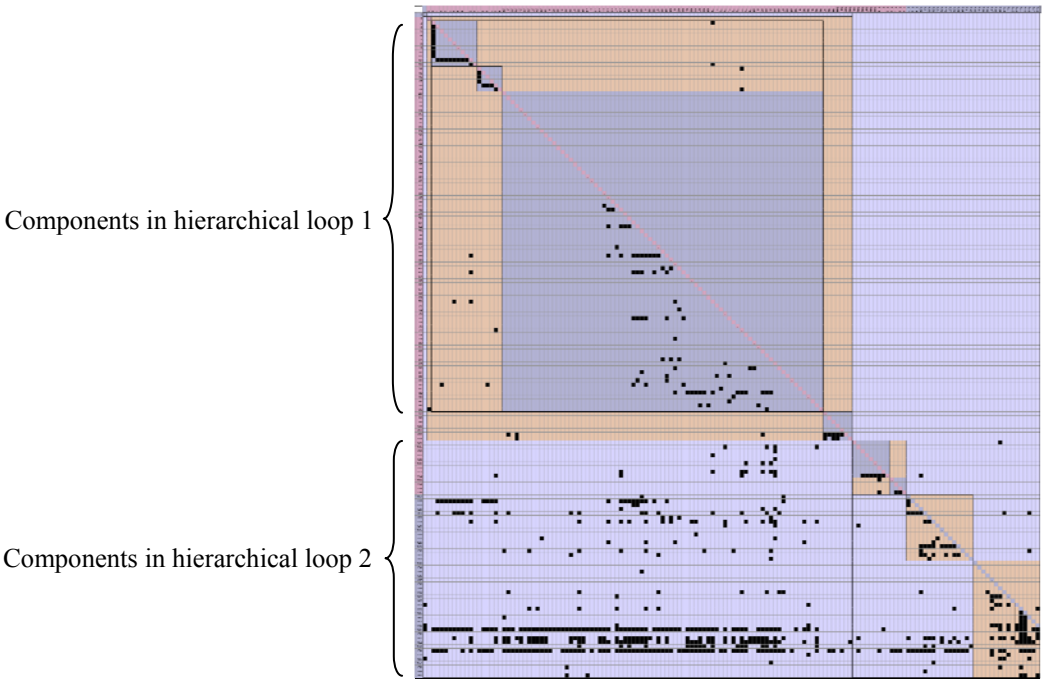

Figure 6: Sequenced hierarchical DSM of Ant 1.4

Finally, since hierarchical loops contain intrinsic loops, we define *delta (system) cyclicality* as the difference between hierarchical and intrinsic cyclicality. This means that delta cyclicality is the fraction of components involved in hierarchical loops that are *not* part of an intrinsic loop. Such components might be especially vulnerable to the results of iterative problem solving associated with intrinsic component loops—as are the components A1, B1, and C1 in Figure 3(d).

2.4 Hypotheses: The Effects of Component Loops on Quality

We argue that a system's cyclicality can significantly affect its expected number of defects. In light of the literature that addresses iterative problem solving in new product development, we expect that the presence of component loops could be an important factor driving the generation of defects because loops often complicate problem solving by introducing design iterations (Roemer et al. 2000, Smith and Eppinger 1997a, Smith and Eppinger 1997b, Steward 1981). Iterative problem solving typically corresponds to difficult and recursive problems that require making assumptions, iterating, and/or

compromising. This process may not converge easily and thus carries a higher risk of residual errors—which are not easily detected and removed during the development process itself—than does serial or parallel problem solving (Eppinger et al. 1994, Krishnan et al. 1997, Terwiesch et al. 2002). In addition, the greater the number of components involved in such iterative problems, the lower the probability of convergence to a feasible solution (Mihm et al. 2003), which also increases the risk of embedding defects in the system. And because these embedded defects are difficult to detect and fix, they may propagate and cause secondary errors in other components, thereby increasing (indirectly) the likelihood of developing a defective product (Clarkson et al. 2004, Terwiesch and Loch 1999).

Previous literature has suggested that iterative dependencies may be detrimental to the development of complex systems (e.g., Clark and Fujimoto 1991), and software development practitioners typically recommend avoiding coupled dependencies (Martin 2002). Moreover, previous work has modeled the impact of design iterations on the cost and lead time of product development (Roemer et al. 2000, Smith and Eppinger 1997a, Smith and Eppinger 1997b). Yet the impact of design iterations on product quality has not been modeled by this stream of research, and neither is there any substantial empirical evidence of this impact. Why and how do design iterations lead to product defects? Design iterations are characterized by feedback dependencies in the design process, which typically force the team to revisit assumptions and tasks that were once considered finished; thus, the arrival of new or updated information is likely to lead to design rework (Clark and Fujimoto 1991). Yet carrying out design iterations, even if necessary, is usually perceived as undesirable because doing so is costly and time-consuming (Roemer et al. 2000). Hence, the risk of significant design rework triggered by feedback interactions can, in turn, increase the risk of leaving unfixed problems that are later discovered as defects. Even if the organization identifies the components involved in a cycle, the organization is likely to struggle to define an order of development tasks (e.g., the sequence in which to build component prototypes or to compile software modules) because in the case of cycles there is no predefined sequence of design activities. This increases coordination costs and the risk of overlooking design details, which can lead to defects.

Our arguments apply also to the development of software products. In software development, although many bugs are discovered and fixed during programming, many remain in the testing and final versions of a system and are discovered by its users. We focus on the architectural determinants of this latter type of defects—those that have been reported in the formal bug-tracking systems. In object-oriented software development, "objects" (components) are designed and "connected" to achieve the various functions required from the application. The design of components is usually assigned to various developers. The typical software development approach involves the constant repetition of "design, build, and test" tasks (MacCormack et al. 2001). This requires the independent designing and building of components as well as their integration with other interdependent components for testing and evaluation. As a result, dependencies among components dictate how components are designed, built, and tested. Component dependencies determine the cognitive

complexity of the problem-solving approach and thus the amount of coordination effort required among developers as they design and assemble entirely functional pieces of software. Because components involved in intrinsic loops (such as the one shown in Figure 3(c)) increase the complexity of the problem and require a higher degree of coordination during the execution of repeated design, build, and test iterations, such components are likely to be at higher risk of suffering coordination pitfalls and this will likely lead to more bugs. There are implications at both the system and component levels. At the system level, the larger the fraction of components involved in intrinsic loops, the greater the likelihood that most of the designers will be involved in iterative problem solving associated with intrinsic loops, which increases the risk of generating bugs. At the component level, if a significant number of bugs are generated by intrinsic loops, then the components involved in those loops are more likely to be affected by bugs. Thus we have our first hypotheses as follows.

H1a: The larger the fraction of components involved in intrinsic loops, the greater the number of bugs associated with the system.

H1b: The components involved in intrinsic loops are more likely to be affected by bugs than are other components.

Our central hypothesis (H1) predicts that the presence of intrinsic loops increases the expected number of bugs in the system. However, because the source code is organized into nested modules, it is important to explore the possibility that hierarchical loops, which account for additional components involvement in loops due to their module membership, may have an additional effect on the product's quality.

We argue that the additional components in hierarchical loops are likely to be more vulnerable than other components in the product (not involved in any loops) to the effects of iterative development associated with intrinsic loops and therefore will likely carry an additional share of defects. To illustrate this point we consider module M, which contains some components that are involved in an intrinsic loop (either within module M or across modules). If we assume that components within module M are largely considered in conjunction during the development process (i.e., they are designed, built, and tested together), then the development of all components in module M are at higher risk of being disrupted by the iterative changes associated with the intrinsic loop than are other components in the product (which are not involved in any component loop). Developers responsible for the development of such modules are hence likely to face a more challenging iterative problem solving—not only because of the lack of precision and stability of the information exchanged concerning their designs (Terwiesch et al. 2002) but also because of the lack of planning associated with iterative information exchanges (Pich et al. 2002, Sommer and Loch 2004).

Therefore, at the system level, larger hierarchical loops can lead to a larger number of bugs because they contain "additional" components linked via module membership to components involved in intrinsic loops. Hence, the designing, building, and testing activities of these additional components are at risk of being disrupted by the intrinsic loops in which they are embedded. Since the

development activities of these additional components are particularly vulnerable to disruptions originating in intrinsic loops, it follows that such additional components are more likely to be affected by bugs than the components outside any loop. This leads to our second hypotheses, which apply to both the system and component levels.

H2a: The larger the fraction of additional components in hierarchical loops (that are not part of intrinsic loops), the greater the number of bugs in the system.

H2b: Components involved in hierarchical loops that are not part of intrinsic loops (i.e., additional components) are more likely to be affected by bugs than are components outside any loops.

3 Empirical Study: The Apache Software Foundation

To test our hypotheses, we studied open-source, Java-based software applications from the Apache Software Foundation (http://www.apache.org/), one of the largest, best-established, and most widely studied open-source communities of developers and users who share values and a standard development process (Roberts et al. 2006). The Apache Software Foundation has a "desire to create high quality software that leads the way in its field." We examined all the Java-based applications developed by Apache, focusing on Java because (1) it is one of the most open and widely used object-oriented programming languages and (2) its source code captures component dependencies in an explicit, structured manner. This minimizes the risk of component dependencies being "masked" in the source code and not appearing until runtime.

Initially, we identified 69 Java-based development projects at the Apache Software Foundation in mid-2008. This provided our initial database. An effective examination of the causal relationship between architectural characteristics and quality requires a longitudinal data set, so we reduced ours to the 37 applications for which we could obtain data for successive major releases. That is, we discarded 32 projects because they had a limited history of only one major release or only a few minor releases. From the 37 remaining applications, we selected those for which we could access, for successive major releases, their precompiled ("prebuilt") code in JAR files (to codify product architecture features), their original source codes (to measure some specific product-related attributes such as lines of code), their bug reports (to determine the number of bugs), and their release notes (to determine the innovative features and other control variables). These filters left us with a set of 122 releases representing 19 applications with an average of 6.4 major releases (or versions) each.[6] At the component level, we additionally required the existence of versioning management tools (e.g., Subversion, which creates "SVN repositories"); this resulted in a sample of 28,395 components with complete data.

We compiled data from five sources. First, we examined the *Bugzilla* and *Jira* bug-tracking systems of the Apache Software Foundation to obtain the bugs associated with each release. Each of

[6] The application size in our samples ranged from 29 to 1,282 components ($\mu = 297$, $\sigma = 212$).

these systems allows users and developers to enter bug reports, which are classified in terms of their potential severity and processed by the development team in a structured way. This process applies to all bugs that are not fixed by a developer during initial programming. The databases of these bug-tracking systems thus record the status and resolution of each bug associated with any release. We developed a web crawler to automate the gathering of data on bugs. Second, we downloaded the precompiled versions of the major releases of each application (available as a JAR file) from the Apache archives and/or the application's website, selecting the versions that were considered to be major releases. We did not normally use minor releases because they typically involve relatively small changes. We used a commercially available software application developed by Lattix, Inc., to translate the structure of the source code (as captured in the JAR file) into DSM representations such as those shown in Figures 1 and 2. Third, we also downloaded the original source code for each of the applications in our data set. Because the correspondence of Java classes to files is almost one-to-one, this step involved locating and downloading more than 120 source packages and examining more than 28,000 source-code files. Accessing original source-code files was important for measuring various dimensions of an application's complexity at both the component and system levels. Fourth, for the component-level analysis, we consulted the SVN repositories to establish a link between the individual bug and the component(s) that it affected, as the version control tool specifies the component(s) altered during the fixing of bugs. Hence, for our purposes we used all the bugs reported in the bug-tracking system that had been fixed or were in the process of being fixed. Since the SVN repositories contain data about timing and authorship, we were able to develop and implement a web crawler to search the repositories for each bug in the bug-tracking system. Finally, we consulted the release notes of each version of all the applications in our sample to find data on newness, age, and other important controls.

Because our hypotheses are formulated at both the system and component levels, we now carry out two separate analyses. At the system level, we test how the propensity of a system for component loops relates to the number of bugs associated with it. At the component level, we test whether components involved in component loops are more likely to be affected by bugs that are discovered after the application is released.

4 System-level Analysis

4.1 Variables

4.1.1 Dependent Variable: Number of Bugs per System

Number of bugs associated with version s of application i (y_{is}). Our main dependent variable counts all the bugs that have been formally identified and attributed to version *s* of application *i*. The identification of a bug is carried out by developers or users (with confirmation by developers). Hence, this variable is a proxy for the number of actual *defects* embedded in version *s* of application *i* after its architecture is established and beta testing starts. As mentioned previously, we use the *Bugzilla* and *Jira* bug-tracking systems as the data sources to quantify this variable. From the complete list of bugs

entered into these systems, we discard any items that could not be verified as actual bugs by developers (classifications: "WORKS_FOR_ME" or "INVALID" for *Bugzilla* and "Cannot Reproduce" or "Not A Problem" for *Jira*). We also discard any bugs that the developers consider to be duplicates of bugs already registered in the system (classification "DUPLICATE" for both *Bugzilla* and *Jira*). Attribution of a bug to a code version is determined by the classification in the system (according to data field "Affected Versions").

4.1.2 Independent Variables

Our key predictor variable is the extent to which the architecture of version *s* of application *i* contains intrinsic and hierarchical component loops. Because we can identify loops in either the presence or absence of the constraints imposed by the hierarchical assignment of components to modules, we define various types of system cyclicality as follows.

- *Intrinsic cyclicality* ($P_{I,is}$) is the probability that a randomly chosen component in version *s* of application *i* belongs to an *intrinsic* loop—that is, the ratio of components involved in loops in the flat DSM ($C_{I,is}$) to the total number of components (N_{is}):

$$P_{I,is} = C_{I,is,} / N_{is} \tag{1}$$

- *Hierarchical cyclicality* ($P_{H,is}$) is the probability that a randomly chosen component of version *s* of application *i* belongs to a *hierarchical* loop. This measure is a function of the number of components that are involved in loops determined while maintaining the constraints of the subsystems and layers used by programmers to organize their code ($C_{H,is}$). To identify $C_{H,is}$, we count the number of components in loops in the sequenced hierarchical DSM, such as the ones shown in Figure 6. Hence,

$$P_{H,is} = C_{H,is} / N_{is} \tag{2}$$

- *Delta cyclicality* ($P_{D,is}$) is the fraction of additional components involved in a hierarchical loop but not an intrinsic loop:

$$P_{D,is} = P_{H,is} - P_{I,is} \geq 0 \tag{3}$$

Finally, to understand further the relationship between intrinsic loops and bug generation, we consider an alternative predictor variable that measures the absolute size of intrinsic loops. *Average intrinsic loop size* ($AVG\_CSIZE_{I,is}$) is the number of components involved in intrinsic loops divided by the number of intrinsic loops ($NL_{I,is}$):

$$AVG\_CSIZE_{I,is} = C_{I,is} / NL_{I,is} \tag{4}$$

4.1.3 Control Variables

We include two sets of control variables. First, we control for exogenous, nonstructural features of the application that are likely to affect the generation of bugs. Second, we control for structural characteristics of the components and their relationships so as to test precisely whether and how cyclicality might influence the generation of bugs.

Nonstructural Controls

- *Age of application at version s* (AGE_{is}). The age of the application is measured by the number

of days since its development began. This assumes that the application is officially "born" on the date of the first release available (as indicated in the release notes) and then ages with successive releases. The cumulative time between releases is likely to increment both the complexity of and knowledge about the architecture, which are factors that are likely to affect the generation of bugs.

- *Days since last release* (*DAYS_BEFORE*$_{is}$). The time between successive releases varies within and across applications, so it is important to control for the time span between the previous release and version *s*. The longer this time, the higher the probability that changes have been introduced that could affect the generation of bugs.
- *Days to next release* (*DAYS_AFTER*$_{is}$). This is the time between the current version *s* and the next release (*s* + 1). The longer this time, the higher the probability that bugs will be discovered, because it corresponds to when the application is most actively scrutinized by testers and users.
- *Newness of application at version s* (*NEWNESS*$_{is}$). Both *new features* (added functionality) and *incremental improvements* (modifications to existing functionality) add uncertainty and complexity to the structure of an application. Implementing these types of changes is likely to introduce unforeseen perturbations and thus bugs. Using information from the release notes, we capture both the overall number of new features and the incremental improvements as measures of the newness of version *s*. "New features" and "improvements" in version *s* are determined by the project's "committers", who are responsible for authorizing the release. Although release notes differ in format from application to application, they all list the new features and improvements made to each version. Our measure counts the items in such lists.
- *Implicit bugs* (*IMPLICIT_BUGS*$_{is}$). Some bugs reported in the bug-tracking system are not explicitly assigned to a specific version. Nonetheless, they still exist. We control for the existence of these "implicit" bugs because their discovery may influence the discovery of bugs that are explicitly assigned to version *s*. We assign a bug that is not explicitly associated with any version to the version that was most recently released when the bug was entered into the bug-tracking system.
- *Number of nominal modules* (*NUM_NOM_MODULES*$_{is}$). The application source codes in our data set are complex systems formed by interrelated components. To manage this complexity, developers group the components (Java classes) into modules (or packages) that are further hierarchically grouped into nested modules. Typically, modules aggregate components that collectively perform certain functions. Such a grouping is likely to affect the cognitive ability of the team to understand the architecture of the source code, so it may influence their propensity to generate bugs. Note that this measure counts only the number of component-based modules, not any nested modules containing only other modules.
- *Average module abstraction deviation* (*AVG_MODULE_ABSTRACTION_DEVIATION*$_{is}$). We define "module abstraction deviation" as the source code's lack of adherence to the *stable-abstraction* principle of agile software development. This variable assesses developers' ineffectiveness at

grouping their components into modules according to a salient and measurable principle of agile software development: *A module should be as abstract as it is stable* (Martin 2002).[7] We use the normalized metric of "distance" originally proposed by Martin (1994), which captures the design team's inability to assign abstract classes into stable modules (Martin 1995, Chapter 3). As indicated by Martin (1994, p. 8) this metric measures the "conformance of a design to a pattern of dependency and abstraction that [based on experience is considered] a good pattern." The variable ranges from 0 to 1, with higher values indicating greater deviation from the recommended balance between stability and abstraction in a module's architecture. We use the LDM tool (developed by Lattix) to calculate this metric directly from the JAR files. Note that this is a module-level variable, which is averaged across all modules in the application to derive an application-level variable.

- *Average cyclomatic complexity* (AVG_CC_{is}). Cyclomatic complexity is the minimum number of linearly independent paths in the control flow graph of a software program (McCabe 1976). The *control flow graph representation* of an application is different from the *class-call-based dependency structure representation* that we use: in a control flow graph, the nodes are basic blocks of code (i.e., sets of instructions executed in a predefined sequence) and the edges represent jumps between basic flows (Legard and Marcotty 1975). For instance, a simple program with only a few sequential calculations—each with only one independent path from beginning to end—has a cyclomatic complexity of 1. However, if the program includes a simple "if … then" statement, then its cyclomatic complexity increases to 2 because the "if" statement creates two independent paths (depending on whether or not the condition is satisfied). Cyclomatic complexity is used to identify the methods[8] of a program that would be harder to test and maintain as a function of the number of independent paths that could be executed by such a method (Henry and Selig 1990, McCabe 1976). Observe that, for the Java applications we analyze, cyclomatic complexity is determined at the method level—in other words, below the component level. We use a readily available tool called JHawk (www.virtualmachinery.com) to examine the source codes of all the Java classes in our sample; this enables us to calculate the cyclomatic complexity of all methods in almost all Java classes in our sample.[9] At the application level, we sum the cyclomatic complexity of all methods in version *s* of

[7] More stable modules contain a large fraction of components on which other components in other modules depend. More abstract modules contain "abstract" components (also called "interface classes"), which serve as the conduit for any dependencies to extra-module components. Intuitively, abstract components act as "standardized interfaces" (Ulrich 1995). Hence, changing components in a stable module that contains abstract Java classes would not affect components in other modules.

[8] A *method* is a self-contained collection of programming instructions that typically include variable instantiation and control flow statements, such as "if … then" and "while … do" statements. A Java class can contain several methods.

[9] We could not calculate the cyclomatic complexity (nor count the lines of source code) of 4% of the Java classes in our sample because the JHawk tool could not parse their source code files properly; these files contained "reserved Java keywords" as variables. Such keywords were not keywords when the source code files were originally created, but they have been declared Java keywords subsequently. Omitting a few Java classes does not pose a threat to our analysis because our averaging cyclomatic complexity over all the methods in the application smoothes out the effect of the missing classes.

application *i* and then divide by the number of methods to obtain the average cyclomatic complexity at the application level.

- *Source lines of code* ($SLOC_{is}$). The overall complexity of a system is a function of the amount of information it carries. One of the most widely used metrics to capture the raw complexity of a software application is the number of its source-code lines[10] (Card and Glass 1990, Henry and Selig 1990, Sommerville 2007, Zhang and Baddoo 2007). We measure the number of source lines of code (in "kilolines") with the JHawk tool, which directly counts the number of statements (excluding comments) in each method of each component of version *s* of application *i*.

Structural Controls

Structural control variables are important to consider in our analysis for two reasons. First, structural variables measure the complexity of an application based on how its components interact. Second, our key predictor variable (cyclicality) is a structural variable, so it is crucial to control for other possible types of structural variables. Source code-level metrics such as lines of code or cyclomatic complexity, assume that components (either Java classes or the methods within them) are independent and therefore assess the complexity of the application simply by aggregating the internal complexities of the individual components. But a system is much more than the mere aggregation of its parts, so we include the following two important structural control variables.

- *Propagation cost* ($PROPAGATION\_COST_{is}$). Complex systems are designed as a set of components, some of which need to be connected to ensure that the whole system operates properly. The presence *or* absence of direct and indirect dependencies can create defects. On one hand, direct and indirect dependencies among components provide an avenue for propagating changes—some neither intented nor planned—and thereby lead to defects (Clarkson et al. 2004, MacCormack et al. 2006, Sosa et al. 2007b). On the other hand, the absence of certain dependencies across components might inhibit product functionality (Martin 2002, Sosa et al. 2007b). In the absence of a clear prediction, we control for the overall connectedness of the components in version *s* of application *i* by calculating its propagation cost (MacCormack et al. 2006). *Propagation cost* is the probability that two randomly chosen components are connected either directly or indirectly (through intermediary components). We determine propagation cost as the density of the binary *visibility matrix* (**V**) of the system. We remark that **V** is a square, binary matrix (similar to the DSM) whose nonzero cells (v_{ij}) indicate that component *i* is connected to component *j* either directly or indirectly via any number of intermediary components. The matrix **V** is obtained by raising the DSM (**D**) to successively higher powers via Boolean multiplication until the number of empty cells in the resultant matrix is stabilized (MacCormack et al. 2006, Sharman and Yassine 2004).
- *Number of component loops* (NUM_LOOPS_{is}). Because our key independent variables do not explicitly control for the number of component loops present in the source code, we include a control

[10] In the data set analyzed here, this metric does indeed exhibit pairwise correlation levels exceeding 0.94 with the number of components and the number of calls across components.

for it whose value depends on whether we are considering intrinsic or hierarchical loops. For instance, the number of intrinsic loops in Ant 1.4 is 3 (Figure 4) and the number of hierarchical loops is 2 (Figure 6). On average, the number of hierarchical loops is smaller than the number of intrinsic loops because a hierarchical loop may contain several intrinsic loops.

4.2 Analysis: Predicting the Number of Bugs in a System

Table 1 gives descriptive statistics and correlations among the variables included in our analysis. On average, 82 bugs are explicitly associated with each release.

Our dependent variable is the number of bugs. Several features of the data make statistical analysis a nontrivial task. Because our dependent variable exhibits skewed count distributions (which take nonnegative values only), standard ordinary least-squares regressions can lead to inefficient and biased estimates. This issue can be dealt with by using Poisson-like regression models developed explicitly to model the count nature of dependent variables (see Cameron and Trivedi 1998). Because the variance of our dependent variable is significantly larger than its mean, negative binomial regression models provide a more accurate estimate of the standard errors of the coefficient estimates (Cameron and Trivedi 1998, Hausman et al. 1984). We therefore estimate a model of the form (Cameron and Trivedi 1998, p. 279):

$$E[y_{is} \mid x_{is}, \alpha_i] = \alpha_i \exp\{x'_{is}\beta\} \tag{5}$$

That is, our regression models predict that the expected number of bugs in version s of application i depends exponentially on a set $\{x_{is}\}$ of linearly independent regressors. The exponential form of our model ensures that the dependent variable is always greater than 0.

The β-coefficients shown in Table 2 are estimated by fitting the model to the data. The coefficient β_j equals the proportionate change in the expected mean if the jth regressor changes by one unit. A significantly positive (negative) β_j coefficient indicates that, all else being equal, an increase (decrease) in regressor j increases (decreases) the expected number of bugs. Of particular interest are the β-coefficients for our key independent variables. For example, a coefficient $\beta_{INTRINSIC\_cyclicality}$ significantly greater than 0 would indicate that the greater is the intrinsic cyclicality in version s of application i, the greater is the expected number of bugs. This result would be in line with hypothesis H1a.

The α_i are application-specific effects, which can be either fixed or random. These effects permit observations of the same application to be correlated across versions, thereby building serial correlation directly into the model. In a fixed-effects model, the α_i absorb time-invariant, unobserved, application-specific features. By including fixed effects in this manner, we effectively control for any unobserved factors such as the "culture" or "baseline experience" of the development team associated with each application—given that these factors are much more likely to differ across applications than to change over successive releases of the same application. For the random-effects model, the α_i are independent and identically distributed random variables that can be estimated by assuming a

distribution for α_i (typically a gamma distribution). We report estimates based on the fixed-effects model that are consistent with the more efficient random-effects estimates of models that pass the Hausman specification test (Hausman et al. 1984). Finally, because software development technologies may change significantly from year to year and such developments might affect the generation of bugs across all of the applications, we include indicator variables for the year of each release.

Table 2 provides the coefficient estimates of the models predicting the expected number of bugs. Model 1 includes the nonstructural control variables. This model shows that the effect of "time to next release" is positive and significant, indicating that—as expected—the longer the time between releases, the greater the number of bugs. Model 1 also suggests that systems with higher average cyclomatic complexity (i.e., an application with methods that, on average, have many independent paths in their control flow charts) are likely to exhibit a larger number of defects. This is consistent with the IS literature that suggests cyclomatic complexity is an important predictor of the effort required to test and maintain software applications (Henry and Selig 1990, McCabe 1976). Also consistent with the IS literature (e.g., Koru et al. 2008) is our controlling for the product's raw complexity by counting the total number of lines of source code (*SLOC*), which exhibits a negative coefficient that becomes significant in the presence of cyclicality measures (see Models 3 and 6). Model 2 includes our main structural control variable (propagation cost) with a positive but not significant coefficient ($p < 0.245$), which seems to suggest that the product's connectedness is not necessarily a significant determinant of system defects. This result can be understood in terms of our component-level analysis, where we will show that dependencies' *directions* matter more than their overall connectedness.

Model 3 tests H1a, which predicts that intrinsic cyclicality is positively associated with the expected number of defects. The positive and significant coefficient for intrinsic cyclicality provides empirical support for our core hypothesis at the system level ($p < 0.051$). It is of interest that we obtain this result even after controlling for propagation cost, which is highly correlated with our main predictor ($\rho = 0.83$). Observe that variance inflation factors (VIFs) for all variables included in Model 3 do not suggest any multicollinearity issues (all VIFs are below 6.3, and the mean VIF is 2.62). In addition, we observe that the coefficient for intrinsic cyclicality remains positive and significant (0.021, $p < 0.058$) if we exclude propagation cost from Model 3. Hence Model 3 suggests that, even though propagation cost and intrinsic cyclicality are highly correlated, they are distinct structural properties and have significantly different impacts on the generation of defects.

To understand further the relationship between intrinsic loops and the expected number of defects, we estimate a model that includes the average size of intrinsic loops as the key predictor of interest (Model 4). This model shows a positive but not significant effect of the average intrinsic loop size. We also estimate an alternative specification to Model 4 that includes the maximum intrinsic loop size

instead of the average size; the coefficient for interest is again positive but not significant (0.001, $p <$ 0.866). This suggests that, at the system level, it is the *fraction* of components involved in intrinsic loops—rather than the average (or maximum) size of such loops—that best predicts the expected number of defects.

To test for the effect of hierarchical cyclicality, we first estimate a model with hierarchical cyclicality as the key predictor of interest. Model 5 shows a positive but not significant coefficient for hierarchical cyclicality (0.013, $p < 0.161$). This indicates that hierarchical cyclicality is not a significant predictor of bugs in the system. Model 6 tests hypothesis H2a, which predicts that the greater the percentage of additional components included in hierarchical loops (in addition to the components involved in the corresponding intrinsic loops), the greater the expected number of defects. Note that our desire to test for the effect of additional components in hierarchical loops means that *delta cyclicality* is the parameter of interest. Model 6 shows a positive yet again not significant coefficient estimate of delta cyclicality (0.009, $p < 0.379$), in disagreement with H2a. Finally, we estimate Model 7 with both intrinsic cyclicality and delta cyclicality as key predictors. This model shows that the effect of intrinsic cyclicality remains positive and significant even after taking into account the effect of delta cyclicality.[11]

In order to test the robustness of our results to alternative ways to control for product connectedness, we estimate our regression models using direct and indirect connectivity rather than propagation cost. We measure direct and indirect connectivity by the absolute number of direct and indirect dependencies between product components, respectively. The results of this additional analysis are reported in Table B1 of Appendix B, which confirm the positive and significant effect of intrinsic cyclicality.

5 Component-level Analysis

The objective of our analysis in this section is to investigate further the mechanisms by which component loops increase the risk of generating bugs. As hypothesized in H1b and H2b, we can take a component-level perspective to test whether components involved in loops are at greater risk of being affected by bugs embedded in the application. Hence, in addition to the variables defined in the system-level analysis, we shall now define variables at the component level. This results in a hierarchical data structure in which some variables are defined at the component level and others at the system level.

For this analysis we use all the bugs reported in the bug-tracking system that have been fixed or are in the process of being fixed. We cannot include unfixed bugs because there is no definitive information about which components are affected by those bugs. However, we do control for the number of unfixed bugs associated with the version to which a component belongs. On average, there

[11] For completeness, we also estimate a model similar to Model 4 but with the average size of hierarchical loops as the key predictor of interest; we find an insignificant positive effect of average size of the hierarchical loop on the number of defects in the system (0.001, $p < 0.461$).

are 45.6 unfixed bugs (standard deviation = 55.2) per application version included in this analysis. In addition, for each component in all the versions of our database, we determine nonstructural variables based on the component-related information available on each bug entered in the bug-tracking system (e.g., types of modifications to a given component, dates of any component modification, names of developers modifying a given component). Finally, we also determine structural controls (at the component level) based on the connectivity patterns captured in the DSM. Our sample includes 28,395 components of 111 versions of 17 applications; 7% of the components are affected by at least one bug.

5.1 Variables

5.1.1 Dependent Variable: Number of Bugs per Component

The main dependent variable in this analysis is the number of bugs explicitly associated with component *c* of version *s* of application *i* (y_{csi}). We assign a bug to a component based on the information reported in the bug-tracking and version control systems.

5.1.2 Independent Variables

Our main predictor variables, at the component level, specify whether or not a component belongs to a loop. Since hierarchical loops contain intrinsic loops, a component that belongs to an intrinsic loop also belongs to a hierarchical loop. Therefore:

$INTRINSIC_{csi}$ = 1 if component *c* belongs to an intrinsic loop of version *s* of application *i*,

$INTRINSIC_{csi}$ = 0 otherwise;

$HIERARCHICAL_{csi}$ = 1 if component *c* belongs to a hierarchical loop of version *s* of application *i*,

$HIERARCHICAL_{csi}$ = 0 otherwise.

A combination of these two dummy variables uniquely determines whether or not a component belongs to a loop. A component that belongs to a hierarchical loop but not an intrinsic loop is an "additional" component ($HIERARCHICAL_{csi}$ = 1 and $INTRINSIC_{csi}$ = 0).

As in the system-level analysis, here we also test for the effect of the size of intrinsic loops at the component level by defining *Size of the intrinsic component loop* ($C\_INTRINSIC\_SIZE_{csi}$) as the number of components that complete an intrinsic loop to which component *c* belongs.

5.1.3 Component-level Control Variables

In addition to the control variables defined at the system level, we include variables that are relevant at the component level as follows.

Nonstructural Controls

- *Component age* (C_AGE_{csi}). This variable captures the number of days since the component was first included in application *i*.
- *Component-explicit non-bug changes* ($C\_EXPL\_CHANGES_{csi}$). This variable measures the number of improvements, new features, and other issues explicitly associated with component *c*. It controls for the total workload (beyond fixing bugs), which is defined as the number of changes in the product that required any type of change in the source code of component *c* of version *s* of application

i.

- *Component-implicit total changes* ($C\_IMPL\_CHANGES_{csi}$). This variable measures the number of bugs, improvements, new features, and other issues implicitly associated with component *c*. Although these types of changes in the source code are not explicitly associated with version *s*, their entry date in the bug tracking system occurs between the release dates of versions *s* and *s* + 1.
- *Cumulative number of changes* ($C\_CUM\_CHANGES_{csi}$). This variable captures the cumulative number of changes associated with component *c prior to* version *s* of application *i*; thus it controls for the cumulative workload generated by component *c* before the current version *s*. This is an important control because one could argue that components that have generated significantly more tasks in previous versions are more likely to motivate developers to discover and fix bugs. Yet one could also argue that, the more bug-fixing workload that component *c* has generated in the past, the more likely it is that such a component is bug-free in current version *s*. In the absence of a clear prediction, we include a control for this variable.
- *Cumulative number of committers* ($C\_CUM\_COMMITTERS_{csi}$). Committers are the people who have "write" access to the Apache code base. That is, in addition to adding and modifying code themselves, they review and approve code submitted by other programmers. This variable captures the cumulative number of committers associated with component *c* prior to version *s* of application *i*.
- *Cumulative number of authors* ($C\_CUM\_AUTHORS_{csi}$). Authors are the people who contribute a code segment to the project, which is then reviewed and approved by the committers. This variable captures the cumulative number of authors associated with component *c* prior to version *s* of application *i*.
- *Module abstraction deviation for component* ($C\_MOD\_ABSTRACT\_DEV_{csi}$). Because we measure each source-code module's lack of adherence to the stable-abstraction principle of agile software development and because each component is uniquely assigned to a module, we estimate the module abstraction deviation score of component *c* by assigning it the same score as all other components belonging to the same module.
- *Average component cyclomatic complexity* ($C\_AVG\_CC_{csi}$). As discussed in the system-level analysis, we calculate the cyclomatic complexity of all of the methods included in each component (see footnote 8 for the definition of "methods"). We aggregate this measure at the component level by averaging across all methods included in component *c*. This measure effectively controls for the internal complexity of component *c* as a function of the average number of linearly independent paths in its methods (McCabe 1976).
- *Source lines of code of the component* (C_SLOC_{csi}). At the component level, lines of code correspond to the number of statements that define the instructions in all the methods that define the component. This variable controls for the raw internal complexity of component *c*.

Structural Controls

Previous work has shown that the *connectivity* of components with other components in the product (also called "coupling" in the IS literature) is an important determinant of how components change over a product's life cycle (Henry and Kafura 1981, MacCormack et al. 2008b, von Krogh et al. 2008). It is therefore important to control for alternative measures of coupling when examining the effect (on quality) of a component's being included in a loop.

- *Fan-in component visibility* ($C\_FAN\_IN_{csi}$). This variable is a function of the number of other components in the product that depend directly or indirectly on component *c* (i.e., component *c* receives calls from other components). Given the binary visibility matrix (***V***) of version *s* of application *i*, which contains *N* components, we calculate fan-in component visibility as follows (Henry and Kafura 1981, MacCormack et al. 2008b):

$$C\_FAN\_IN_{csi} = \frac{\sum_k v_{ck}}{N-1} \tag{6}$$

where the numerator is the sum of the nonzero cells in *row c* of ***V***. This measure captures the percentage of components that could be affected if changes in component *c* propagate to other components.

- *Fan-out component visibility* ($C\_FAN\_OUT_{csi}$). This variable is a function of the number of other components upon which component *c* depends directly or indirectly. We have

$$C\_FAN\_OUT_{csi} = \frac{\sum_k v_{kc}}{N-1} \tag{7}$$

where the numerator is now the sum of the nonzero cells in *column c* of ***V***. This measure captures the percentage of components that could generate a requirement to change component *c*.

5.2 Analysis: Predicting the Number of Bugs Affecting a Component

Table 3 gives descriptive statistics and correlations of the component-level variables.

Because our dependent variable in this analysis counts the number of bugs explicitly affecting component *c*, we estimate a Poisson hierarchical regression model. (Note that our component-level "count" dependent variable does not exhibit signs of overdispersion because its variance is not significantly larger than its mean.) Given the hierarchical structure of our component-level data, we must use a hierarchical modeling framework to complete this analysis (Raudenbush and Bryk 2002). In addition to the panel data structure of the previous analysis, component *c* is nested into application *i*; we observe several successive versions of *i*, thus forming a panel of hierarchical data. This type of analysis allows us to test for component-level effects in the presence of system-level covariates. For estimation purposes, we use the *xtmepoisson* procedure recently implemented in Stata 10.1, which assumes that the conditional distribution of the dependent variable in the presence of random effects is Poisson (Rabe-Hesketh and Skrondal 2005). We estimate a model of the following form:

$$E[y_{cis} \mid x_{cis}, \alpha_{is}, \alpha_i] = \alpha_{is}\alpha_i \exp\{x'_{is}\beta_{is} + x'_{cis}\beta_{cis}\} \tag{8}$$

Consistently with the hierarchical linear modeling approach, this model fits a multilevel mixed-effects Poisson regression that contains "fixed" effects (the β-coefficients), which are analogous to standard regression coefficients, as well as "random" intercepts (the α-parameters), which are assumed to vary (following a Gaussian distribution) across versions and applications. These regression models predict that the expected number of bugs affecting component c in version s of application i depends exponentially on two sets of linearly independent regressors: a first set $\{x_{is}\}$ defined at the system level and a second set $\{x_{cis}\}$ defined at the component level. The models reported in Table 4 are random-intercept models. We test the robustness of our formulation by estimating random-coefficient models in which version-specific random effects vary with intrinsic cyclicality.[12] These alternative models yield substantially similar results to those reported in Table 4, but they offer no significant improvement in goodness of fit to data and so we report our results based on the most parsimonious model specification.

The β-coefficients shown in Table 4 are estimated by fitting the model to the data. For continuous regressors, the coefficient β_j equals the proportionate change in the expected mean if the jth regressor changes by one unit. For indicator variables, such as our key predictor variables in this analysis, a β-coefficient indicates that the expected conditional mean of the number of bugs is $\exp\{\beta\}$ larger if the indicator variable is 1 rather than 0 (Cameron and Trivedi 1998). For example, a coefficient $\beta_{INSTRINSIC}$ significantly greater than 0 would indicate that a component involved in an intrinsic loop has $\exp\{\beta_{INSTRINSIC}\}$ significantly greater expected number of bugs than another component that is not involved in an intrinsic loop, all else being equal.

Table 4 shows the results of fitting the data. Model 1 includes the system-level variables. Observe first that the number of unfixed bugs in the version to which the focal component c belongs does not significantly influence the expected number of bugs affecting this component.[13] Moreover, excluding such a system-level control yields substantially similar results as the ones reported here. Consistently with our system-level analysis, Model 1 shows a positive and significant effect of time to next release; this suggests that, the longer the time a version is active (before the next version is released), the greater is the probability of discovering bugs. Average cyclomatic complexity of the application also has a positive and significant baseline effect on all components in the release. Model 1 includes system cyclicality measures for both intrinsic and delta cyclicality (cf. Model 7 in Table 2). Like most

[12] We estimate these models with two alternative covariance structures and derive similar results. First, we use an "independent" covariance structure that allows a distinct variance for each random effect (for both application- and version-specific effects) and assumes that all covariances are zero. Second, we use an "exchangeable" covariance structure that has common variances and one common pairwise covariance.

[13] This is consistent with the conflicting assumptions behind the relationship between unfixed bugs of version s of application i and number of bugs of component c (in version s of application i). On one hand, if we assume that there might be a natural upper bound for the total number of bugs per given application, then it is possible that the larger the number of unfixed bugs per application, the lower the pool of bugs (either fixed or in the process of being fixed) that could be associated with component c; the result would be a negative coefficient for this effect. On the other hand, we could assume that there is no upper bound for the total number of bugs associated with an application and that the factors generating fixed bugs are similar to the factors generating unfixed bugs; this assumption would lead to a positive association between them.

of the system-level variables, the system cyclicality measures do not exhibit a significant effect at the component level. In other words, the expected number of bugs associated with component *c* appears to be *independent* of the amount of cyclicality in the system. This suggests that the variation in the number of bugs across components is primarily explained by component-level variables.

Model 2 adds nonstructural component-level controls and shows that the cyclomatic complexity and the number of lines of code of a component are important determinants of the number of bugs associated with that component (cf. Card and Glass 1990, Henry and Selig 1990). We also control for the number of changes—such as incremental improvements and the addition of new features—associated with the focal component *c* in version *s* of application *i*. The positive and significant coefficients of the number of changes (both explicit and implicit) associated with component *c* in version *s* of application *i* suggest that the number of bugs associated with a component is positively correlated with the number of other changes (in addition to bug fixing) that affect the component. In addition, we control for the amount of organizational attention and resources associated with the focal component since its inception in application *i*. First, it is important to acknowledge the difference between committers and authors: committers act as release managers who ultimately *rule on* (approve or reject) changes to the source code, whereas authors merely *propose* changes to source code—not only to fix bugs but also to implement functional improvements. The negative and significant coefficient for cumulative changes to the source code of component *c* (prior to the current version *s*) suggests that components that have been dealt with in previous versions of the application are *less* likely to be affected by bugs in the current version. However, the larger the number of authors dealing with a component in the past, the *more* likely it is that such a component will be associated with a higher number of bugs in the current version. The nonsignificant coefficient for cumulative committers is not entirely surprising when one considers the high correlation of this control variable with both cumulative authors and cumulative changes. Moreover, after excluding the cumulative changes control variable from our models, we obtain a substantially similar pattern of results (including our key predictor variables). In such models, however, the coefficient for cumulative committers is significantly negative while the coefficient for authors remains significantly positive. This suggests that the interplay of both attention and resources are important (and confounding) determinants of defects at the component level: on one hand, prior changes (i.e., attention) to the components might have a debugging effect, leading to fewer bugs in version *s*; on the other hand, many distinct authors changing component *c* may increase its likelihood of defects (resulting from the "patches over patches" approach).

Model 3 includes two types of structural controls at the component level: fan-in visibility (how much other components in version *s* depend on component *c*) and fan-out visibility (how much component *c* depends on other components in version *s*). Model 3 shows a negative (and not yet significant) coefficient for fan-in visibility but a positive and significant coefficient for fan-out visibility. This suggests that components that depend (directly and/or indirectly) on other components

are likely to be affected by a larger number of bugs. This is consistent with recent work showing that the connectivity of components affects their evolution over time (MacCormack et al. 2008b, von Krogh et al. 2008) and also suggests that the directionality of component connectivity has an impact on the propensity for defects (Kan 1995). Hence, it is important to emphasize the importance of considering the directionality of components' dependencies when examining their impact on product performance.

Model 4 tests H1b. This model shows a positive (and significant) coefficient for *INTRINSIC*, which indicates that the expected number of bugs affecting components that belong to an intrinsic loop is significantly larger than the expected number of bugs affecting other components. This result provides empirical support for our core hypothesis at the component level. Model 5 provides further insight about the effect of the absolute size of intrinsic loops by including a predictor variable that counts the number of components comprised by the intrinsic loop to which component *c* belongs. (Note that $C\_INTRINSIC\_SIZE_{csi} = 0$ if component *c* does not belong to an intrinsic loop.) Model 5 shows a positive and significant coefficient for $C\_INTRINSIC\_SIZE_{csi}$, which suggests that size really does matter at the component level: the bigger the intrinsic loop involving component *c*, the greater the expected number of bugs associated with that component. Combining this result with the effect of average intrinsic loop size at the system level (Model 4 of Table 2), we learn that smaller intrinsic loops indeed carry lower risk of generating bugs (i.e., an intrinsic loop of size *X* will be associated with significantly fewer bugs than an intrinsic loop of size 2*X*); however, this does not mean that applications with smaller average size of intrinsic loop have fewer bugs. Rather, it suggests, for instance, that an application with one intrinsic loop of size *X* is likely to have the same expected number of bugs as an application with two intrinsic loops, each of size *X*/2 (average loop size = X/2).[14]

We may begin to examine the effects of hierarchical loops via Model 6, which includes a positive and significant coefficient for *HIERARCHICAL*. This result suggests that components involved in hierarchical loops are likely to be affected by a significantly larger number of bugs than components outside any loops. Model 7 tests H2b, which predicts that the additional components involved in hierarchical loops (but not intrinsic loops) are more likely to be affected by bugs than components outside loops. Model 7 includes both *HIERARCHICAL* and *INTRINSIC* indicator variables, and it shows positive and significant coefficients for both indicator variables. This suggests that components that are involved in hierarchical loops but are not part of the intrinsic loops within them—in other words, the "additional" components in hierarchical loops—are likely to be affected by a significantly

[14] To compare the magnitude of the effect caused by the size of the intrinsic loop to which a component *c* belongs with the positive effect of fan-out visibility of component *c*, we estimate an alternative specification to Model 4 that includes a normalized measure of the size of intrinsic loop (which is divided by the number of components in version *s* of application *i*). This alternate model results in a positive and significant coefficient for the normalized size of intrinsic loop (0.013, $p < 0.012$) that is slightly greater than (but of the same order of magnitude as) the positive and significant effect of fan-out visibility (0.011, $p < 0.0001$).

larger number of bugs than components outside loops (in line with H2b), whereas components that belong to intrinsic loops will likely exhibit an even greater number of defects than any such additional components (in line with H1b). In fact, the positive and significant coefficient for *HIERARCHICAL* shown in Model 7 indicates that an "additional" component of a hierarchical loop has a 1.22 (= exp(0.202)) times more expected number of bugs than does a component that is not involved in any loop. However, the positive and significant coefficient for *INTRINSIC* indicates that a component in an intrinsic loop has an even larger number of expected bugs: it has 1.37 (= exp(0.314)) times more expected number of bugs than does an "additional" component in a hierarchical loop (all else equal). This provides further evidence for our core argument that the components of intrinsic loops are the root cause of iterative and difficult problem solving and thus are at greater risk of defect proneness. Such risk can spill over to other components in the modules to which the intrinsic components belong.[15]

Similar to our system-level analysis, we test the robustness of our results to the way we control for component connectivity by estimating alternative regressions models controlling for the absolute number of direct and indirect in-coming and out-going dependencies instead of controlling for fan-in and fan-out component visibility. The results of this alternative analysis, which are consistent with the ones reported here, are exhibited in Table B2 of Appendix B.

6 Discussion

This paper identifies a new fundamental architectural property of designed systems that is significantly related to product performance. We are the first to use an information-processing view of product architectures to identify *component loops* that are likely to cause defects because they are associated with iterative problem solving during the development effort. Component loops are formed by the set of components involved in cyclical dependencies. We identify component loops in two ways. *Intrinsic* loops are identified while ignoring the hierarchical structures of nested modules in which the source code resides, whereas *hierarchical* loops take this arrangement into account. We present empirical evidence—at both the system and component levels of a considerable sample of open-source, Java-based software products—that intrinsic loops are a significant determinant of the number of defects. At the system level, we find that the expected number of bugs is positively associated with intrinsic cyclicality. At the component level, we find that components involved in intrinsic loops should be affected by a larger number of bugs than any other components in the system.

Another finding of interest is that the organizational decisions by which developers group components into nested modules create a marginal additional negative effect on quality. In particular:

[15] That components in intrinsic loops carry a significantly larger share of bugs than the additional components that form hierarchical loops suggests an explanation for the lack of significance of delta cyclicality in our system-level analysis. The effect of intrinsic cyclicality dominates the effect the effect of delta cyclicality, which in the application-level analysis does appear to be significant.

we test for whether the components that are constrained by their module membership to be directly influenced by intrinsic loops exhibited more than the expected number of defects; and we find marginal empirical evidence for this conjecture. At the system level we find that hierarchical loops are not a significant predictor of defects in the system, but at the component level we find that components belonging to hierarchical loops (but not to intrinsic loops) are at higher risk of carrying defects than components outside any loops. This suggests that (i) the intrinsic cycles defined by the dependency structure of the design carry the main risk of creating defects and (ii) that such risk is neither easily mitigated nor substantially worsened by the architecture of arranging components into modules. This also suggests that the way "modules" are defined does not necessarily capture the dependency structure of their design. Perhaps it once did, but considering the rapid evolution of software designs, means that a configuration of nested modules is likely a legacy of past choices that do not reflect the current dependency structure determining defect proneness (MacCormack et al. 2008b).

From a conceptual viewpoint, our work highlights the importance of considering the two distinct but interrelated views of product architectures: the hierarchical view, in which components are organized into nested modules; and the intrinsic (or structural) view, which is determined by patterns of dependencies between product components. Hierarchical structures are important (and real) in the design of complex systems. As Simon (1996, p. 128) suggested, "complex systems might be expected to be constructed in a hierarchy of levels, or in-boxes-within-boxes form. The basic idea is that the several components in any complex system will perform particular subfunctions that contribute to the overall function." Hence our examination of the relationship between system cyclicality and quality—with and without the presence of hierarchies—was warranted. However, the fact that intrinsic (and not hierarchical) cyclicality is the most substantial determinant of defects suggests that, in order to take any actions based on these results, managers must look beyond the hierarchical arrangements used to design their systems and focus on the system's actual dependency structure. We thus advocate that the focus of managerial attention be shifted from the *intended* system architecture (the modules that managers typically consider to be the building blocks of the system architecture) to the *actual* architecture (as defined by the dependency structure of the design). We believe that this paper provides clear guidance on where (and how) to look for the architectural areas that increase the risk of generating bugs and therefore require special managerial attention during the development process.

In addition, we provide an alternative perspective regarding the effect of connectedness (or coupling) on quality. Instead of assuming that greater connectedness among components generates more bugs, we argue that a particular *type* of connectedness (creating cycles) is among the most important determinants of bugs. This contrasts with previous work that has emphasized the cost of change propagation due to both high direct *and* indirect connectivity among components (Eckert et al. 2004, MacCormack et al. 2006). Although we agree that the connectedness (or coupling) of a system has many undesirable features that can hinder several dimensions of quality, including extendibility

and maintainability of the design (MacCormack et al. 2008b), our results suggest that systems built in layers (such that components in lower layers or libraries serve components in higher layers) require and benefit from direct and indirect dependencies that obey this directional design rule (Baldwin and Clark 2000). Hence, our results constitute a warning about the generalization that connectedness among components is necessarily an undesirable feature.

Our results from the system-level analysis suggest that the larger the fraction of components involved in intrinsic loops, the higher the risk of generating bugs. Consistently, our component-level analysis indicates that a component involved in intrinsic loops has a significantly greater expected number of bugs than any other component that is not involved in intrinsic loops. Hence, by identifying which components are involved in intrinsic loops, managers can anticipate which components will require more attention and resources to fix bugs. We therefore offer the following recommendations for improving product quality that are based on better management of product architecture knowledge.

- Visualize the *actual* architecture: First, it is crucial for managers to understand the key components of the architecture of the products they design: nested modules that group components and dependencies among components. Visualizing the architecture is fundamental to improving one's understanding of the organization around it. Further illustrations of this technique are available for hardware products (Sosa et al. 2003, Sosa et al. 2007b) as well as for software products (MacCormack et al. 2006).
- Identify intrinsic loops: Follow our approach to identify intrinsic component loops based on the product's dependency structure. Identify precisely the components that belong to those loops and the actors responsible for their design. These people will need special managerial attention while they are iterating their interdependent designs. It is important to emphasize here that, in order to identify intrinsic loops, managers must *disregard* the constraints imposed by the way in which components are arranged into nested modules.
- Reduce the size of the component loops: Larger component loops are at higher risk of generating more bugs. Identify the dependency that would make the loop smaller, and consider ways to "freeze" or "standardize" the relevant interface. Also consider ways to break a large loop into smaller, more manageable loops.
- Preempt the cycles: Enforce design rules—not just generally, but in a prioritized way. Design rules (see Baldwin and Clark 2000, Sangal et al. 2005) provide ways of specifying which types of component relationships are allowed. Disallowed relationships could include those that create cycles or those that inappropriately cross modules or layers. Moreover**,** because a module is typically defined in terms of the common functionality of its components, it is important for modules to be designed such that they encapsulate intrinsic loops; this will facilitate the management of attention toward them and of their resource use.

The analysis presented here takes advantage of our data's longitudinal nature to suggest a causal link between architecture and quality. Since the bugs captured by our dependent variables are measured after the application architecture has been established and since our predictor variables measure architectural properties determined before the release, the potential for reverse causality is unlikely. Even so, the data do introduce some limitations. Because our analysis is carried out on a sample of Java-based applications developed by the open-source Apache Software Foundation, studies in other settings (e.g., closed-source software applications, complex hardware products) are warranted before the findings here can be more fully generalized. Another limitation concerns our conceptualization of dependencies as being of a single type; in reality, there are various types of dependencies. Future work should investigate whether (and how) the *type* of dependency moderates the effect of cyclicality on product performance.

Examining the relation between the architecture of software applications and their quality has allowed us to establish the mechanisms by which component loops are likely to influence the generation of bugs. Yet many open questions remain to be addressed by future research. How does system cyclicality affect the time to fix bugs? What other characteristics of an architecture might generate bugs? How do open-source and closed-source software architectures differ? Do any such differences lead to differences in quality? Our ongoing research efforts seek to provide answers to these questions.

References

Alexander, C. 1964. *Notes on the Synthesis of Form*. Harvard University Press, Cambridge, MA.

Baldwin, C. Y., K. B. Clark. 2000. *Design Rules: The Power of Modularity*. MIT Press, Cambridge, MA.

Basili, V. R., B. T. Perricone. 1984. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*. **27**(1) 42-52.

Briand, L. C., J. Daly, J. Wüst. 1999. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*. **25**(1) 91-121.

Briand, L. C., H. Lounis, S. V. Ikonomovski, J. Wüst. 1998. A Comprehensive Investigation of Quality Factors in Object-Oriented Designs: An Industrial Case Study. International Software Engineering Research Network, Technical Report ISERN-98-29.

Briand, L. C., W. L. Melo, J. Wüst. 2002. Assessing the Applicability of Fault-Proneness Models Across Object-Oriented Software Projects. *IEEE Transactions on Software Engineering*. **28**(7) 706-720.

Browning, T. R. 2001. Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions. *IEEE Trans. on Eng. Mgmt.* **48**(3) 292-306.

Cameron, A. C., P. K. Trivedi. 1998. *Regression Analysis of Count Data*. Cambridge University Press, Cambridge, U.K.

Card, D. N., R. L. Glass. 1990. *Measuring Software Design Quality*. Prentice-Hall, Englewood Cliffs, NJ.

Cataldo, M., P. Wagstrom, J. D. Herbsleb, K. M. Carley. 2006. Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools. *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, Banff, Alberta, 353-362.

Chidamber, S., C. Kemerer. 1994. A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*. **20**(6) 476-493.

Clark, K. B., T. Fujimoto. 1991. *Product Development Performance: Strategy, Organization, and Management in the World Auto Industry*. Harvard Business School Press, Boston.

Clarkson, P. J., C. Simons, C. Eckert. 2004. Predicting Change Propagation in Complex Design. *J. of Mech. Design*. **126** 788-797.

Eckert, C. M., P. J. Clarkson, W. Zanker. 2004. Change and Customization in Complex Engineering Domains. *Research in Engineering Design*. **15**(1) 1-21.

Eppinger, S. D., D. E. Whitney, R. P. Smith, D. A. Gebala. 1994. A Model-Based Method for Organizing Tasks in Product Development. *Res. in Eng. Design*. **6**(1) 1-13.

Fenton, N. E., M. Neil. 1999. A Critique of Software Defect Prediction Models. *IEEE Transactions on Software Engineering*. **25**(5) 675-689.

Fenton, N. E., N. Ohlsson. 2000. Quantitative Analysis of Faults and Failures in Complex Software Systems. *IEEE Transactions on Software Engineering*. **26**(8) 797-814.

Gokpinar, B., W. J. Hopp, S. M. R. Iravani. 2010. The Impact of Misalignment of Organizational Structure and Product Architecture on Quality in Complex Product Development. *Management Sci.* **56**(3) 468-484.

Hausman, T., B. H. Hall, Z. Griliches. 1984. Econometric Models for Count Data with an Application to the Patents-R&D Relationship. *Econometrica*. **52**(4) 909-938.

Henderson, R. M., K. B. Clark. 1990. Architectural Innovation: The Reconfiguration of Existing Product Technologies and the Failure of Established Firms. *Admin. Sci. Quarterly*. **35** 9-30.

Henry, S. M., D. Kafura. 1981. Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering*. **7**(5) 510-518.

Henry, S. M., C. Selig. 1990. Predicting Source-Code Complexity at the Design Stage. *IEEE Software*. **7**(2) 36-44.

Kan, S. H. 1995. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, Reading, MA.

Koru, A. G., K. E. Emam, D. Zhang, H. Liu, D. Mathew. 2008. Theory of Relative Defect Proness: Replicated Studies on the Functional Form of the Size-defect Relationship. *Empirical Software Engineering*. **13** 473-498.

Koru, A. G., H. Liu. 2007. Identifying and Characterizing Change-prone Classes in Two Large Scale Open-source Products. *The Journal of Systems and Software*. **80** 63-73.

Koru, A. G., J. Tian. 2004. Defect Handling in Medium and Large Open Source Projects. *IEEE Software* 54-61.

Krishnan, V., S. D. Eppinger, D. E. Whitney. 1997. A Model-Based Framework to Overlap Product Development Activities. *Management Sci.* **43**(4) 437-451.

Lai, X., J. K. Gershenson. 2006. Representation of Similarity and Dependency for Assembly Modularity. *Proceedings of the ASME Design Engineering Technical Conferences - 18th International Conference on Design Theory and Methodology*, Philadelphia, PA.

Legard, H. F., M. Marcotty. 1975. A Genealogy of Control Structures. *Communications of the ACM*. **18**(11) 629-639.

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

MacCormack, A., J. Rusnak, C. Y. Baldwin. 2008a. Exploring the Duality between Product and Organizational Architectures: A Test of the Mirroring Hypothesis. Harvard Business School, Working Paper 08-039.

MacCormack, A., C. Y. Baldwin, J. Rusnak. 2008b. The Impact of Component Modularity on Design Evolution: Evidence from the Software Industry. Harvard Business School, Working Paper 08-038.

MacCormack, A. D., R. Verganti, M. Iansiti. 2001. Developing Products on "Internet Time": The Anatomy of a Flexible Development Process. *Management Sci.* **47**(1) 133-150.

Martin, R. C. 1994. OO Design Quality Metrics - An Analysis of Dependencies (Position Paper), in *Proc. Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA'94*.

Martin, R.C. 1995. *Designing Object Oriented C++ Applications using the Booch Method.* Prentice Hall, Englewood Cliffs, NJ.

Martin, R. C. 2002. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, Englewood Cliffs, NJ.

McCabe, T. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering*. **2**(4) 308-320.

Meier, C., A. A. Yassine, T. R. Browning. 2007. Design Process Sequencing with Competent Genetic Algorithms. *J. of Mech. Design*. **129**(6) 566-585.

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

Mihm, J., C. Loch, A. Huchzermeier. 2003. Problem-Solving Oscillations in Complex Engineering Projects. *Management Sci.* **49**(6) 733-750.

Parnas, D. L. 1972. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*. **15**(12) 1053-1058.

Parnas, D. L. 1979. Designing Software for Ease of Extension and Contraction. *Transactions on Software Engineering*. **5**(2).

Pich, M. T., C. H. Loch, A. D. Meyer. 2002. On Uncertainty, Ambiguity and Complexity in Project Management. *Management Sci.* **48**(8) 1008-1023.

Pimmler, T. U., S. D. Eppinger. 1994. Integration Analysis of Product Decompositions. *Proceedings of the ASME International Design Engineering Technical Conferences (Design Theory & Methodology Conference)*, Minneapolis, Sep.

Rabe-Hesketh, S., A. Skrondal. 2005. *Multilevel and Longitudinal Modeling Using Stata*. Stata Press.

Raudenbush, S., A. Bryk. 2002. *Hierarchical Linear Models, Applications and Data Analysis Methods*. 2nd Edition, Sage Publications.

Roberts, J. A., I.-H. Hann, S. A. Slaughter. 2006. Understanding the Motivations, Participation, and Performance of Open Source Software Developers: A Longitudinal Study of the Apache Projects. *Management Sci.* **52**(7) 984-999.

Roemer, T. A., R. Ahmadi, R. H. Wang. 2000. Time-Cost Trade-Offs in Overlapped Product Development. *Operations Res.* **48**(6) 858-865.

Sangal, N., E. Jordan, V. Sinha, D. Jackson. 2005. Using Dependency Models to Manage Complex Software Architecture. *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages And Applications (OOPSLA)*, San Diego, CA, Oct 16-20, 167-176.

Sharman, D. M., A. A. Yassine. 2004. Characterizing Complex Product Architectures. *Systems Eng.* **7**(1) 35-60.

Shaw, M., D. Garlan. 1996. *Software Architecture*. Prentice Hall, Upper Saddle River, NJ.

Simon, H. A. 1996. *The Sciences of the Artificial*. 3rd Edition, MIT Press, Cambridge, MA.

Smith, R. P., S. D. Eppinger. 1997a. A Predictive Model of Sequential Iteration in Engineering Design. *Management Sci.* **43**(8) 1104-1120.

Smith, R. P., S. D. Eppinger. 1997b. Identifying Controlling Features of Engineering Design Iteration. *Management Sci.* **43**(3) 276-293.

Sommer, S.C., C. H. Loch. 2004. Selectionism and Learning in Projects with Complexity and Unforeseeable Uncertainty. *Management Sci.* **50**(10) 1334-1347.

Sommerville, I. 2007. *Software Engineering*. 8th Edition, Addison-Wesley, New York, NY.

Sosa, M. E. 2008. A Structured Approach to Predicting and Managing Technical Interactions in Software Development. *Research in Engineering Design*. **19**(1) 47-70.

Sosa, M.E., T.R. Browning, J. Mihm. 2007a. Studying the Dynamics of the Architecture of Software Products. *Proceedings of the ASME 2007 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference (IDETC/CIE 2007)*, Las Vegas.

Sosa, M. E., S. D. Eppinger, C. M. Rowles. 2003. Identifying Modular and Integrative Systems and Their Impact on Design Team Interactions. *J. of Mech. Design*. **125**(2) 240-252.

Sosa, M. E., S. D. Eppinger, C. M. Rowles. 2004. The Misalignment of Product Architecture and Organizational Structure in Complex Product Development. *Management Sci.* **50**(12) 1674-1689.

Sosa, M. E., S. D. Eppinger, C. M. Rowles. 2007b. A Network Approach to Define Modularity of Components in Product Design. *J. of Mech. Design*. **129**(11) 1118-1129.

Stein, C., G. Cox, L. Etzkorn. 2005. Exploring the Relationship Between Cohesion and Complexity. *Journal of Computer Science*. **1**(2) 137-144.

Stevens, W. P., G. J. Myers, L. L. Constantine. 1974. Structured Design. *IBM Systems Journal*. **13**(2) 115-139.

Steward, D. V. 1981. The Design Structure System: A Method for Managing the Design of Complex Systems. *IEEE Trans. on Eng. Mgmt.* **28**(3) 71-74.

Sullivan, K. J., W. G. Griswold, Y. Cai, B. Hallen. 2001. The Structure and Value of Modularity in Software Design. *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Vienna, Austria, Sep. 10-14, 99-108.

Terwiesch, C., C. H. Loch. 1999. Managing the Process of Engineering Change Orders: The Case of the Climate Control System in Automobile Development. *J. of Product Innov. Mgmt.* **16**(2) 160-172.

Terwiesch, C., C. H. Loch, A. D. Meyer. 2002. Exchanging Preliminary Information in Concurrent Engineering: Alternative Coordination Strategies. *Organization Sci.* **13**(4) 402-419.

Thompson, J. D. 1967. *Organizations in Action*. McGraw-Hill, New York.

Ulrich, K. T. 1995. The Role of Product Architecture in the Manufacturing Firm. *Res. Policy*. **24**(3) 419-440.

von Krogh, G., M. Stuermer, M. Geipel, S. Spaeth, S. Haefliger. 2008. How Component Dependencies Predict Change in Complex Technologies. ETH Zurich, Working Paper.

von Krogh, G., E. von Hippel. 2006. The Promise of Research on Open Source Software. *Management Sci.* **52**(7) 975-983.

Warfield, J. N. 1973. Binary Matrices in System Modeling. *IEEE Transactions on Systems, Man, and Cybernetics*. **3**(5) 441-449.

Zhang, M., N. Baddoo. 2007. Performance Comparison of Software Complexity Metrics in an Open Source Project. *Proceedings of the 14th European Conference on Software Process Improvement (EuroSPI 2007)*, Potsdam, Germany, Sep 26-28, 160-174.

Table 1. Descriptive Statistics and Correlations of System-level Variables (N = 122)

	Mean	STD	Min	Max	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1. Number of bugs (y_{is})	81.5	122.1	0	647	1.00															
2. AGE_{is}	708.9	644.6	0	2855	0.23	1.00														
3. $DAYS\ BEFORE_{is}$	204.3	222.6	0	1230	0.34	0.55	1.00													
4. $DAYS\ AFTER_{is}$	252.5	271.8	7	1342	0.40	0.29	0.33	1.00												
5. $NEWNESS_{is}$	31.6	44.6	0	258	0.44	0.14	0.36	0.12	1.00											
6. $IMPLICIT\_BUGS_{is}$	19.2	40.5	0	288	0.24	0.02	-0.01	-0.02	0.08	1.00										
7. $NUM\_MODULES_{is}$	29.8	26.1	2	181	0.10	0.14	-0.03	0.01	-0.06	0.40	1.00									
8. $AVG\_MOD\_ABS_{s}$	0.1	0.1	0	0.3	0.19	0.20	0.08	0.07	-0.03	0.16	0.55	1.00								
9. AVG_CC_{is}	2.5	2.5	1.3	26	0.03	0.07	0.03	0.16	0.03	0.02	-0.10	-0.15	1.00							
10. $SLOC_{is}$	20.3	17.0	1.0	108.0	0.25	0.23	0.07	0.05	0.01	0.23	0.69	0.62	-0.09	1.00						
11. $PROPAGATION_{is}$	14.7	9.8	0.9	48.6	0.06	-0.20	-0.11	0.02	0.08	-0.14	-0.33	-0.17	-0.02	-0.11	1.00					
12. Intrinsic NUM_LOOPS_{is}	3.9	2.6	1	15	0.30	0.33	0.06	0.18	-0.12	0.26	0.35	0.22	-0.08	0.59	-0.03	1.00				
13. Hierarchical NUM_LOOPS_{is}	2.3	1.5	1	6	0.04	-0.13	-0.19	-0.19	-0.19	0.26	0.54	0.37	-0.15	0.27	-0.36	0.26	1.00			
14. Intrinsic cyclicality ($P_{I,is}$)	17.2	11.3	0.5	60.1	0.31	-0.04	0.01	0.11	0.10	-0.01	-0.16	-0.01	0.03	0.20	0.83	0.26	-0.24	1.00		
15. Hierarchical cyclicality ($P_{H,is}$)	81.9	18.3	24.3	100	0.20	0.03	0.06	0.17	0.02	0.12	0.13	0.23	-0.21	0.30	0.38	0.35	0.01	0.42	1.00	
16. Delta cyclicality ($P_{D,is}$)	64.7	17.1	6.3	96.6	0.01	0.06	0.06	0.11	-0.04	0.14	0.25	0.26	-0.25	0.19	-0.14	0.20	0.17	-0.22	0.80	1.00
17. AVG Intrinsic Size ($AVG\_CSIZE_{I,is}$)	14.8	18.5	2	113	0.22	-0.07	0.05	0.01	0.15	-0.05	0.00	0.14	0.00	0.25	0.59	-0.10	-0.17	0.77	0.25	-0.24

Correlations > |0.15| are significant at $p < .01$

Table 2. Negative Binomial Regressions Predicting Expected Number of Bugs per Application (N = 122)

Independent variables	Model 1 controls	Model 2 propagation	Model 3 intrinsic	Model 4 size	Model 5 hierarchical	Model 6 delta	Model 7 full
AGE_{is} !	.140 (.286)	.125 (.284)	.162 (.304)	.113 (.294)	.128 (.306)	.076 (.333)	.054 (.349)
$DAYS\ BEFORE_{is}$!	.129 (.436)	.202 (.436)	-.006 (.454)	.061 (.473)	.160 (.438)	.258 (.464)	.083 (.467)
$DAYS\ AFTER_{is}$!	1.348*** (.332)	1.287*** (.339)	1.388*** (.337)	1.365*** (.346)	1.074*** (.380)	1.147*** (.404)	1.365*** (.399)
$NEWNESS_{is}$!	2.247 (2.044)	1.768 (2.029)	2.910 (2.072)	2.567 (2.143)	1.898 (1.992)	1.838 (2.060)	2.994 (2.109)
$IMPLICIT\_BUGS_{is}$!	-1.998 (2.330)	-1.871 (2.307)	-3.397 (2.465)	-2.285 (2.381)	-2.567 (2.436)	-2.386 (2.539)	-4.127 (2.764)
$NUM\_MODULES_{is}$!	6.990 (6.762)	7.998 (6.831)	9.089 (6.986)	7.722 (6.889)	9.302 (7.112)	9.259 (7.226)	10.798 (7.420)
$AVG\_MOD\_ABST\_DEV_{is}$	-.807 (2.421)	-.121 (2.493)	-.625 (2.505)	-.824 (2.652)	.268 (2.609)	.454 (2.670)	-.118 (2.651)
AVG_CC_{is}	.066*** (.024)	.068*** (.025)	.067*** (.022)	.063** (.025)	.066** (.031)	.067** (.030)	.067*** (.022)
$SLOC_{is}$	-.015 (.010)	-.017* (.010)	-.026** (.012)	-.024* (.013)	-.019* (.010)	-.020 (.012)	-.029** (.014)
$PROPAGATION\_COST_{is}$		.014 (.012)	-.037 (.028)	-.002 (.020)	.008 (.013)	.018 (.014)	-.042 (.035)
Intrinsic NUM_LOOPS_{is}			.003 (.049)	.069 (.073)		.022 (.062)	.024 (.059)
Intrinsic cyclicality ($P_{I,is}$)			**.054* (.028)**				**.056* (.030)**
AVG Intrinsic Loop Size ($AVG\_CSIZE_{,I,is}$)				.011 (.011)			
Hierarchical NUM_LOOPS_{is}					-.030 (.105)	-.037 (.116)	-.076 (.119)
Hierarchical cyclicality ($P_{H,is}$)					.013 (.009)		
Delta cyclicality ($P_{D,is}$)						.009 (.011)	.001 (.014)
Log Likelihood	-455.444	-454.808	-452.942	-454.277	-453.888	-454.389	-452.729
McFadden's pseudo R^2	0.0739	0.0752	.0790	.0763	0.0771	0.0761	0.0795
Wald Chi-sq	97.39***	99.83***	105.16***	98.89***	101.21***	101.01***	106.61***

*< .1 ** < .05 *** < .01 (two-tailed). Standard errors are shown between parentheses.

! Coefficients are multiplied by 1000 to facilitate exposition of results.

All models include application-specific fixed effects and year effects.

- Excluding *propagation cost* from Model 3 yields a positive and significant effect of $P_{I,is}$ (.021, p < .058).
- A likelihood-ratio test comparing Model 3 against a restricted model excluding $P_{I,is}$ yields a significant improvement in the goodness of fit. T_{LR} = -2 × (-454.750 + 452.942) = 3.616 and $\chi^2(1)$, which results in p < 0.058.
- Excluding *propagation cost* from Model 4 yields a positive, not significant, effect of $AVG\_CSIZE_{,I,is}$ (.010, p < .119).
- Excluding *propagation cost* from Model 7 yields a positive and significant effect of $P_{I,is}$ (.023, p < .058).
- A likelihood-ratio test comparing Model 7 against a restricted model excluding $P_{I,is}$ yields a significant improvement in the goodness of fit. T_{LR} = -2 × (-454.389 + 452.729) = 3.321 and $\chi^2(1)$, which results in p < 0.068.

Table 3. Descriptive Statistics and Correlations of Component-level Variables (N = 28,395)

	Mean	STD	Min	Max	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1. Number of bugs (y_{csi})	0.09	0.41	0	10	1.00													
2. C_AGE_{csi}	400.30	494.06	0	2513	0.09	1.00												
3. $C\_EXPL\_CHANGES_{csi}$	0.62	3.17	0	113	0.23	0.04	1.00											
4. $C\_IMPL\_CHANGES_{csi}$	0.19	0.68	0	19	0.27	-0.02	0.08	1.00										
5. $C\_CUM\_CHANGES_{csi}$	2.95	9.74	0	307	0.23	0.24	0.40	0.23	1.00									
6. $C\_CUM\text{-}COMMITTERS_{csi}$	0.98	1.98	0	33	0.16	0.31	0.37	0.21	0.81	1.00								
7. $C\_CUM\text{-}AUTHORS_{csi}$	0.95	2.21	0	37	0.32	0.11	0.23	0.32	0.70	0.57	1.00							
8. $C\_MOD\_ABSTR\_DEV_{csi}$	0.29	0.25	0	0.944	-0.04	0.04	0.00	-0.02	-0.04	0.02	-0.12	1.00						
9. $C\_AVG\_CC_{csi}$!	2.09	1.98	0	48	0.05	0.03	0.04	0.11	0.08	0.10	0.11	0.07	1.00					
10. C_SLOC_{csi}!	72.53	148.25	1	4086	0.13	0.06	0.06	0.18	0.17	0.14	0.22	0.02	0.39	1.00				
11. $C\_FAN\_IN_{csi}$!	15.81	19.36	0	87.10	0.02	0.05	0.01	0.04	0.05	0.00	0.04	0.09	-0.04	0.05	1.00			
12. $C\_FAN\_OUT_{csi}$!	15.00	21.51	0	85.79	0.10	0.04	0.03	0.11	0.10	0.07	0.15	0.08	0.24	0.19	0.10	1.00		
13. $INTRINSIC_{csi}$	0.22	0.41	0	1	0.09	0.06	0.04	0.11	0.11	0.06	0.14	0.08	0.15	0.25	0.41	0.53	1.00	
14. $HIERARCHICAL_{csi}$	0.87	0.34	0	1	0.04	0.00	0.03	0.05	0.07	0.07	0.07	-0.03	0.03	0.05	0.16	0.16	0.21	1.00
15. $C\_INTRINSIC\_SIZE_{csi}$	12.56	30.88	0	169	0.00	0.05	-0.02	0.04	0.04	0.03	0.05	0.17	0.11	0.17	0.43	0.47	0.77	0.16

Correlations > |0.015| are significant at $p < .01$

Table 4. Hierarchical Poisson Regressions Predicting Number of Bugs per Component (*N* = 28,395)

Independent variables	Model 1	Model 2	Model 3	Model 4	Model 5	Model 6	Model 7
Coefficients of system-level variables							
Number unfixed $bugs_{is}$ †	-.728 (.637)	-.683 (.749)	-.676 (.725)	-.675 (.728)	-.664 (.732)	-.675 (.726)	-.675 (.729)
AGE_{is} †	-.095 (.085)	-.055 (.074)	-.054 (.075)	-.053 (.075)	-.055 (.075)	-.054 (.075)	-.053 (.075)
$DAYS\ BEFORE_{is}$ †	.044 (.105)	.069 (.111)	.070 (.109)	.071 (.109)	.070 (.110)	.070 (.109)	.070 (.109)
$DAYS\ AFTER_{is}$ †	.330*** (.090)	.258*** (.093)	.260*** (.092)	.260*** (.092)	.260*** (.092)	.259*** (.092)	.259*** (.092)
$NEWNESS_{is}$ †	.747 (.903)	.704 (.826)	.691 (.827)	.685 (.827)	.687 (.829)	.696 (.825)	.689 (.826)
$IMPLICIT\_BUGS_{is}$ †	.656 (.958)	.991 (.957)	.968 (.946)	.986 (.947)	.980 (.953)	.972 (.945)	.988 (.947)
$NUM\_MODULES_{is}$ †	-2.184 (1.435)	-1.270 (1.341)	-1.332 (1.339)	-1.319 (1.340)	-1.391 (1.346)	-1.334 (1.337)	-1.321 (1.338)
$AVG\_MOD\_ABSTR\_DEV_{is}$	.489 (7.602)	-3.661 (7.493)	-3.408 (7.465)	-3.449 (7.465)	-3.552 (7.497)	-3.506 (7.450)	-3.527 (7.454)
AVG_CC_{is}	.108** (.052)	.090* (.049)	.089* (.049)	.090* (.049)	.090* (.049)	.089* (.049)	.089* (.049)
$SLOC_{is}$	-.041 (.028)	-.028 (.024)	-.029 (.024)	-.029 (.024)	-.029 (.024)	-.029 (.024)	-.029 (.024)
$PROPAGATION\_COST_{is}$	-.053 (.062)	-.009 (.059)	-.026 (.058)	-.016 (.058)	-.020 (.058)	-.025 (.058)	-.015 (.058)
Intrinsic NUM_LOOPS_{is}	.113 (.141)	.114 (.148)	.112 (.146)	.112 (.146)	.114 (.146)	.112 (.146)	.112 (.146)
Intrinsic cyclicality ($P_{I,is}$)	-.006 (.063)	-.007 (.060)	-.007 (.060)	-.012 (.060)	-.013 (.059)	-.009 (.059)	-.014 (.060)
Hierarchical NUM_LOOPS_{is}	.129 (.301)	.082 (.319)	.095 (.313)	.098 (.314)	.091 (.316)	.097 (.313)	.100 (.314)
Delta cyclicality ($P_{D,is}$)	-.014 (.028)	.001 (.022)	.000 (.022)	.000 (.022)	.000 (.022)	-.002 (.022)	-.002 (.022)

Table continues on the next page

† Coefficients are multiplied by 100 to facilitate exposition of results.

Table 4. Cont.

	Model 1 cont.	Model 2 cont.	Model 3 cont.	Model 4 cont.	Model 5 cont.	Model 6 cont.	Model 7 cont.
Coefficients of component-level non-structural control variables							
C_AGE_{csi} !		-.002 (.046)	-.052 (.047)	-.061 (.047)	-.055 (.047)	-.037 (.047)	-.048 (.047)
$C\_EXPL\_CHANGES_{csi}$		.098*** (.006)	.092*** (.006)	.094*** (.006)	.096*** (.006)	.093*** (.006)	.095*** (.006)
$C\_IMPL\_CHANGES_{csi}$		.100*** (.013)	.090*** (.013)	.088*** (.013)	.087*** (.013)	.090*** (.013)	.089*** (.013)
$C\_CUM\text{-}COMMITTERS_{csi}$		-.015 (.013)	-.014 (.013)	-.005 (.013)	-.018 (.013)	-.014 (.013)	-.006 (.013)
$C\_CUM\text{-}AUTHORS_{csi}$		.176*** (.007)	.156*** (.008)	.150*** (.008)	.156*** (.008)	.154*** (.008)	.149*** (.008)
$C\_CUM\_CHANGES_{csi}$		-.020*** (.002)	-.017*** (.002)	-.017*** (.002)	-.017*** (.002)	-.017*** (.002)	-.017*** (.002)
$C\_MOD\_ABSTR\_DEV_{csi}$		.960*** (.143)	.938*** (.144)	.950*** (.144)	.912*** (.145)	.968*** (.145)	.977*** (.145)
$C\_AVG\_CC_{csi}$		.053*** (.010)	.036*** (.010)	.036*** (.010)	.036*** (.010)	.037*** (.010)	.038*** (.010)
C_SLOC_{csi}		.511*** (.082)	.547*** (.085)	.534*** (.085)	.527*** (.086)	.548*** (.084)	.536*** (.085)
Coefficients of component-level structural control variables							
$C\_FAN\_IN_{csi}$			-.001 (.001)	-.006*** (.001)	-.005*** (.001)	-.002 (.001)	-.006*** (.001)
$C\_FAN\_OUT_{csi}$			.013*** (.001)	.009*** (.001)	.011*** (.001)	.012*** (.001)	.009*** (.001)
Coefficients of component-level predictor variables							
$\boldsymbol{INTRINSIC_{csi}}$				**.325*** (.063)**			**.314*** (.063)**
$C\_INTRINSIC\_SIZE_{csi}$					.005*** (.001)		
$\boldsymbol{HIERARCHICAL_{csi}}$						**.229*** (.082)**	**.202** (.082)**
Log Likelihood	-6915.5	-5956.0	-5878.4	-5865.1	-5871.7	-5874.3	-5861.9
McFadden's pseudo R^2	0.0426	0.1754	0.1862	0.1880	0.1871	0.1867	0.1884
Wald Chi-sq	63.53***	2571.8***	2664.9***	2668.5***	2664.6***	2671.4***	2675.1***

* < .1 ** < .05 *** < .01. Standard errors are shown between parentheses.

! Coefficients are multiplied by 1000 to facilitate exposition of results.

All models include version and application-specific random effects and fixed year effects

- A likelihood-ratio test comparing Model 4 and Model 3 yields a significant improvement in the goodness of fit. $T_{LR} = -2 \times (-5878.4 + 5865.1) = 26.608$ and $\chi^2(1)$, which results in $p < 0.00001$
- A likelihood-ratio test comparing Model 6 and Model 3 yields a significant improvement in the goodness of fit. $T_{LR} = -2 \times (-5878.4 + 5874.3) = 8.238$ and $\chi^2(1)$, which results in $p < 0.005$
- A likelihood-ratio test comparing Model 7 and Model 4 yields a significant improvement in the goodness of fit. $T_{LR} = -2 \times (-5865.1 + 5861.9) = 6.307$ and $\chi^2(1)$, which results in $p < 0.01$

Appendix A: Algorithm for Calculating Hierarchical Component Loops

Overview

1. *Partitioning the matrix while keeping hierarchical relations intact:* The idea of hierarchical decomposition is to find the smallest cycles but not to allow for modules being torn apart. We do this by recursively applying the following algorithm. Start by listing all the modules one level below the root. Note that the *link strength* between modules is the number of links between individual classes in the modules. Then apply a standard sequencing algorithm to minimize the size of the loops weighted by their strength (see strength sequencer details below). As a result, there will be as few marks as possible above the diagonal, and the marks farthest away from the diagonal will have the lowest strength numbers. Next, for each module, list all submodules in the module and sequence them according to the strength sequencer details (available from the authors upon request). Apply this algorithm recursively until the leaf nodes (the Java classes) are reached.
2. *Finding the cycles in the partitioned matrix:* Next we need to find the (hierarchical) loops. For that we first create a set of the lowest-level modules that have a "1" in the upper diagonal of the partitioned matrix and define these modules as *tainted*. Then we test to see whether all tainted modules are adjacent to one another within a higher-level module. If so, then we stop and define all tainted modules as part of the loop. If not, then we find (within the hierarchical tree of modules) the module farthest from the root and call it *farthest*. Now we define as tainted the module one level up from farthest and test again to see whether all tainted modules are adjacent. This procedure is repeated until all tainted modules are adjacent. If there is a tie for which module is farthest, we select the module that ranks higher alphabetically (which module is chosen to break the tie does not affect the final result). The algorithm may proceed up to the root of the hierarchical module tree.

Appendix B: Additional regression models with alternative structural control variables

Table B1. Negative Binomial Regressions Predicting Expected Number of Bugs per Application (N = 122)

Independent variables	Model 1 controls	Model 2 propagation	Model 3 intrinsic	Model 4 size	Model 5 hierarchical	Model 6 delta	Model 7 full
AGE_{is} !	.140 (.286)	.136 (.302)	.084 (.322)	.111 (.316)	.046 (.326)	-.046 (.356)	-.033 (.361)
$DAYS\ BEFORE_{is}$!	.129 (.436)	.148 (.451)	.236 (.446)	.068 (.442)	.234 (.442)	.350 (.480)	.345 (.469)
$DAYS\ AFTER_{is}$!	1.348*** (.332)	1.346*** (.350)	1.367*** (.358)	1.387*** (.369)	1.038*** (.395)	1.308*** (.428)	1.270*** (.407)
$NEWNESS_{is}$!	2.247 (2.044)	2.319 (2.088)	1.933 (1.929)	2.502 (1.966)	2.032 (1.990)	2.103 (2.084)	1.871 (1.913)
$IMPLICIT\_BUGS_{is}$!	-1.998 (2.330)	-2.128 (2.349)	-2.836 (2.365)	-2.316 (2.382)	-3.228 (2.446)	-3.064 (2.574)	-3.559 (2.556)
$NUM\_MODULES_{is}$!	6.990 (6.762)	8.104 (6.737)	12.396* (6.852)	9.713 (6.676)	11.640* (6.947)	11.272 (7.121)	14.125** (7.133)
$AVG\_MOD\_ABST\_DEV_{is}$	-.807 (2.421)	-2.783 (2.763)	-3.275 (2.829)	-3.692 (2.861)	-1.920 (2.827)	-2.146 (2.911)	-2.604 (2.958)
AVG_CC_{is}	.066*** (.024)	.067*** (.025)	.072*** (.026)	.064** (.027)	.067** (.032)	.069*** (.025)	.072** (.028)
$SLOC_{is}$	-.015 (.010)	-.016 (.024)	-.010 (.024)	-.015 (.024)	-.018 (.024)	-.020 (.024)	-.013 (.025)
Direct connectivity (K_{is}) ! [a]		.596 (.598)	.717 (.614)	.612 (.630)	.652 (.621)	.771 (.643)	.747 (.635)
Indirect connectivity (κ_{is}) !		-.017 (.012)	-.030** (.013)	-.024* (.013)	-.020* (.012)	-.022* (.013)	-.030** (.013)
Intrinsic NUM_LOOPS_{is}			.000 (.051)	.069 (.065)		.025 (.062)	.020 (.058)
Intrinsic cyclicality ($P_{I,is}$)			**.034*** (.012)**				**.033** (.013)**
AVG Intrinsic Loop Size ($AVG\_CSIZE_{I,is}$)				.015** (.007)			
Hierarchical NUM_LOOPS_{is}					-.105 (.099)	-.131 (.109)	-.084 (.112)
Hierarchical cyclicality ($P_{H,is}$)					.014 (.009)		
Delta cyclicality ($P_{D,is}$)						-.002 (.012)	.003 (.012)
Log Likelihood	-455.411	-454.433	-451.062	-452.426	-452.643	-453.711	-450.741
McFadden's pseudo R^2	0.0739	0.0760	.0829	.0801	0.0796	0.0775	0.835
Wald Chi-sq	96.39***	99.39***	103.51***	97.7***	101.29***	98.87***	104.86***

*< .1 ** < .05 *** < .01 (two-tailed). Standard errors are shown between parentheses.

! Coefficients are multiplied by 1000 to facilitate exposition of results.

All models include application-specific fixed effects and year effects.

[a] Substantially similar results are obtained if excluding direct connectivity as a control variable.

- A likelihood-ratio test comparing Model 3 against a restricted model excluding $P_{I,is}$ yields a significant improvement in the goodness of fit. T_{LR} = -2 × (-454.427 + 451.062) = 6.729 and $\chi^2(1)$, which results in $p < 0.01$.
- A likelihood-ratio test comparing Model 7 against a restricted model excluding $P_{I,is}$ yields a significant improvement in the goodness of fit.T_{LR} = -2 × (-453.711 + 450.741) = 5.940 and $\chi^2(1)$, which results in $p < 0.015$.

Table B2.Hierarchical Poisson Regressions Predicting Number of Bugs per Component (*N* = 28395)

Independent variables	Model 1	Model 2	Model 3	Model 4	Model 5	Model 6	Model 7
Coefficients of system-level variables							
Number unfixed $bugs_{is}$ †	-.512 (.708)	-.663 (.824)	-.615 (.797)	-.626 (.790)	-.604 (.794)	-.618 (.797)	-.629 (.791)
AGE_{is} †	-.112 (.091)	-.075 (.084)	-.075 (.084)	-.077 (.085)	-.078 (.085)	-.076 (.084)	-.078 (.084)
$DAYS\ BEFORE_{is}$ †	.086 (.113)	.099 (.120)	.096 (.118)	.098 (.118)	.097 (.119)	.096 (.118)	.098 (.118)
$DAYS\ AFTER_{is}$ †	.318*** (.093)	.264*** (.094)	.262*** (.093)	.266*** (.093)	.265*** (.093)	.262*** (.093)	.265*** (.093)
$NEWNESS_{is}$ †	.769 (.912)	.805 (.830)	.767 (.825)	.757 (.824)	.750 (.827)	.773 (.824)	.762 (.824)
$IMPLICIT\_BUGS_{is}$ †	.359 (1.087)	.802 (1.021)	.809 (1.017)	.816 (1.020)	.813 (1.029)	.810 (1.016)	.817 (1.019)
$NUM\_MODULES_{is}$ †	-3.070 (2.080)	-2.367 (1.822)	-2.325 (1.836)	-2.374 (1.855)	-2.381 (1.866)	-2.341 (1.834)	-2.385 (1.853)
$AVG\_MOD\_ABSTR\_DEV_{is}$	-2.533 (7.842)	-6.619 (8.589)	-6.753 (8.410)	-6.401 (8.319)	-6.300 (8.340)	-6.845 (8.387)	-6.489 (8.307)
AVG_CC_{is}	.104** (.052)	.098* (.052)	.097* (.051)	.098* (.052)	.098* (.052)	.097* (.051)	.098* (.052)
$SLOC_{is}$	-.042 (.026)	-.035 (.023)	-.034 (.023)	-.034 (.024)	-.034 (.024)	-.034 (.023)	-.034 (.024)
Direct connectivity (K_{is}) !	1.314 (1.515)	.911 (1.434)	.743 (1.423)	.754 (1.429)	.729 (1.439)	.753 (1.422)	.761 (1.428)
Indirect connectivity (κ_{is}) !	-.027 (.045)	-.008 (.042)	-.009 (.042)	-.007 (.042)	-.008 (.043)	-.009 (.042)	-.007 (.042)
$Intrinsic\ NUM\_LOOPS_{is}$	.099 (.154)	.092 (.163)	.097 (.160)	.098 (.159)	.103 (.160)	.098 (.160)	.098 (.159)
Intrinsic cyclicality $(P_{I,is})$	-.045 (.047)	-.029 (.042)	-.034 (.042)	-.036 (.043)	-.037 (.043)	-.037 (.042)	-.038 (.043)
Hierarchical NUM_LOOPS_{is}	.090 (.306)	.038 (.324)	.057 (.318)	.056 (.317)	.047 (.320)	.058 (.318)	.057 (.317)
Delta cyclicality $(P_{D,is})$	-.011 (.027)	.000 (.021)	-.001 (.022)	-.002 (.022)	-.002 (.022)	-.003 (.021)	-.004 (.022)
Table continues on the next page							

† Coefficients multiplied by 100 to facilitate exposition of results.

Table B2. Cont.

	Model 1 cont.	Model 2 cont.	Model 3 cont.	Model 4 cont.	Model 5 cont.	Model 6 cont.	Model 7 cont.
Coefficients of component-level non-structural control variables							
C_AGE_{csi}!		-.002 (.047)	-.029 (.047)	-.046 (.047)	-.036 (.047)	-.015 (.047)	-.034 (.047)
$C\_EXPL\_CHANGES_{csi}$		.096*** (.006)	.091*** (.006)	.090*** (.006)	.092*** (.006)	.091*** (.006)	.090*** (.006)
$C\_IMPL\_CHANGES_{csi}$		.092*** (.013)	.067*** (.014)	.066*** (.014)	.066*** (.014)	.067*** (.014)	.066*** (.014)
$C\_CUM\text{-}COMMITTERS_{csi}$		-.007 (.014)	.007 (.014)	.015 (.014)	.006 (.014)	.007 (.014)	.014 (.014)
$C\_CUM\text{-}AUTHORS_{csi}$		.182*** (.008)	.154*** (.008)	.148*** (.008)	.155*** (.008)	.153*** (.008)	.148*** (.008)
$C\_CUM\_CHANGES_{csi}$		-.021*** (.002)	-.019*** (.002)	-.019*** (.002)	-.019*** (.002)	-.019*** (.002)	-.019*** (.002)
$C\_MOD\_ABSTR\_DEV_{csi}$		.989*** (.150)	1.042*** (.151)	1.052*** (.152)	1.008*** (.152)	1.077*** (.152)	1.081*** (.153)
$C\_AVG\_CC_{csi}$		.054*** (.010)	.034*** (.011)	.036*** (.011)	.035*** (.011)	.036*** (.011)	.038*** (.011)
C_SLOC_{csi}		.538*** (.083)	.272*** (.094)	.243** (.096)	.236** (.096)	.277*** (.094)	.249*** (.095)
Coefficients of component-level structural control variables							
$C\_OUT\text{-}DEGREE_{csi}$!			-5.963*** (2.137)	-5.801*** (2.143)	-5.876*** (2.133)	-5.895*** (2.142)	-5.753*** (2.146)
$C\_IN\text{-}DEGREE_{csi}$!			23.522*** (2.879)	23.357*** (2.951)	23.870*** (2.919)	23.287*** (2.870)	23.156*** (2.941)
$C\_OUT\text{-}INDIRECT_{csi}$!			.402 (.326)	-.587 (.387)	-.540 (.481)	.270 (.330)	-.646* (.388)
$C\_IN\text{-}INDIRECT_{csi}$!			2.071*** (.372)	1.280*** (.407)	1.433*** (.443)	1.971*** (.374)	1.236*** (.407)
Coefficients of component-level predictor variables							
$\boldsymbol{INTRINSIC_{csi}}$				**.289*** (.059)**			**.274*** (.060)**
$C\_INTRINSIC\_SIZE_{csi}$					.004*** (.002)		
$\boldsymbol{HIERARCHICAL_{csi}}$						**.230*** (.082)**	**.194** (.083)**
Log Likelihood	-6722.3	-5772.0	-5694.1	-5682.4	-5690.4	-5689.9	-5679.5
McFadden's pseudo R^2	0.0693	0.2009	0.2117	0.2133	0.2122	0.2123	0.2137
Wald Chi-sq	64.06***	2511.6***	2683.8***	2700.0***	2688.1***	2689.5***	2704.8***

* < .1 ** < .05 *** < .01. Standard errors are shown between parentheses.

! Coefficients are multiplied by 1000 to facilitate exposition of results.

All models include version and application-specific random effects and fixed year effects

- A likelihood-ratio test comparing Model 4 and Model 3 yields a significant improvement in the goodness of fit. $T_{LR} = -2 \times (-5694.1 + 5682.4) = 23.337$ and $\chi^2(1)$, which results in $p < 0.00001$
- A likelihood-ratio test comparing Model 6 and Model 3 yields a significant improvement in the goodness of fit. $T_{LR} = -2 \times (-5694.1 + 5689.9) = 8.283$ and $\chi^2(1)$, which results in $p < 0.004$
- A likelihood-ratio test comparing Model 7 and Model 4 yields a significant improvement in the goodness of fit. $T_{LR} = -2 \times (-5682.4 + 5679.5) = 5.717$ and $\chi^2(1)$, which results in $p < 0.017$

Europe Campus
Boulevard de Constance
77305 Fontainebleau Cedex, France
Tel: +33 (0)1 60 72 40 00
Fax: +33 (0)1 60 74 55 00/01

Asia Campus
1 Ayer Rajah Avenue, Singapore 138676
Tel: +65 67 99 53 88
Fax: +65 67 99 53 99

www.insead.edu

Printed by INSEAD

INSEAD

The Business School
for the World®