Linking Cyclicality and Product Quality

Manuel SOSA
Jürgen MIHM
Tyson BROWNING

2012/137/TOM
(Revised version of 2012/103/TOM)

Linking Cyclicality and Product Quality

Manuel Sosa *

Jürgen Mihm**

Tyson Browning***

Revised version of 2012/103/TOM

(This paper is accepted for publication in *Manufacturing & Service Operations Management*)

We appreciate support from Lattix, Inc., which provided us with the software used to document the architecture of the products and sequence our matrices to calculate some of our control variables. We thank Neeraj Sangal and Frank Waldman for insightful feedback throughout this research and Christoph Loch for his comments on an earlier version of this paper. We also want to thank Bhavani Shanker, Aman Neelappa, and Gaurav Shahlot for their research assistance. We appreciate inputs received at research seminars at Rady School of Business (University of California, San Diego), London Business School, IESE Business School, Judge School of Business (University of Cambridge), Georgia Tech College of Management, Singapore Management University, and the University of Southern Denmark. The first two authors are grateful for the financial support from the INSEAD R&D committee (grants no. 2520-360 and 2520-519). The third author is grateful for support from the Neeley Summer Research Award Program from the Neeley School of Business at TCU and a grant from the U.S. Navy, Office of Naval Research (grant no. N00014-11-1-0739). Finally, we would like to thank the entire editorial team for the insightful comments and clear guidance throughout the review process.

* Associate Professor of Technology and Operations Management at INSEAD, 1 Ayer Rajah Avenue, Singapore 138676, Singapore. Email: manuel.sosa@insead.edu

** Associate Professor of Technology and Operations Management at INSEAD, Boulevard de Constance 77305 Fontainebleau, France. Email: jurgen.mihm@insead.edu

*** Associate Professor, Department of Information Systems and Supply Chain Management at Neeley School of Business, Neeley School of Business, TCU Box 298530 Fort Worth, Texas 76129, USA. Email: t.browning@tcu.edu

Abstract

This paper examines the impact of architectural decisions on the level of defects in a product. We view products as collections of components linked together to work as an integrated whole. Previous work has established *modularity* (how decoupled a component is from other product components) as a critical determinant of defects, and we confirm its importance. Yet our study also provides empirical evidence for a relation between product quality and *cyclicality* (the extent to which a component depends on itself via other product components). We find cyclicality to be a determinant of quality that is distinct from, and no less important than, modularity. Extending this main result, we show how the cyclicality–quality relation is affected by the centrality of a component in a cycle and the distribution of a cycle across product modules. These findings, which are based on analysis of open source software development projects, have implications for the study and design of complex systems.

Keywords: Product architecture; Cycles; Modules; Iterative problem solving; Defects

Linking Cyclicality and Product Quality

This paper examines the impact of architectural decisions on the level of defects in a product. We view products as collections of components linked together to work as an integrated whole. Previous work has established *modularity* (how decoupled a component is from other product components) as a critical determinant of defects, and we confirm its importance. Yet our study also provides empirical evidence for a relation between product quality and *cyclicality* (the extent to which a component depends on itself via other product components). We find cyclicality to be a determinant of quality that is distinct from, and no less important than, modularity. Extending this main result, we show how the cyclicality–quality relation is affected by the centrality of a component in a cycle and the distribution of a cycle across product modules. These findings, which are based on analysis of open source software development projects, have implications for the study and design of complex systems.

Keywords: Product architecture; Cycles; Modules; Iterative problem solving; Defects

1 Introduction

Designing high-quality products poses important managerial challenges for organizations. This paper studies the relationship between the decisions that establish a product's architecture and their consequences for product quality (cf., Ulrich and Eppinger 2008). Studying this architecture–quality relationship is vital because, despite the wealth of research on product architecture, we still do not understand many aspects of how architectural decisions actually affect product quality (Henderson and Clark 1990, Ulrich 1995, Baldwin and Clark 2000, Sosa et al. 2004, Ramachandran and Krishnan 2008). This paper focuses on one particular architectural property, cyclicality, whereby components depend on themselves via other components. Our key theoretical and empirical objective is to study whether and how cyclicality is related to the levels of defects in a product.

In addressing our research goal, we first confirm previous results suggesting that modularity, the architectural property most salient in the literature, prevents defects in a product. Then, as our key contribution, we establish empirically that cyclicality is a distinct architectural determinant of the level of product defects and that its effect on product quality is as sizeable as the effect of modularity. Finally, we deepen our understanding of the cyclicality–quality relationship by examining how product defects are related to different facets of cyclicality, such as the centrality of each component in a cycle and the distribution of a cycle across product modules.

In examining this architecture–quality relationship, we consider a product (hardware or software) as a web of interconnected components (as in Clarkson et al. 2004, MacCormack et al. 2006, Braha and Bar-Yam 2007, Sosa et al. 2007b, Gokpinar et al. 2010). Previous research on network-based architecture has focused on (component) *modularity*, the extent to which a component is decoupled (or independent) from other components in the product, as the main architectural feature of interest (Card and Agresti 1988, Clarkson et al. 2004, MacCormack et al. 2006, Sosa et al. 2007b). This research has empirically established that modularity is associated with the design of less defective products (Card and Agresti 1988, Briand et al. 1999, Aggarwal et al. 2007, Burrows et al. 2010).

However, there are good reasons to believe that focusing on modularity, as the main architectural determinant of quality, is too simplistic. Recognizing cyclicality as another fundamental architectural property is important for both conceptual and empirical reasons. Conceptually, component cycles inhibit the proper decomposition of design problems (because there is no self-evident sequence in which to design, build, and test components involved in cycles) and therefore require iterative problem solving, which results in cognitive and organizational challenges (Smith and Eppinger 1997a, Smith and Eppinger 1997b, Mihm et al. 2003). In contrast, if there are no cycles then problems can be properly decomposed and be solved in a serial manner, one subproblem at a time (Eppinger et al. 1994). Empirically, component modularity and cyclicality can co-occur and be correlated, making it difficult to determine which factor is the principal driver of the observed effects.

In order to develop a thorough understanding of the cyclicality–quality relationship, we investigate how various aspects of cyclicality relate to product quality. Is the extent to which a component is involved in cyclical dependency patterns a significant determinant of defects? Is the effect of cyclicality on quality as substantial as the effect of modularity? Are all components in a cycle equally prone to defects? In any system of even moderate complexity, components are typically organized into modules (Simon 1996). Does the organization of components into modules affect the relationship between cyclicality and quality? By addressing these questions, we aim to close an important gap between the information systems literature and the operations management literature. The former has not considered cyclicality to be an important determinant of defects. And even though the latter has developed methods to uncover cyclicality in complex development settings, its research has not yet linked cyclicality to the levels of product defects.

Previous research in information systems and computer science has investigated the determinants of defects in software products (e.g., Card and Agresti 1988, Chidamber and Kemerer 1994, Briand et al. 1999, Aggarwal et al. 2007). It has classified such determinants into two broad categories: intracomponent and intercomponent. (Intuitively, a component in a software product is a cohesive collection of source code.) Intracomponent determinants are concerned with aspects characterizing the individual component whereas intercomponent determinants are concerned with components' interactions. With respect to the most salient intracomponent determinants of quality, previous findings suggest that larger and more internally complex components are likely to have a higher number of defects (McCabe 1976, Henry and Selig 1990, Kan 1995). As for intercomponent determinants, previous research on software architecture has focused on the modularity of a component as the main feature of interest (e.g., MacCormack et al. 2006, MacCormack et al. 2008). If a component is more modular—that is, if a component depends on few other components—then it is likely to exhibit a lower level of defects (Card and Agresti 1988, Card and Glass 1990, Chidamber and Kemerer 1994, Kan 1995, Briand et al. 1999, Aggarwal et al. 2007). Despite the multitude of determinants that research in information systems has explored, it has yet to recognize the importance of architectural cyclicality as a determinant of defect proneness.

The operations management literature has also explored different aspects of modularity (Ulrich 1995, Baldwin and Clark 2000). Yet beyond modularity, researchers in this area have developed representations and methods for identifying cyclical structures when modeling the new product development process as a collection of networked tasks (Steward 1981, Eppinger et al. 1994, Mihm et al. 2003). This stream of research has led to a critical insight: tasks that are interrelated in a cyclical manner tend to require more managerial attention (Smith and Eppinger 1997a, Smith and Eppinger 1997b) because cyclical structures entail design iterations that can affect the time required to complete a development effort (Mihm et al. 2003, Braha and Bar-Yam 2007). However, this literature has taken a modeling approach to formulate its predictions; therefore, it has established no empirical links between cyclicality and outcome measures (such as product quality) and has not been able to explore the cyclicality construct in ways that would build a more nuanced understanding of how its different facets affect product quality.

Given how difficult it is to capture a comprehensive longitudinal data set that contains product architecture, quality, and resource attributes, it is hardly surprising that the cyclicality–quality relationship has escaped rigorous empirical examination. We overcome this challenge by taking advantage of the emergence of open source software to build a substantial data set that includes 28,394 observations of 7,103 product components in 111 releases of 17 Java-based applications developed by the Apache Software Foundation. We focus on open source software applications for several reasons: they are complex systems in which cyclicality patterns are present (yet difficult to identify); they exhibit relatively fast rates of change (much like fruit flies in studies of biological evolution); and their source code constitutes an accessible, efficient, reliable, and standardized means of capturing all the architectural features relevant to our study of product architecture. Moreover, open source development settings typically feature centralized systems used for tracking and managing of quality issues.

2 Theory and Hypotheses

We begin this section by taking a network view of product architecture in order to (a) establish, in a calibration hypothesis, component modularity as the most studied characteristic of product architecture and (b) define cyclicality and hypothesize about its consequences for product quality. Yet, the network view, with its focus on how product components connect with each other, ignores the hierarchical organization of components into modules. Hence, to gain a more comprehensive view of the effects of cyclicality on quality, we close the section by examining how a product's hierarchical module structure influences the cyclicality-quality relationship.

A network view of a product's architecture considers products as interlinked components or subsystems. This view emphasizes the role played by dependencies among product components (Eppinger and Browning 2012). A dependency is established by *any* direct relationship between two product components (Sosa et al. 2007b): *spatial* dependencies are the result of two components requiring a specific spatial configuration, *structural* dependencies arise when there is a required transmission of mechanical loads between components, *energy* dependencies emerge due to energy flow requirements between components, *material* dependencies capture the flow of material (e.g., water, oil, steam, air) between components, and finally *information* dependencies map how different components interact to process information. Mapping the totality of dependency patterns for a complex system such as an aircraft engine requires capturing multiple types (Sosa et al. 2007b). However, software products are particular in that their components are linked exclusively by information flows. For instance, MacCormack et al. (2006) captured the architecture of large, open source software systems by mapping how their components connected with each other via function calls.

Different systems vary significantly in their dependency patterns, and these patterns affect the level of difficulty experienced by development organizations when designing, building, and testing products. It is only natural that such difficulties should affect product quality. Figure 1 depicts the three basic patterns of dependencies that can be observed within a system (Thompson 1967, Eppinger et al. 1994). In panel (a) of the figure, the three components are independent and thus have no effect on each other (barring resource constraints); hence each component can be designed independently. With respect to quality, this is a trivial case. Because there is no dependency among the components, neither are there any network-based differences among them. Therefore, any variation in their number of defects must be due to their inherent characteristics (e.g., their internal complexity) and not to network-related aspects. We argue next that connectivity per se has a significant effect on product quality, with specific mechanisms depending on the type or pattern of connectivity.

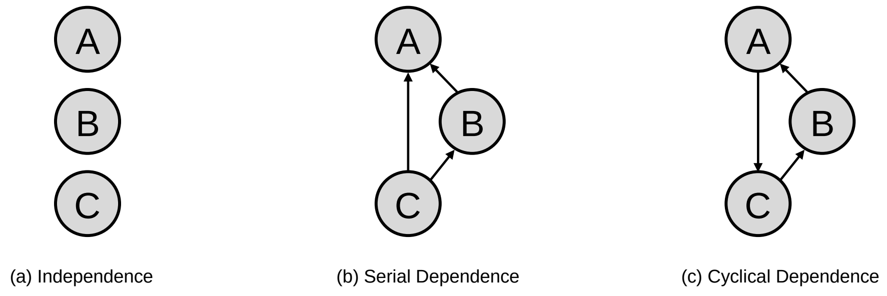

Figure 1. Three dependency structure patterns for components A, B, and C

2.1 Effects of Component Modularity: A Calibration Hypothesis

In Figure 1(b), components are connected in a serial manner (i.e., component C provides input to components A and B, and B provides input to A). This pattern of dependency is consistent with a fundamental characteristic of directed graphs, *reachability*: the property of being able to "walk" from one node to another via a set of directed "edges" (Harary 1969, Gould 1988). From a managerial viewpoint, reachability implies that components should be designed in a serial order (i.e., C, B, and then A in Figure 1(b)). Components A and C have opposite reachability characteristics: A is reached by all other components whereas C reaches all other components. More generally, a component in any directed design chain is likely to be positioned in either the so-called upstream or downstream end of the chain. Previous work in both the product architecture and software development literatures has shown that such positioning affects the component's defect proneness.

First we consider downstream components. An argument well grounded on the notion of design change propagation has been established and empirically verified for why downstream components are particularly error-prone. During a typical development process, the design of each component changes repeatedly. Such design changes may propagate from upstream components (here, component C) to downstream components (component A) via their dependencies (Clarkson et al. 2004, Giffin et al. 2009). Hence, downstream components of a design chain have been shown, for both hardware and software, to be at increased risk of induced design changes (Card and Agresti 1988, Krishnan et al. 1997, Terwiesch et al. 2002). Design changes in downstream components, triggered to adapt to design changes made in upstream components, are likely to destabilize the downstream components' designs and hence increase the risk of quality issues (Card and Agresti 1988, Terwiesch and Loch 1999, Burrows et al. 2010).

The argument for downstream components being more defect-prone is fully in line with the expected benefits of *component modularity*. A downstream component with low reachability is largely decoupled from other components in the system and thus enjoys a high level of component modularity, which by the preceding arguments should have a positive effect on its quality (Ulrich 1995, Baldwin and Clark 2000, Sosa et al. 2007b, MacCormack et al. 2008).

In contrast to their downstream counterparts, a conclusive theory for the error proneness of upstream components is lacking. Upstream components do not pose the challenge of dealing with design changes induced by the fate of other components (Braha and Bar-Yam 2007). Not surprisingly, then, several empirical studies in the information systems literature have found no significant relationship between upstream components and defect proneness (Kan 1995, Briand et al. 1999, Aggarwal et al. 2007).

Thus, consistent with past findings in the literature we posit the following calibration hypothesis that examines the relationship between downstream component modularity and defect levels; we leave the effect to upstream component modularity as an empirical question.

H1 (downstream component modularity helps)*: The number of defects exhibited by a focal component is positively associated with the extent to which such a component depends upon other product components.*

2.2 The Effect of Component Cyclicality

Figure 1(c) illustrates a second fundamental characteristic of directed graphs and the network-based view of architectures: *cyclicality* (Harary 1969, Gould 1988).[1] In this panel, all of the components are involved in a cyclical dependence. Component A depends on input from B, which depends on input from C, which depends on input from A. We use the term *in-cycle component* for any component that is part of a cycle, where "cycle" is the set of components for which a dependency path exists from each component to every other. (A component that is not part of such a cycle is referred to as a *non-cycle component*.) Thus, an in-cycle component depends on itself via other components in the cycle, and component cyclicality is the extent to which a component depends on itself via other components—i.e., the number of components in the cycle.

Developing in-cycle components in complex systems is especially challenging because (in contrast to non-cycle components) they form problem structures that require i) iterative problem solving, and ii) additional coordination efforts.

Because cyclical dependency patterns imply no natural sequence in which components can be conceptualized and designed, developers must address the development of in-cycle components in an iterative fashion. Iterative problem solving can occur in either sequential or parallel fashion (or any combination of them). Developers may iterate *in a sequential fashion* by myopically considering and redesigning in-cycle components one at a time until the entire cycle design converges to a commonly accepted solution (Smith and Eppinger 1997b, Mihm et al. 2003). Alternatively, developers may iterate in parallel by considering and redesigning *all in-cycle components at once* until an overarching solution for the entire system is achieved (Smith and Eppinger 1997a, Mihm et al. 2003). Either approach, however, is likely to increase the error-proneness of in-cycle components relative to non-cycle components. Sequential iteration is likely to suffer from numerous repeated component redesigns due to feedback signals coming from design revisions of other components in the cycle (Mihm et al. 2003). Such repeated

[1] There are graph-theoretic reasons for the prominence of reachability and cyclicality within a network view of product architectures. A product architecture can be represented as a graph—that is, a set of nodes and edges. All graph properties can be defined with respect to walks (where a "walk … is a finite alternating sequence of [nodes] and edges that begins with the [node] *x* and ends with the [node] *y* and in which each edge in the sequence joins the [node] that precedes it in the sequence to the [node] that follows it in the sequence" (Gould 1988, p. 8). In practice, most graph properties are defined in terms of walks. There are two fundamental types of walks, closed and open (e.g., Gould 1988, p. 9). *Open walks* determine whether (and how) the starting node reaches the end node, so they are well suited to characterizing different reachability aspects. *Closed walks* define cycles. Graphs may thus be categorized as either cyclical or acyclical (Wasserman and Faust 1994) depending on whether they do or do not, respectively, contain at least one closed walk.

redesigns (which are absent or less likely in non-cycle components) increase the risk of errors during development. In parallel iteration, all in-cycle components are designed and redesigned at once, which may limit the amount of revisions necessary. However, compared to designing each component one at a time, it entails considerable cognitive effort as the amount of information that needs to be concurrently dealt with increases drastically, which again makes in-cycle components susceptible to errors (Miller 1956). In sum, because in-cycle components are part of difficult-to-decompose design problems that are hard to solve, they are more likely to exhibit more defects than non-cycle components, which form part of linear or independent problem structures that do not require iterative problem solving.

In addition to iterative problem solving, in-cycle components are likely to require more coordination needs than non-cycle components. During the development of complex hardware or software systems, different components are typically developed by different designers or different design teams (Mockus et al. 2000, Sosa et al. 2004, Cataldo et al. 2006, MacCormack et al. 2006). Hence, actors designing components involved in cyclical problem structures are likely to face higher coordination needs than those designing components involved in non-cyclical problem structures because of either the *repeated* design–build–test iterations to which such in-cycle components are likely to be subjected or the concurrency of design efforts. Hence, such components are more likely to suffer from coordination pitfalls and thus have more defects (Gokpinar et al. 2010).

Our arguments so far have focused on the dichotomy between in-cycle and non-cycle components. However, larger cycles are expected to be more error-prone because both iterative problem solving and coordination needs increase with the number of components involved in a cycle. Iterative problem-solving approaches that must address a larger number of components increase either the number of design revisions before reaching convergence or the number of components that need to be considered concurrently (Mihm et al. 2003). Either way larger cycles make iterative problem solving more problematic. Arguments about coordination costs associated with cycles lead to the same conclusion. The greater is the number of components involved in a cycle, the more elements there are that at risk of a coordination breakdown. Hence, we formulate the following hypothesis.

H2 (component cyclicality hurts)*: The number of defects exhibited by a component increases with component cyclicality.*

2.3 The Effect of Cyclicality Centrality

Although component cyclicality is constant for all components involved in a specific cycle, such in-cycle components may differ in their network positions within that cycle, which may have further consequences for the number of defects affecting each of those in-cycle components. Even though all components in a cycle are interconnected, some components may occupy more central positions in the cycle's network structure than do others.

Centrality, a concept developed and extensively studied in the context of social networks (Freeman 1979, Wasserman and Faust 1994), refers to the identification of the most important or prominent nodes in the network. Central nodes exhibit the properties of a center node in a star-shaped graph (Freeman 1979). They exhibit a maximum number of direct connections to, a minimum distance to, and a maximum likelihood of being in between all other nodes in the graph. Since we study the notion of cyclicality, we limit the boundaries of the graph to the components forming a cycle and the dependencies among them.

Given the above properties of central components in a cycle, they are more likely to be involved in a sequence of sequential iterative problem solving than are peripheral cycle components. Hence, central in-cycle components are at higher risk of experiencing the unanticipated nonlinear effects that characterize sequential iterative problem solving. To see this, consider all paths that link back to a focal component in a cycle (touching another component in the cycle at most once). For any in-cycle component, at least one such feedback path must exist; if there is more than one path, then the feedback along all of those paths needs to be incorporated into the component design before the design iteration can converge. In other words, more paths means more feedback and thus a greater chance that the iteration either does not converge or produces unwanted results (Mihm et al. 2003). Components that are more central in the cycle are touched by many such paths (Wasserman and Faust 1994), which leads to our next hypothesis.

H3 (increasing cyclical centrality hurts worse)*: The number of defects exhibited by a component is increasing in its centrality within the cycle.*

2.4 The Effect of Grouping Components into Modules

Complex engineered systems are typically conceived in terms of a hierarchy of modules and submodules—that is, "in a boxes-within-boxes form" (Simon 1996, p.128). It is therefore reasonable to examine how the decisions to group components into modules influence the relation between cyclicality and quality. An example of this situation is illustrated by Figure 2, which gives the hierarchical view of an architecture by grouping the membership of Figure 1(c)'s components as well as three additional components (A1, B1, C1) into three distinct, two-component modules: M_A, M_B, and M_C (the modules are shaded for visual distinction).

Although modules can be formed in many different ways, the dominant modularization strategy is to (i) group functionally similar and highly interdependent components into modules and (ii) minimize the dependence of modules on each other (Parnas 1972, Simon 1996, Baldwin and Clark 2000). Thus designers typically concentrate connections among components within modules while limiting connections across module boundaries (Baldwin and Clark 2000); this is known as the principle of "near decomposability" (Simon 1996). In software development terms, following this strategy maximizes module cohesion and minimizes coupling across modules (Stevens et al. 1974, Chidamber and Kemerer 1994, Briand et al. 1999). This grouping principle facilitates problem solving by decoupling modules and thereby allowing the design–build–test process for each module to "be carried out with some degree of independence of the design of others" (Simon 1996, p. 128).

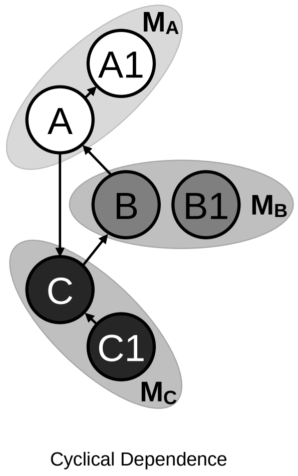

Figure 2. Six-component system grouped into three modules

We expect that a cycle (such as the one in Figure 2) whose components are distributed across multiple modules will be especially prone to defects. Such defect proneness is a consequence of increased coordination requirements and decreased visibility, two factors that are more predominant across than within module boundaries. Modules in the product domain usually mirror developer groups in the organization domain (Henderson and Clark 1990, Sosa et al. 2004, MacCormack et al. 2011). Such mirroring is even true for open-source software development projects, which often do not exhibit formal organizational groupings. Developers working on different modules are likely to be cognitively more "distant" than developers developing components that are functionally similar since "developers tend to work on problems that are identified with areas of the code they are most familiar. Some work on the product's core services, while others work on particular features that they developed" (Mockus et al. 2000, p. 4). Specialization largely dictates how developers self-assign to modules. The resulting organizational distance that is fostered by the module structure implies a lack of familiarity of the interdependent actors involved; coordination breakdowns result and the risk of component defects increases (Staats 2012).

In addition to requiring coordinated communication among developers, dependencies across module boundaries run the risk of remaining unidentified (Allen 1977, Sosa et al. 2004) because modules are normally considered to be self-contained entities. Hence cycles whose components are distributed across multiple modules are less likely to be identified as in-cycle components. In that case, efforts to plan and manage the iterative problem solving required by such cycles would be compromised (Pich et al. 2002), which in turn would lead to higher levels of defects. All these considerations lead to our final hypothesis.

H4 (modules failing to encapsulate cycles hurt)*: The number of defects exhibited by an in-cycle component is increasing in the number of modules involved in its cycle.*

3 Empirical Study: Open source Software Development

To test our hypotheses, we studied open source, Java-based software applications from the Apache Software Foundation (http://www.apache.org/), which is one of the largest, most established, best studied open source communities of developers and users who share values and a standard development process (Roberts et al. 2006). We chose to focus on Java because it is one of the most widely used object-oriented programming languages and because its source code captures component dependencies and module constituents in an explicit and structured way.

3.1 Data

As our initial database, we identified 69 Java-based development projects at the Apache Software Foundation in mid-2008. An effective examination of the causal relationship between architectural characteristics and quality requires a longitudinal data set, so we focused on the 37 applications for which successive major releases were available. From these 37, we selected those for which we could access all necessary data sources: bug-tracking systems (to determine the number of bugs), precompiled ("prebuilt") code in so-called JAR files (to codify product architecture features), original source code (to measure such product-related attributes as source lines of code, SLOC), release notes (to determine product-related innovative features), and version management tools (to associate bugs with the components they affected). These filters left us with a set of 28,394 observations of 7,103 product components across 111 releases (versions) of 17 applications—an average of 256 components per version.

We compiled an integrated data set from those five sources. First, we examined Apache's *Bugzilla* and *Jira* bug-tracking systems to obtain the bugs associated with each version. Each of these systems allows users and developers to enter bug reports, which are classified in terms of their potential severity and are 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, and the databases of these bug-tracking systems record the status and resolution of each bug associated with any version. We developed a web crawler to automate the gathering of these data. Second, we downloaded the precompiled version of each major release of each application (available as a JAR file) from the Apache archives or the application's website; we did not use minor releases because they typically involve relatively small changes. We used

LDM, a commercially available software application developed by Lattix, Inc. (www.lattix.com), to build a design structure matrix (DSM) representation from the source code (as captured in the JAR file) and to extract the module membership of components. Third, we downloaded the original source code for each version. This step involved locating and downloading more than 110 source packages and—since the correspondence between Java classes and files is nearly one-to-one—examining more than 28,000 source code files. Accessing these files was necessary in order to measure various dimensions of the complexity of each Java class. Fourth, we consulted each version's official release notes to gather information on newness, age, and other important application-level controls. Finally, we developed a second web crawler to retrieve and extract data from the "change log" files of the version control tools, the so-called subversion (SVN) repositories, in order to link each bug to the Java class(es) that it affected—in other words, we counted all of the bugs that affected each Java class, noting that some bugs affected more than one class. From the SVN repositories we were also able to mine data about timing and authorship. Our analysis was based on all the reported bugs that had been fixed or were in the process of being fixed (they are all "patched" bugs); we did not include "unpatched" bugs owing to the lack of information on which components they affected. (No selection bias was thereby introduced into our analysis because we were able to consider all components in all of the product versions in our sample.) However, we did control for the number of unpatched bugs associated with the version to which a component belonged. Finally, we also checked that patched and unpatched bug groups were not significantly different with respect to their ratio of severe and non-severe bugs.

3.2 Dependent and Predictor Variables

3.2.1 Dependent Variable: Number of Bugs per Component

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

3.2.2 Independent Variables

We define three sets of independent variables. First, we use measures of fan-out and fan-in to test for the effects of downstream and upstream component modularity, respectively. Second, we discuss how to identify in-cycle components in order to measure component cyclicality and cyclicality centrality (to test H2 and H3). Third, we define a variable that captures precisely how the presence of hierarchical modules encapsulates cycles (to test H4).

Software applications are systems of connected components grouped into modules (Shaw and Garlan 1996, Martin 2002, Sangal et al. 2005). Modeling a system requires defining what constitutes a component. Although we could perform our analysis at the level of methods or even lines of code, we use the "Java class" as our unit of analysis. (A *method* is a self-contained collection of programming instructions that typically includes variable instantiation and control flow statements, such as “if … then” and “while … do” statements; a Java *class* is a collection of methods. A class in our data set contains, on average, ten methods. Our data set contains only Java applications, wherein files and classes are typically coextensive.) There are three reasons for choosing the Java class as our unit of analysis. First, a class tends to provide a set of common functionality (e.g., a set of low-level mathematical functions) maintained as one cohesive piece of software, often in one source file supplied by an individual developer. Second, significant attributes of the architecture are apparent at the class level, rendering further decomposition unnecessary for our purposes. Third, this level of decomposition is consistent with past work on software architecture (e.g., Sangal et al. 2005, MacCormack et al. 2006).

Our measures are based upon a design structure matrix representation of the dependency structure of the components in each version of each application in our sample. A DSM is a square matrix whose rows and columns are both labeled with N components and whose off-diagonal cells indicate the components’ directed dependencies (Browning 2001). A dependency results from a “call” made by one component to another (Sangal et al. 2005, Cataldo et al. 2006, MacCormack et al. 2011).[2] An off-diagonal mark in cell (i, j) of the DSM indicates that the Java class in column j calls the class in row i and hence that the class in column j depends on the class in row i. For example, consider the left panel of Figure 3, which shows a “flat” DSM representation of the entire Ant (version 1.4) application with 160 components and 676 directed dependencies. (The term “flat” signifies that this DSM does not capture the arrangement of components into modules.)

As for metrics, we follow MacCormack et al. (2008) in using fan-out and fan-in to measure the *lack* of component modularity. This is consistent with previous work that studies modularity at the component level and measures component modularity as the lack of dependency among product components (Sosa et al. 2007b, Cabigiosu and Camuffo 2011). First, to measure the lack of downstream component modularity needed to test H1, we calculate *component fan-out* ($C\_FAN\_OUT_{cis}$) as the fraction of product components on which component c depends, either directly or indirectly. With reference to Figure 1(b), for example, the fan-out of component A is 100% because component A depends (directly or indirectly) on all other components in that system; in contrast, the fan-out of components B and C are (respectively) 50% and 0%. In general, component fan-out is derived from the *visibility matrix* V, which is a square binary matrix of size N (where N still denotes the number of components) whose nonzero cells ($v_{i,j}$)

[2] Dependencies include: *invocations* (static, virtual, and interface), which allow for various types of method calls; *inheritances* (extensions and implementations), which allow a class to extend or define new behaviors; *data member references*, which refer to the field of a class; and *constructs*, a method call for creating an object. We include these dependencies because they are typically integral to the system’s design and because developers create them deliberately. They comprise the various ways in which classes “connect” with other classes in a Java-based software application.

indicate that component j depends on component i, either directly or indirectly, via any number of intermediary components (Sharman and Yassine 2004, MacCormack et al. 2006). This matrix V is the binary sum of the first N powers of the flat DSM (applying Boolean matrix multiplication). The middle panel of Figure 3 shows the visibility matrix for Ant 1.4. We calculate fan-out component visibility as follows (cf. MacCormack et al. 2008):

$$C\_FAN\_OUT_{cis} = \frac{\sum_k v_{kc}}{N-1},$$

where the numerator is the sum of all nonzero cells in column c of V. This measure captures the fraction of components that might affect c as their changes propagate to c.

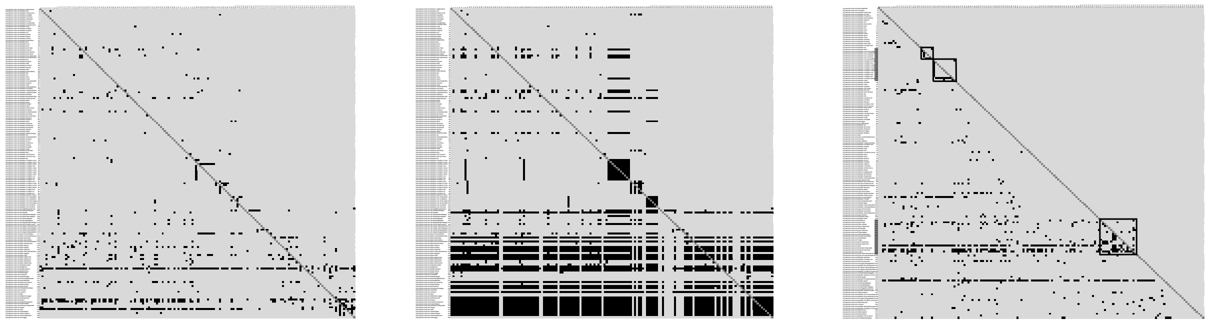
Figure 3. Representations of Ant 1.4: flat DSM (left panel); visibility matrix (middle panel); sequenced DSM (right panel)

Although we do not explicitly hypothesize for the effects of upstream component modularity, we still control for its potential existence. In order to measure the lack of upstream component modularity, we calculate *component fan-in* ($C\_FAN\_IN_{cis}$) as the fraction of components that depend (either directly or indirectly) on *c*. Referring again to Figure 1(b), the fan-in of component C is 100% because all other components in that system depend (directly or indirectly) on component C; in contrast, the fan-in of components B and A are (respectively) 50% and 0%. We calculate fan-in component visibility as follows:

$$C\_FAN\_IN_{cis} = \frac{\sum_k v_{ck}}{N-1};$$

here the numerator is the sum of all nonzero cells in row c of V. This measure captures the fraction of components that might be affected by a change in component c.

To determine whether a component is involved in a cycle, we apply the procedure described by Warfield (1973) to the DSM, although we substitute Tarjan's (1972) more efficient (linear order of growth) algorithm to identify the unique sets of in-cycle components. The right panel of Figure 3 shows the result of applying this algorithm to the flat DSM of Ant 1.4; it reveals that Ant 1.4 contains three component cycles—the three highlighted blocks along the diagonal—which contain 6, 11, and 18 components, respectively. We can now define the component-level indicator variable: $IN\_CYCLE_{cis} = 1$ if component c belongs to a component cycle of version s of application i (and $IN\_CYCLE_{cis} = 0$ otherwise). Then, to capture component cyclicality, we measure $CYCLE\_SIZE_{cis}$ as the number of components in the cycle to which component *c* belongs. ($CYCLE\_SIZE_{cis}$ = 0 for non-cycle components.) Finally, we measure the centrality of a component within a cycle, $IN\_CYCLE\_DEGREE_{cis}$, as the number of other components in the cycle to which component *c* belongs that are directly connected with component *c* (Freeman 1979, Wasserman and Faust 1994).

To calculate a measure that captures how modules encapsulate cycles, we identify the module structure of the code for version s of application *i*. Since the nested subdirectory structure (and thus hierarchical organization of the source code into nested modules) is captured in the name of each Java class, we are able to associate each component uniquely with a set of hierarchically structured modules. Figure 4 shows a hierarchical DSM representation of Ant 1.4, which overlays the nested module structure onto the dependency structure of the product. Although each component is uniquely assigned to a component module (i.e., a module that contains only components), the existence of "modules of modules" results in the three-level hierarchical module structure shown. Because we study the implications of grouping components into (component) modules as a first-order effect of the architecture's hierarchical aspects, we count the number of cross-module boundaries spanned by the cycle. That number is denoted by the variable $MODULES\_CROSS\_CYCLE_{cis}$.

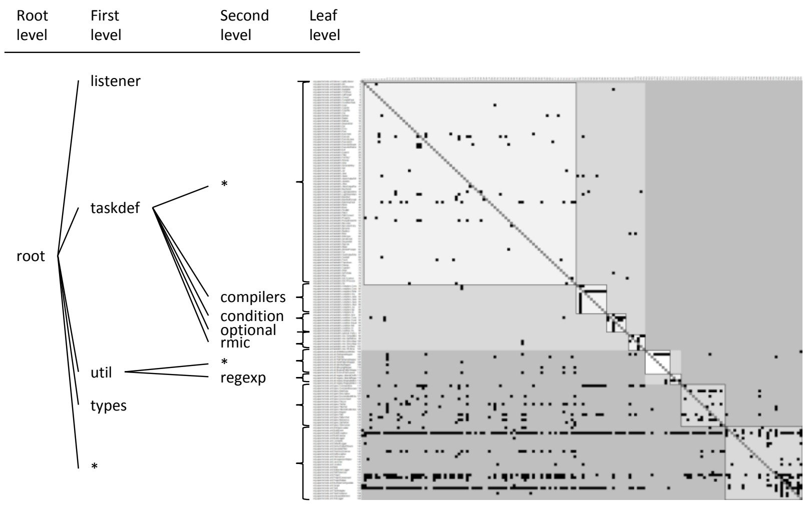

Figure 4. Hierarchical DSM of Ant 1.4, including module boundaries

3.3 Control Variables

Other factors may be related to the number of bugs affecting a component. We include two sets of control variables: system-level and component-level factors (see Table 1, for which a more detailed version is available in appendix A of the online supplement). Table 2 gives descriptive statistics and pairwise correlations for the variables included in the analysis. Among the system-level controls, the breadth and depth of the hierarchical module structure are (as expected) positively correlated; this suggests that, as applications grow in the number of their component modules, they also grow in the number of their modules of modules. However, these two variables are negatively correlated with the application's propagation cost, which indicates that applications using more hierarchical module structures have a dependency structure that is less interconnected (MacCormack et al. 2006). The positive correlation between an application's age and its average cyclomatic complexity is consistent with the conjecture that older applications are made up of components that are internally complex. Among the component-level controls, the strong positive correlations among cumulative changes and cumulative committers and authors indicate that, as expected, workload and use of resources are positively associated with each other. Not surprisingly, the two measures of intracomponent complexity (SLOC and the average cyclomatic complexity of the methods constituting a component) are positively correlated. Finally, the cyclicality variables are, as expected, highly correlated among themselves and also with component fan-in and fan-out. The positive correlations between *IN_CYCLE* and *C_FAN_IN* and *C_FAN_OUT* are consistent with the fact that these three measures depend (in different ways) on the connectivity patterns of the focal component with other components and how these other components interact.

Table 1. Control Variables

System-level factors (version *s* of application *i*)	
$UNPATCHED\_BUGS_{is}$	Number of unpatched bugs (not yet associated with specific components) in this version
AGE_{is}	Number of days between the first version available and this version
$DAYS\_BEFORE_{is}$	Number of days between this and the previous version
$DAYS\_AFTER_{is}$	Number of days between this and the next version
$NEWNESS_{is}$	Number of *new features* (added functionality) and *improvements* (modifications to existing functionality) in this version
APP_SLOC_{is}	Number of kilolines of source code in this version
$APP\_AVG\_CC_{is}$	Average cyclomatic complexity of all methods in this version (*cyclomatic complexity* is the minimum number of linearly independent paths in the control flow graph of a method in a software program; (McCabe 1976))
$NUM\_NOM\_MODULES_{is}$	Number of modules that contain actual components (not simply nested modules) in this version; this variable measures the overall *breadth* of the hierarchical module structure in this version
$HIERARCHY\_DEPTH_{is}$	Maximum number of levels between the leaf (component) levels and the root level in this version; this variable measures the overall *depth* of the hierarchical module structure of this version
$AVG\_INTERFACE\_USAGE_{is}$	Average of Martin's (2002, p. 267) *distance* metric across all modules in this version; this metric gauges the developers' deviation from the "recommended" source code structure that best handles cross-module dependencies
$PROPAGATION\_COST_{is}$	Average fan-in and fan-out visibility of all components in this version (MacCormack et al. 2006) as computed from the visibility matrix *V*
Component-level factors (component *c* of application *i* in version *s*)	
C_AGE_{cis}	Number of days since the component was first included in this application
$C\_EXPL\_CHANGES_{cis}$	Number of nonbug changes (improvements, new features, etc.) *explicitly* associated with this component in this version
$C\_IMPL\_CHANGES_{cis}$	Number of total changes (bugs, improvements, new features, etc.) *implicitly* associated with this component in this version because of their introduction time (explicit assignment is not available)
$C\_CUM\_CHANGES_{cis}$	Cumulative number of changes associated with the component *prior to* this version

$C\_CUM\_COMMITTERS_{cis}$	Cumulative number of committers associated with the component *prior to* this version
$C\_CUM\_AUTHORS_{cis}$	Cumulative number of authors associated with the component *prior to* this version
$C\_INTERFACE\_USAGE_{cis}$	Distance metric (Martin 2002, p. 267) for the component's module in this version
$C\_AVG\_CC_{cis}$	Average cyclomatic complexity (McCabe 1976) of the component's methods in this version
C_SLOC_{cis}	Number of kilolines of source code in this version of the component

4 Analysis and Results

The dependent variable in our analysis counts the number of bugs affecting component c. Because our data has both a hierarchical structure (component c belongs to application i) and a panel structure (component c can be observed in any of several versions, s, of application i), we must use a hierarchical modeling framework with panel data (Raudenbush and Bryk 2002). This type of analysis allows us to test for component-level effects in the presence of system-level covariates while controlling for the lack of independence in observations from the same component and also from the same application. In addition, because our dependent variable is a bug count for component c, we estimate a hierarchical Poisson regression model (Cameron and Trivedi 1998). (Note that our "count" dependent variable does not exhibit signs of overdispersion: its variance is not significantly larger than its mean.) For estimation purposes, we use the *xtmepoisson* procedure recently implemented in Stata 12 with a component-specific random intercept that is nested in its corresponding application-specific random intercept (Raudenbush and Bryk 2002, Rabe-Hesketh and Skrondal 2008). Hence, our baseline model is a random-intercept model of the following form:

$$E[y_{cis} \mid \mathbf{x}_{cis}, \mathbf{z}_{is}] = \exp(\gamma\mathbf{z}_{is} + \beta\mathbf{x}_{cis} + \zeta_{ci} + \zeta_i + \epsilon_{cis}).$$

Consistent with the hierarchical linear modeling approach, this model fits a multilevel, mixed-effects Poisson regression that contains not only "fixed" effects (the β, γ-coefficients), which are analogous to standard regression coefficients, but also "random" intercepts (the ζ-parameters), which are assumed to vary (following a Gaussian distribution) across components and applications. Our regression models predict that the expected number of bugs affecting component c in application i depends exponentially on two sets of linearly independent regressors: a first set $\{\boldsymbol{z}_{is}\}$ defined at the system level and a second set $\{\boldsymbol{x}_{cis}\}$ defined at the component level, both instantiated in version s. We use raw data in our estimations, but our results are robust to both grand-mean and application-level centering (Kreft et al. 1995).

When testing for the hypothesized effects of cyclicality, we enhance our baseline model by including random coefficients for the cyclicality variables of interest with an *unstructured* covariance structure of the random parameters (Singer and Willett 2003, Rabe-Hesketh and Skrondal 2008). These models provide a significantly better fit to our data—than do the corresponding random-intercept models—by relaxing the assumption that cyclicality effects are the same across all the applications in the sample. Intuitively, a random-coefficient model is analogous to a model that includes interaction effects between the variable of interest (here, a cyclicality variable) and a group-level indicator variable (here, an application-level dummy variable). Note that, given the hierarchical nature of our data, we cannot include application-level indicator variables.

Estimates for the β- and γ-coefficients in our final set of regression models are reported in Table 3. Models 1 to 3 are random intercept models while Models 4-7 are random coefficient models. As recommended by Singer and Willett (2003), we included an application-specific random coefficient of the cyclicality variable of interest only if doing so significantly reduces the model's deviance statistic with respect to the nested model that excludes such a coefficient. (In our models, a model's deviance statistic is $-2 \cdot (\text{log-likelihood statistic})$.) In our on-line appendix, we include Table B providing details of how we evaluated the reduction in deviance statistic associated with the inclusion of each application-specific random coefficient of the cyclicality variables of interest. Table B also shows the standard deviation of both component- and application-level main random parameters of all the models shown in Table 3.

4.1 Testing the Hypotheses

Model 1 includes system-level control variables. Most of the coefficients for these control variables are significant, which confirms their relevance. This model shows that components in large applications (as measured by the number of kilolines of the application's source code in version s) and with higher average cyclomatic complexity are likely to exhibit a greater number of defects. This is consistent with the information systems literature, which suggests that both SLOC and cyclomatic complexity are good predictors of the effort required to build, test, and maintain software applications (McCabe 1976, Henry and Selig 1990). The hierarchical grouping of components by the modules in an application seems to influence component quality: having more breadth and more depth in the hierarchical module structure is associated with fewer defects. (Given the high correlation between *NUM_NOM_MODULES* and *HIERARCHY_DEPTH*, we test the robustness of our results to the exclusion of *HIERARCHY_DEPTH* and confirm that they are not sensitive to that change.) Finally, components in applications whose modules deviate from the code structure recommended for handling interfaces across modules seem to be more defect prone (Martin 2002), and components in applications with higher propagation cost seem to have, on average, fewer defects (MacCormack et al. 2006).

Model 2 adds component-level controls. This model controls for the number of changes (e.g., incremental improvements and the addition of new features) associated with the focal component c. The positive and significant coefficients for $C\_EXPL\_CHANGES$ and $C\_IMPL\_CHANGES$ suggest that, for a given component, the number of bugs is positively correlated with the number of non–bug-fixing changes that affect it. In addition, we control for the amount of organizational attention and resources associated with the focal component, since its inception, in application i. The negative and significant coefficient for cumulative changes to the source code of component *c* prior to the current version *s*, $C\_CUM\_CHANGES$, suggests that components dealt with in previous versions of the application are *less* likely to be affected by bugs in the current version. However, the greater the number of authors and committers 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—in other words, the coefficients for $C\_CUM\_AUTHORS$ and $C\_CUM\_COMMITTERS$ are both positive and significant. Given the high correlation between these two variables and $C\_CUM\_CHANGES$, we test the robustness of our results to the exclusion of $C\_CUM\_CHANGES$ and find that our results are not sensitive to such exclusion. Finally, model 2 confirms that a component's cyclomatic complexity (*C_AVG_CC*) and source lines of code (*C_SLOC*) are important determinants of how many bugs it has (Card and Glass 1990, Henry and Selig 1990).

Model 3 tests for the effects of downstream component modularity (H1). Here we include two types of architectural variables: C_FAN_OUT (how much component *c* depends on other components) and C_FAN_IN (how much other components depend on *c*). The coefficient for *C_FAN_OUT* is positive and significant (consistent with H1) and significantly larger than *C_FAN_IN* ($p < .001$). (To test the difference of these two coefficients, we estimated an alternative model that includes *C_FAN_OUT* and a new variable defined as the sum of *C_FAN_OUT* and *C_FAN_IN*. In that model, the coefficient estimate of *C_FAN_OUT* tests the significance of the difference of the parameters of interest.) This model confirms, in line with the bulk of previous research, that the directionality of dependencies influences the relationship between a component's modularity and its number of defects (e.g., Card and Agresti 1988, Kan 1995, Briand et al. 1999, Aggarwal et al. 2007, Burrows et al. 2010). Components that depend on many other components are likely to be associated with higher number of bugs than are components that depend on fewer other components. Observe that, despite the positive and significant coefficient for *C_FAN_IN* in this partial model, the effect becomes insignificant when, in subsequent models, the effect of cyclicality is also included. Such instability of the effect of *C_FAN_IN* in our models is fully consistent with the inconclusive findings reported in the literature addressing the relationship between fan-in and quality (Briand et al. 1999). It also highlights the importance of accounting for the effect of component cyclicality when evaluating the effects of other component architectural features.

Model 4 tests hypothesis H2, which predicts a detrimental effect on quality for components with higher component cyclicality. This model includes the predictor variable $CYCLE\_SIZE$, the number of components in the cycle to which component *c* belongs. This model shows a positive and significant coefficient for $CYCLE\_SIZE$, indicating that the larger the cycle involving component *c*, the greater the expected number of bugs associated with that component. Note that model 4 is a random-coefficient model that includes an application-specific random slope associated with $CYCLE\_SIZE$, relaxing the assumption that the effect of $CYCLE\_SIZE$ is constant across all applications in the sample. This random-coefficient model fits our data better than does the nested random-intercept model (Singer and Willett 2003): the difference in the deviance statistic of the random intercept model (17209.29) and the random coefficient model (17129.44) is 79.85, which easily exceeds the .05 critical value of a χ^2 distribution with 2 *d.f.* (5.99). This suggests that the nested random-intercept model should be rejected in favor of the random-coefficient model, even though both models strongly support H2.

Model 5 tests H3, which predicts that, if we control for the size of the cycle, then a component‘s degree of in-cycle centrality is positively associated with its number of defects. The positive and significant coefficient for *IN_CYCLE_DEGREE* lends empirical support to this hypothesis and suggests that components in same-size cycles can differ in their number of defects: those components occupying more central (resp., more peripheral) positions in their cycle are likely to have a larger (resp., smaller) number of defects. Model 5 includes application-specific random slopes for both *CYCLE_SIZE* and *IN_CYCLE_DEGREE* because their inclusions make a significant deviance reduction when compared to the nested random-coefficient model that includes a random slope of *CYCLE_SIZE* only. (Deviance reduction = 8.55 > Critical χ^2 (0.05 right tailed, 3) = 7.81.)

Finally, model 6 tests H4. (Model 6 is a random coefficient model that includes application-specific random slopes for both *CYCLE_SIZE* and *IN_CYCLE_DEGREE* but not for *MODULES_CROSS CYCLE*. We do not include an application-specific random slope for *MODULES_CROSS_CYCLE* because doing so does not yield a significant reduction of deviance statistic with respect to the nested model that includes application-specific random slopes for both *CYCLE_SIZE* and *IN_CYCLE_DEGREE*. Deviance reduction = 8.91 < Critical χ^2 (0.05 right tailed, 4) = 9.49.) H4 predicts that components involved in cycles that cross a greater number of module boundaries are more likely to exhibit a greater number of defects. The model yields a positive and significant coefficient for $MODULES\_CROSS\_CYCLE$, which indicates that—beyond the effect of *CYCLE_SIZE* and *IN_CYCLE_DEGREE*—in-cycle components are likely (in line with H4) to have even more defects when they are not encapsulated by a single module. In our sample, 87% of the cycles cross at least one module boundary, which suggests that multi-module cycles are common. This suggests that, in an open source software development context, developers seem to neglect the negative consequences of dealing with cyclical dependencies across modules or are not aware of the existence of such cycles spanning multiple modules.

We tested the robustness of all findings reported in this section with respect to alternative model specifications. First, all the results are robust to the inclusion of quadratic terms for fan-out and fan-in, which were both negative and significant. This alternative specification suggests that the relationship between fan-out (and fan-in) and a component’s defects may be captured by an inverted “U” shape. However, the shapes of the quadratic functions are appreciably different for fan-out and fan-in. After including the effect of *CYCLE_SIZE* in our models, the quadratic function of fan-out does not peak within the range of fan-out values in our sample, suggesting a decreasing marginal return effect of fan-out (instead of an inverted U-shape form). For fan-in, the quadratic function is a "shallow" inverted U-shape that peaks about its mean. Second, we estimate hierarchical Poisson regression models that include—instead of a component-specific random effect nested in its corresponding application—a version-specific random effect nested in its corresponding application random effect. Our results are robust to this hierarchical model specification. Third, we estimate a zero-inflated Poisson (nonhierarchical) regression model with version-level fixed effects to ensure that our data do not contain too many zeros. The Vuong test (Vuong 1989), which compares a zero-inflated Poisson regression with a standard Poisson regression (featuring version-level fixed effects), does not significantly favor the zero-inflated model.

Although the analysis provides strong support for our hypotheses, a discussion of causality is in order. First, since our dependent variable is measured within a time span that does not commence until after all the independent variables have been measured (i.e., bugs are not discovered until after a version of the product has been released), it is unlikely that the existence of unidentified defects leads to the establishment of specific dependency patterns, such as cyclical dependencies, among product components. This reduces the risk of reverse causality. Second, the lag between the independent variables and our dependent variable mitigates the risk of unobserved factors (e.g., contemporaneous measurement errors) affecting both the dependent variable and our predictor variables in a similar manner. Of course, these considerations are not sufficient to guarantee causality in the strictest sense.

4.2 Effect Size of Cyclicality

In this section we not only estimate the magnitude of the effect of component cyclicality but also compare it with the effect size of component modularity (measured by the lack of component fan-out). In order to estimate an overall effect of component cyclicality, we estimate a random coefficient regression model that includes *IN_CYCLE* as the only cyclicality variable of interest. Such a model (Model 7 in Table 3) fits our data better than does the corresponding nested random-intercept model. (Deviance reduction = 31.76 > Critical χ^2 (0.05 right tailed, 2) = 5.99.) This model yields a positive and significant coefficient of *IN_CYCLE* (0.459, $p < .001$). According to that model, an in-cycle component has, on average, 58.3% ($e^{0.459} - 1$) more defects than a non-cycle component. In comparison, such a model also shows a positive and significant coefficient of *C_FAN_OUT* (0.014, $p < .001$). Hence, an increase of a single standard deviation in a component's fan-out is correlated with 35.3% more defects ($e^{(0.014)(21.6)} - 1$). Thus our regression results suggest that, on average, the overall effect of a component being in a cycle is of the same order of magnitude as the effect of component fan-out.

In order to test the robustness of our effect size estimates against potential confounding effects due to the high correlations between *IN_CYCLE* and *C_FAN_OUT* (and *C_FAN_IN*), we reestimate the effect size of *IN_CYCLE* using a matching approach. Toward this end, we implement a propensity score matching approach commonly used in medical trials and economics when seeking to evaluate a treatment effect in nonrandomized observational studies (Rosenbaum and Rubin 1983). The rationale behind this approach is to define a *propensity score* as the conditional probability of a component being in a cycle (i.e., of being a "treated" component) in terms of the component's other characteristics (e.g., its fan-out and fan-in). One must then identify components that have both a similar propensity score and a similar, "balanced" set of covariates—in this case, fan-in and fan-out—for the same range of propensity scores. Matched groups of components with similar propensity scores and a balanced set of covariates are used to estimate the *average effect of treatment on the treated* (ATT) as the difference between the expected number of bugs for the treated units (the in-cycle components) versus the untreated units (the non-cycle components) of the matched sample. We are ultimately interested in whether such a difference (percentage wise) in the number of bugs is the same as (or greater than) the difference obtained from our regression results.

For estimation purposes we use the *pscore* and *atts* methods implemented in Stata by Becker and Ichino (2002). The *pscore* method determines the propensity score by estimating a logistic regression that includes component-specific attributes likely to be associated with the inclusion of a component in a cycle. These component-level covariates include age, average cyclomatic complexity, and number of source code lines as well as fan-in and fan-out. We also include version-level fixed effects. As expected, the most salient predictors of being in a cycle are the fan-in and fan-out variables (their coefficients have *z*-scores that exceed 60).

Because it is virtually impossible to find two units in a sample that have the exact same propensity score, one must also devise an algorithm to identify matching groups that have both similar propensity scores and balanced covariates. For this, we use the stratification method executed by *atts* in Stata because, by definition, it guarantees a balanced set of matched samples if the outcome of the *pscore* is also balanced (Becker and Ichino 2002). In a sample of 6,064 components that is matched and balanced with respect to fan-in and fan-out, we find an ATT of 0.141 ($p < .001$)—in other words, the average in-cycle component has 0.141 more bugs than the average non-cycle component in our matched sample. More importantly, in that sample the in-cycle components have, on average, 79.7% more bugs than the non-cycle components (0.318 versus 0.177). (Unfortunately, a propensity score matching approach is not suitable to test for the effect of a continuous variable such as fan-out.) Overall, the results from using this matching approach to estimate the effect size of cyclicality indicate that the effect of cyclicality is (i) comparable to the one estimated from our regression results and (ii) not confounded with the effect of either fan-out or fan-in.

5 Discussion

Product quality matters. The competitiveness of most companies depends on it. Both the popular press and academic research have documented the negative consequences of poor quality. For example, the decline of market share among the Big Three US automakers in recent decades has been attributed to mediocre product quality (Klier 2009); Firestone even faced demise when a product design fault led to several fatal accidents (Pinedo et al. 2000). A wealth of studies in different contexts has documented the consequences of poor quality on firm survivability (Li and Hamblin 2003), market share (Mohrman et al. 1995), and profits (Fuentes-Fuentes et al. 2004). Conversely, the long-term survival and widespread adoption of systems based on open source software has been (at least partially) attributed to code quality (Ajila and Wu 2007). Many drivers of product quality have been recognized, and strategies for improving it have received widespread attention (Cua et al. 2001). Hence, improving our understanding of the factors that drive product is of paramount importance.

The literature has already identified product architecture as a major factor (Ulrich 1995, Ulrich and Eppinger 2008). However, previous research has focused on modularity as the most salient architectural characteristic in the architecture–quality relationship (e.g., Briand et al. 1999, Aggarwal et al. 2007, Burrows et al. 2010). Our study demonstrates that a second architectural feature, cyclicality, is similarly important. We empirically link cyclicality to quality, and we identify particular aspects of cyclicality that significantly affect quality. First, we find that component cyclicality is a significant predictor of component defectiveness whose effect is of the same order of magnitude as is modularity. Second, in untangling the cyclicality construct, we learn that a component's centrality in a cycle plays a significant role: components that occupy a more central position in a cycle are more prone to defects than are components that occupy peripheral positions. Finally, we show that architecture is determined not only by the dependency structure but also by the hierarchical grouping of components into modules: the defect proneness of product components increases with the number of module boundaries crossed by their cyclical dependencies.

Establishing an empirical link between component cyclicality and the level of defects of product components highlights the importance of studying the relationship between architectural properties of product components and other dimensions of performance. Given the iterative nature and higher coordination needs associated with in-cycle components, we would expect these components not only to be more defective but also to be at higher risk of missing schedule and budget targets, and thus to negatively impact multiple dimensions of product development. Yet, empirical evidence for this assertion is currently lacking. In addition, looking at the dynamic evolution of products, one could argue that cycles play a significant role in how products evolve (MacCormack et al. 2006, Sosa et al. 2007a, MacCormack et al. 2008): considering again the iterative approaches and coordination needs associated with in-cycle components, one could expect these components to exhibit different rates of redesign, upgrade, and removal than non-cycle components.

Our work also has implications for understanding product architecture on a conceptual level. Our results show that the interplay of the module and dependency structures relates to product quality. This interplay (and its consequences for quality) is intriguing, because the forces that shape a product's module structure are substantially different from those that shape its dependency structure. According to the classical trope of the architecture literature, establishing a hierarchical module structure entails breaking the product into several major building blocks or subsystems and then mapping the product's functionality to each (Ulrich 1995). This process is repeated, top-down, in a nested manner, for each subsystem until all functions of a system have been assigned to components (Ulrich and Eppinger 2008), resulting in the system's *intended* architecture (i.e., the modules that system architects and managers deliberately set up as the system's building blocks). In contrast, the product's dependency structure, which determines the existence of cycles in the product, is the result of myriad local decisions that are typically made by technical personnel who optimize performance in terms of the local criteria associated with their components. Such decisions are made in a bottom-up manner and thus, from the viewpoint of management, simply "emerge," resulting in the system's *actual* architecture. Our results highlight the importance of aligning both aspects of architecture by showing how misalignment between module structure and dependency structure (e.g., modules failing to fully encapsulate cycles) has a negative effect on quality. This means that system architects should—as early in the design process as possible—look beyond the hierarchy of a system or product's modules to examine its actual dependency structure where cycles reside. Defect proneness can be mitigated by properly aligning the product's hierarchical modules with its dependency structure.

Given our results, the information systems literature should explicitly take into account the role of architectural cyclicality when studying the factors that drive software performance. For instance, an important software architecture decision in the information systems literature is the refactoring of computer code. Software code quality tends to "decay" over time because additions and changes to the code often fail to follow the prescribed design rules, such as where to add allowable dependencies. Refactoring improves the internal structure of the code by "redistribute[ing] classes, variables and methods across the class hierarchy in order to facilitate future adaptations and extensions" (Mens and Tourwé 2004, p. 1) and is vital for the long-term survival of a software application. However, source code decay may be driven by the divergence of the actual and intended architectures (even if they were originally aligned) as dependency decisions continue to be made in a distributed way by a large number of decision makers (whereas architecture decisions are seldom revisited, and even then by only a central architect or small group of architects). This divergence may well call for realignment, which is the purpose of refactoring. Several methods have been proposed to help identify code segments in need of refactoring (e.g., Kataoka et al. 2001, Simon et al. 2001); yet these methods have overlooked the need to monitor the existence and characteristics of cycles as important determinants in the refactoring. The analysis presented in this paper suggests that the dependency structure—and especially the presence of cycles—is an important additional clue in the search for code elements to be refactored.

Our findings suggest two architectural action fields that managers should consider to improve product quality in a typical development process.

Visualize the architecture. It is crucial for managers to understand the key components of product architectures: dependencies among components (especially when they form cycles) and nested modules that group components. Visualizing the architecture is fundamental for improving one's understanding of both its technical and organizational aspects, since product architecture decisions influence formal and informal organizational structures (Sosa et al. 2004, MacCormack et al. 2006, Eppinger and Browning 2012).

Identify and manage component cycles. Managers should routinely identify component cycles (stemming from the product's dependency structure) as well as the actors responsible for their design, since the cycles and actors both will require disproportionate attention. In order to identify component cycles, managers must actually disregard the constraints imposed by the particular arrangement of components into nested modules. Our empirical results suggest several steps that can be taken to mitigate the negative effects of identified cycles. First, try to break the cycle by rerouting the critical dependencies that form them, especially where they stem from components central to the cycle. If a cycle cannot be broken then its size should be reduced, since larger cycles are more detrimental in placing a larger fraction of components at risk of defects. Third, reduce cycle complexity, especially for components that are central to the cycle. Reducing their centrality in the cycle can improve code quality. Fourth, ensure that modules encapsulate cycles. Finally, although managers must identify and monitor component cycles in the short run, they should preempt cycles in the long run by establishing and enforcing design rules (Baldwin and Clark 2000) that specify types of allowable component relationships (Sangal et al. 2005).

Our study has some limitations. In particular, the analysis was performed on a sample of Java-based applications developed by the open source Apache Software Foundation. To be able to understand the limits of any attempt to generalize our findings, two important attributes of our empirical context merit discussion. First, because the organizational structures in open-source development settings are typically geographically distributed, the product architectures that emerge from such settings are likely to be less interconnected than the architectures of products developed in closed-source development settings (MacCormack et al. 2011). We could therefore expect the occurrence of cycles to be less salient in open-source than in closed-source projects. Yet, without further studies on closed settings, it is unclear whether the effect of cyclicality would be any different in equivalent open and closed settings. Second, although open-source development does not necessarily rely on formal organizational structures defined by any particular firm, nor on face-to-face communications for informal inter-personal coordination, coordination efforts between interdependent actors take place through different mechanisms such as committers who act as project leaders and in on-line discussion lists which enable direct communication among authors (Mockus et al. 2000). Hence, studies in other settings are needed before our findings can be fully generalized.

We have been able to apply methods—developed in the context of exploring the task structure of development processes—to the wealth of data available in the open source space, thereby linking the concept of cyclicality to quality outcomes and explicating cyclicality's different facets. Our findings raise important questions. This paper has focused on the consequences of a product's cyclical dependencies, but what are their antecedents? Where in the product are cycles likely to form? Under what circumstances do they arise? How do they grow or shrink? Can we define an architecture that is optimal in terms of minimizing defects? (For a first attempt in this direction, see (Sosa et al. 2011).) How do architectural and organizational patterns interact and co-evolve over time? (See Colfer and Baldwin 2010, MacCormack et al. 2011.) How would such co-evolution influence defect proneness and other performance metrics? Further exploration of the consequences of cycles might ask how the presence of cycles affects the time required to fix defects. Addressing these questions poses interesting challenges for future research. This paper provides an important step toward the development of an empirically validated theory of product architecture design.

References

Aggarwal, K. K., Y. Singh, A. Kaur, R. Malhotra. 2007. Investigating Effect of Design Metrics on Fault Proneness in Object-Oriented Systems. *Journal of Object Technology*. **6**(10) 127-141.

Ajila, S. A., D. Wu. 2007. Empirical Study of the Effects of Open Source Adoption on Software Development Economics. *Journal of Systems and Software*. **80**(9) 1517-1529.

Allen, T. J. 1977. *Managing the Flow of Technology*. MIT Press, Cambridge, MA.

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

Becker, S. O., A. Ichino. 2002. Estimation of Average Treatment Effects Based on Propensity Scores. *The Stata Journal*. **2**(4) 358-377.

Braha, D., Y. Bar-Yam. 2007. The Statistical Mechanics of Complex Product Development: Empirical and Analytical Results. *Management Sci.* **53**(7) 1127-1145.

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.

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.

Burrows, R., F. C. Ferrari, O. A. L. Lemos, A. Garcia, F. Taïani. 2010. The Impact of Coupling on the Fault-Proneness of Aspect-Oriented Programs: An Empirical Study. *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering (ISSRE2010)*, San Jose, CA, Nov 1-4.

Cabigiosu, A., A. Camuffo. 2011. Beyond the 'Mirroring' Hypothesis: Product Modularity and Interorganizational Relations in the Air Conditioning Industry. *Organization Sci.*(forthcoming).

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

Card, D. N., W. W. Agresti. 1988. Measuring Software Design Complexity. *Journal of Systems and Software*. **8** 185-197.

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.

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

Colfer, L., C. Y. Baldwin. 2010. The Mirroring Hypothesis: Theory, Evidence and Exceptions. Harvard Business School, Working Paper 10-058.

Cua, K. O., K. E. McKone, R. G. Schroeder. 2001. Relationships between Implementation of TQM, JIT, and TPM and Manufacturing Performance. *J. of Operations Mgmt.* **19** 675-694.

Eppinger, S.D., T.R. Browning. 2012. *Design Structure Matrix Methods and Applications*. MIT Press, Cambridge, MA.

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.

Freeman, L. 1979. Centrality in social networks. Conceptual clarification. *Social Networks* **1**, pp. 215-239.

Fuentes-Fuentes, M. M., C. A. Albacete-Sáez, F. J. Lloréns-Montes. 2004. The Impact of Environmental Characteristics on TQM Principles and Organizational Performance. *Omega*. **32**(6) 425-442.

Giffin, M., O. d. Weck, G. Bounova, R. Keller, C. Eckert, P. J. Clarkson. 2009. Change Propagation Analysis in Complex Technical Systems. *J. of Mech. Design*. **131**(8) 081001.

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.

Gould, R. J. 1988. *Graph Theory*. Benjamin-Cummings, Menlo Park, CA.

Harary, F. 1969. *Graph Theory*. Addison-Wesley, Reading, MA.

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., 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.

Kataoka, Y., M. D. Ernst, W. G. Griswold, D. Notkin. 2001. Automated Support for Program Refactoring using Invariants. *Proc. of the International Conference on Software Maintenance*, Florence, Italy, Nov 6-10, 736-743.

Klier, T. 2009. From Tail Fins to Hybrids: How Detroit Lost its Dominance of the U.S. Auto Market. *Economic Perspectives*. **33**(2).

Kreft, I., J. d. Leeuw, L. Aiken. 1995. The Effect of Different Forms of Centering in Hierarchical Linear Models. *Multivariate Behavioral Research*. **30**(1) 1-21.

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

Li, X., D. J. Hamblin. 2003. The Impact of Performance and Practice Factors on UK Manufacturing Companies' Survival. *Int. J. of Production Res.* **41**(5).

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

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. 2011. Exploring the Duality between Product and Organizational Architectures: A Test of the Mirroring Hypothesis. Harvard Business School, Working Paper 08-039.

Martin, R. C. 2002. *Agile Software Development*. Prentice Hall, Englewood Cliffs, NJ.

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

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

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

Miller, G. A. 1956. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *Psychological Review*. **63**(2) 81-97.

Mockus, A., R. T. Fielding, J. Herbsleb. 2000. A Case Study of Open Source Software Development: The Apache Server. *Proc. of the 22nd Int. Conf. on Software Engineering (ICSE)*, Limerick, Ireland, June 4-11, 263-272.

Mohrman, S. A., R. V. Tenkasi, E. E. L. III, G. G. L. Jr. 1995. Total Quality Management: Practice and Outcomes in the Largest US Firms. *Employee Relations*. **17**(3) 26-41.

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

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

Pinedo, M., S. Sehasdri, E. Zemel. 2000. The Ford-Firestone Case. Stern School of Business, NYU, Teaching Case.

Rabe-Hesketh, S., A. Skrondal. 2008. *Multilevel and Longitudinal Modeling Using Stata*. 2nd Edition, Stata Press.

Ramachandran, K., V. Krishnan. 2008. Design Architecture and Introduction Timing for Rapidly Improving Industrial Products. *Manufacturing & Service Operations Management*. **10**(1) 149-171.

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. *Mgmt. Sci.* **52**(7) 984-999.

Rosenbaum, P., D. Rubin. 1983. The Central Role of the Propensity Score in Observational Studies for Causal Effects. *Biometrika*. **70**(1) 41-55.

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, F., F. Steinbrückner, C. Lewerentz. 2001. Metrics Based Refactoring. *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*, Lisbon, Portugal, Mar 14-16, 30-38.

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

Singer, J. D., J. B. Willett. 2003. *Applied Longitudinal Data Analysis*. Oxford University Press, New York, NY.

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

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

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 & Computers and Information in Engineering Conference (IDETC/CIE 2007),* Las Vegas, NV, Sep. 4-7.

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.

Sosa, M. E., J. Mihm, T. R. Browning. 2011. Degree Distribution and Quality in Complex Engineered Systems. *J. of Mech. Design*. **133**(10) 101008.

Staats, B. 2012. Unpacking Team Familiarity: The Effects of Geographic Location and Hierarchical Role. *Production and Ops. Mgmt.* **21**(3) 619-635.

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.

Tarjan, R. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*. **1**(2) 146-160.

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.

Ulrich, K. T., S. D. Eppinger. 2008. *Product Design and Development*. 4th Edition, McGraw-Hill, New York.

Vuong, Q.H. 1989. Likelihood Ratio Tests for Model Selection and Non-nested Hypotheses. *Econometrica*. **57** 307-333.

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

Wasserman, S., K. Faust. 1994. *Social Network Analysis*. Cambridge University Press, Cambridge, UK.

Table 2. Descriptive Statistics and Correlations of Variables (*N* = 28,394)

Variables	**Mean / STD**	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
1 Number of bugs, y_{cis}	0.2 / 0.6	1.00																									
2 $UNFIXED\_BUGS_{is}$	45.6 / 55.2	.18	1.00																								
3 AGE_{is}	786.5 / 652.6	.06	.23	1.00																							
4 $DAYS\_BEFORE_{is}$	230.9 / 259.6	.07	.27	.46	1.00																						
5 $DAYS\_AFTER_{is}$	299 / 324.6	.18	.44	.43	.32	1.00																					
6 $NEWNESS_{is}$	31.6 / 43.4	.17	.40	.14	.32	.16	1.00																				
7 APP_SLOC_{is}	22.3 / 20.6	-.06	-.02	.09	.11	.11	-.12	1.00																			
8 $APP\_AVG\_CC_{is}$	2.2 / 0.9	.03	.02	.38	.09	.05	.01	-.14	1.00																		
9 $NUM\_NOM\_MODULES_{is}$	43 / 42.6	-.07	.17	.05	-.12	.00	-.06	-.24	-.33	1.00																	
10 $HIERARCHY\_DEPTH_{is}$	4.3 / 1.1	-.07	.09	.03	-.02	.09	-.10	-.06	-.41	.81	1.00																
11 $AVG\_INTERFACE\_USAGE_{is}$	0.2 / 0.0	-.10	.08	.15	.24	.17	-.19	.31	-.07	.26	.33	1.00															
12 $PROPAGATION\_COST_{is}$	15.4 / 8.7	.00	-.03	-.07	-.09	-.07	.19	-.06	-.01	-.28	-.45	-.22	1.00														
13 C_AGE_{cis}	400.3 / 494.1	.09	.11	.64	.53	.35	.05	.18	.25	-.13	-.09	.15	.04	1.00													
14 $C\_EXPL\_CHANGES_{cis}$	0.6 / 3.2	.35	.17	.01	.07	.19	.31	-.05	.05	-.11	-.13	-.09	-.01	.04	1.00												
15 $C\_IMPL\_CHANGES_{cis}$	0.2 / 0.6	.26	.02	.01	.02	-.07	.05	-.13	-.01	.03	-.02	-.09	.06	.07	.06	1.00											
16 $C\_CUM\_CHANGES_{cis}$	2.9 / 9.7	.45	.26	.09	.23	.14	.44	-.06	.00	-.05	-.06	-.12	.02	.24	.40	.31	1.00										
17 $C\_CUM\_COMMITTERS_{cis}$	1.0 / 2.0	.36	.33	.16	.38	.19	.50	-.12	.01	.02	-.01	-.03	-.05	.31	.37	.30	.81	1.00									
18 $C\_CUM\text{-}AUTHORS_{cis}$	0.9 / 2.2	.41	.07	-.01	.01	-.07	.20	-.14	-.05	.01	-.01	-.26	.00	.11	.23	.45	.70	.57	1.00								
19 $C\_INTERFACE\_USAGE_{cis}$	0.3 / 0.2	-.02	.21	.04	.04	.10	.06	-.10	-.07	.27	.16	.25	.17	.04	.00	.00	-.04	.02	-.12	1.00							
20 $C\_AVG\_CC_{cis}$	2.1 / 2.0	.10	.06	.05	.00	.02	.03	.00	.02	.03	-.02	-.01	.07	.03	.04	.11	.08	.10	.11	.07	1.00						
21 C_SLOC_{cis}	0.1 / 0.1	.23	.01	.03	.00	.01	-.02	.06	-.01	.02	.01	.04	.03	.06	.06	.18	.17	.14	.22	.02	.39	1.00					
22 $C\_FAN\_OUT_{cis}$	15.1 / 21.6	.11	-.01	-.02	-.03	-.03	.08	-.04	.03	-.11	-.19	-.09	.40	.04	.03	.13	.10	.07	.15	.08	.24	.19	1.00				
23 $C\_FAN\_IN_{cis}$	15.9 / 19.4	.04	-.03	-.05	-.05	-.04	.07	-.01	-.04	-.13	-.19	-.09	.46	.05	.02	.08	.05	.00	.04	.09	-.04	.05	.10	1.00			
24 $CYCLE\_SIZE_{cis}$	12.6 / 30.9	.06	.13	.00	-.02	.04	.02	.06	-.12	.09	.05	.06	.23	.05	-.02	.08	.04	.03	.05	.17	.11	.17	.47	.43	1.00		
25 $IN\_CYCLE\_DEGREE_{cis}$	1.4 / 4.5	.19	.10	.02	-.01	.02	.03	.02	-.01	.02	-.02	.01	.13	.06	.04	.20	.16	.12	.19	.08	.16	.39	.35	.31	.57	1.00	
26 $MODULES\_CROSS\_CYCLE_{cis}$	1.3 / 3.9	.08	.12	.02	-.03	.01	.05	-.09	-.09	.25	.15	.04	.12	.00	-.01	.16	.10	.13	.12	.15	.13	.15	.32	.34	.72	.48	1.00
27 IN_CYCLE_{cis}	0.2 / 0.4	.14	.08	.01	-.01	.02	.02	.02	-.05	.01	-.04	.01	.19	.06	.04	.12	.11	.06	.14	.08	.15	.25	.53	.41	.77	.59	.62

Note: Correlations > |0.02| are significant at $p < .01$.

Table 3. Hierarchical Poisson Regressions Predicting the Number of Bugs per Component (*N* = 28,394)

	Model 1	**Model 2**	**Model 3**	**Model 4**	**Model 5**	**Model 6**	**Model 7**
$UNFIXED\_BUGS_{is}$	0.004*** (0.001)	0.002*** (0.001)	0.002*** (0.001)	0.002*** (0.001)	0.003*** (0.001)	0.002*** (0.001)	0.002*** (0.001)
AGE_{is}	-0.002*** (0.000)	-0.003*** (0.000)	-0.002*** (0.000)	-0.003*** (0.000)	-0.003*** (0.000)	-0.002*** (0.000)	-0.002*** (0.000)
$DAYS\_BEFORE_{is}$	-0.001*** (0.000)	-0.001*** (0.000)	-0.001*** (0.000)	-0.001*** (0.000)	-0.001*** (0.000)	-0.001*** (0.000)	-0.001*** (0.000)
$DAYS\_AFTER_{is}$	0.002*** (0.000)	0.002*** (0.000)	0.002*** (0.000)	0.002*** (0.000)	0.002*** (0.000)	0.002*** (0.000)	0.002*** (0.000)
$NEWNESS_{is}$	0.001 (0.001)	0.001 (0.001)	0.001 (0.001)	0.001 (0.001)	0.001 (0.001)	0.001 (0.001)	0.001 (0.001)
APP_SLOC_{is}	0.011*** (0.004)	0.012*** (0.004)	0.011*** (0.004)	0.015*** (0.004)	0.015*** (0.004)	0.015*** (0.004)	0.011*** (0.004)
$APP\_AVG\_CC_{is}$	0.151*** (0.014)	0.137*** (0.013)	0.137*** (0.013)	0.125*** (0.013)	0.127*** (0.013)	0.128*** (0.013)	0.135*** (0.013)
$NUM\_NOM\_MODULES_{is}$	-0.022*** (0.003)	-0.023*** (0.003)	-0.022*** (0.003)	-0.025*** (0.003)	-0.025*** (0.003)	-0.025*** (0.003)	-0.023*** (0.003)
$HIERARCHY\_DEPTH_{is}$	-0.210** (0.105)	-0.237** (0.108)	-0.225** (0.107)	-0.261** (0.111)	-0.249** (0.110)	-0.256** (0.110)	-0.219** (0.108)
$AVG\_INTERFACE\_USAGE_{is}$	3.673** (1.53)	4.234*** (1.575)	4.044** (1.568)	5.176*** (1.655)	4.376*** (1.665)	4.885*** (1.704)	4.229*** (1.581)
$PROPAGATION\_COST_{is}$	-0.062*** (0.008)	-0.064*** (0.008)	-0.087*** (0.008)	-0.115*** (0.010)	-0.110*** (0.010)	-0.108*** (0.01)	-0.081*** (0.008)
C_AGE_{cis}		0.000 (0.000)	0.000 (0.000)	0.000* (0.000)	0.000* (0.000)	0.000* (0.000)	0.000 (0.000)
$C\_EXPL\_CHANGES_{cis}$		0.030*** (0.003)	0.027*** (0.003)	0.027*** (0.003)	0.027*** (0.003)	0.028*** (0.003)	0.029*** (0.003)
$C\_IMPL\_CHANGES_{cis}$		0.070*** (0.013)	0.063*** (0.013)	0.047*** (0.013)	0.045*** (0.013)	0.045*** (0.013)	0.062*** (0.013)
$C\_CUM\_CHANGES_{cis}$		-0.015*** (0.002)	-0.013*** (0.002)	-0.012*** (0.002)	-0.012*** (0.002)	-0.012*** (0.002)	-0.012*** (0.002)
$C\_CUM\text{-}COMMITTERS_{cis}$		0.162*** (0.013)	0.140*** (0.013)	0.148*** (0.013)	0.151*** (0.013)	0.150*** (0.013)	0.142*** (0.013)
$C\_CUM\text{-}AUTHORS_{cis}$		0.054*** (0.009)	0.052*** (0.009)	0.047*** (0.009)	0.045*** (0.009)	0.046*** (0.009)	0.047*** (0.009)
$C\_INTERFACE\_USAGE_{cis}$		0.021 (0.101)	0.020 (0.099)	0.013 (0.100)	0.026 (0.100)	0.000 (0.101)	-0.022 (0.099)
$C\_AVG\_CC_{cis}$		0.095*** (0.009)	0.073*** (0.009)	0.072*** (0.009)	0.073*** (0.009)	0.074*** (0.009)	0.072*** (0.009)
C_SLOC_{cis}		0.002*** (0.000)	0.002*** (0.000)	0.002*** (0.000)	0.002*** (0.000)	0.002*** (0.000)	0.002*** (0.000)
$\boldsymbol{C\_FAN\_OUT_{cis}}$			0.019*** (0.001)	0.015*** (0.001)	0.015*** (0.001)	0.015*** (0.001)	0.014*** (0.001)
$\boldsymbol{C\_FAN\_IN_{cis}}$			0.004*** (0.001)	0.000 (0.002)	0.000 (0.002)	0.000 (0.002)	0.001 (0.001)
$\boldsymbol{CYCLE\_SIZE_{cis}}$				0.013*** (0.005)	0.012*** (0.004)	0.009** (0.004)	
$\boldsymbol{IN\_CYCLE\_DEGREE_{cis}}$					0.016** (0.008)	0.017** (0.008)	
$\boldsymbol{MODULES\_CROSS\_CYCLE_{cis}}$						0.027** (0.011)	
$\boldsymbol{IN\_CYCLE_{cis}}$							0.459*** (0.140)
Log-likelihood (*LL*)	-9329.459	-8745.888	-8623.929	-8564.721	-8557.613	-8554.501	-8590.827
AIC	18702.92	17553.78	17313.86	17201.44	17195.23	17191.00	17253.65

Notes: All models include component- and application-specific nested random effects as well as year fixed effects. In addition, models 4–7 include application-specific random coefficients of the cyclicality variables.
Standard errors are given in parentheses. $^{*}p < .1$, $^{**}p < .05$, $^{***}p < .01$

Linking Cyclicality and Product Quality

(*On-line electronic companion*)

Appendix A: Full Description of Control Variables

Appendix B: Standard deviation of random-effects parameters and goodness-of-fit statistics for model selection for all regression models

Appendix A: Full Description of Control Variables

System-level factors (version *s* of application *i*)

Unpatched bugs $UNPATCHED\_BUGS_{is}$	Our dependent variable counts all bugs fixed or being fixed ("patched" bugs), but not unpatched bugs since they are not yet associated with specific components. We therefore control for the number of unpatched bugs in the version to which a component belongs.
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 version available (as indicated in the release notes) and then ages with successive versions.
Days since last release $DAYS\_BEFORE_{is}$	The time between the current and previous version. The longer this period, the higher the probability that changes have been introduced that could generate bugs.
Days to next release $DAYS\_AFTER_{is}$	The time between the release of the current and the next version. The longer this period, the higher the probability that bugs will be discovered, because during that period the application is most actively scrutinized by testers and users. For applications in their last version, we set the next date as the last day of our data collection (or the last date the application was officially available) because bugs could still be identified until such a date.
Application newness $NEWNESS_{is}$	Both *new features* (added functionality) and *improvements* (modifications to existing functionality) are likely to introduce unforeseen perturbations and thus bugs. We count these items in the release notes (as determined by the project's "committers," who are responsible for authorizing the release).
Source lines of code APP_SLOC_{is}	Source Lines of Code is one of the most widely used metrics to capture the raw complexity of a software application (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 a readily available tool called JHawk (www.virtualmachinery.com), which directly counts the number of statements (excluding comments) in the source code of each method[1] in each component.
Average cyclomatic complexity $APP\_AVG\_CC_{is}$	Measures the complexity of a version based on the internal complexity of its components, calculated as the average cyclomatic complexity of all methods in the version. Cyclomatic complexity is the minimum number of linearly independent paths in the control flow graph of a method in a software program (McCabe 1976). This is a method-level variable, not an architectural measure. It is used to identify the methods of a program that would be harder to test and maintain (McCabe 1976, Henry and Selig 1990). We use the JHawk tool to calculate cyclomatic complexity by examining the source codes of all the methods in all components in our sample.
Number of nominal modules $NUM\_NOM\_$ $MODULES_{is}$	Modular groupings are likely to affect the cognitive ability of the development team to understand the architecture of the source code, and may thus influence their propensity to generate bugs. This measure counts the number of modules that contain components. It represents the breadth of the hierarchical structure.

[1] A *method* is a self-contained collection of programming instructions that typically includes variable instantiation and control flow statements, such as "if … then" and "while … do" statements. The average Java class (our unit of analysis) in our sample contains 10 methods.

Depth of the hierarchy $HIERARCHY\_DEPTH_{is}$	This construct measures the second dimension of the hierarchical structure. It represents the maximum of the count of the number of nested modules between the application root and any component in the version of the application—i.e., the maximum depth of the hierarchy.
Avg interface classes usage $AVG\_INTERFACE\_USAGE_{is}$	In object-oriented programming, as enabled by Java, the use of "interface-type" components[2] is customary to decouple modules. Because our sample includes Java-based applications, we aim to control for the ability of developers to properly use "interface-type" classes to handle dependencies across modules. To do so, we use the normalized metric of "distance" proposed by Martin (2002, p. 267), which assesses developers' ineffectiveness at grouping interface-type components into potentially stable modules. We use the LDM tool (developed by Lattix) to calculate this metric directly from the JAR files. This is a module-level variable, which is averaged across all modules in the application to derive a version-level measure.
Propagation cost $PROPAGATION\_COST_{is}$	The presence *or* absence of direct and indirect dependencies between the components of a complex system can create defects (Clarkson et al. 2004, MacCormack et al. 2006, Sosa et al. 2007). We control for the overall connectedness of the components in a product release by calculating its propagation cost as defined by (MacCormack et al. 2006).

Component-level factors (component *c* of application *i* in version *s*)

Component Age C_AGE_{cis}	The age of the component is measured by the number of days since the component was released in any of the versions of the application. This assumes that the component is officially "born" on the release date of first version that the component was part of (as indicated in the release notes) and then ages with successive releases.
Component-explicit non-bug changes $C\_EXPL\_CHANGES_{cis}$	The number of improvements, new features, and other issues explicitly associated with a component as indicated in the bug tracking systems; controls for sources of potential perturbations in the component and thus for potential bugs.
Component-implicit total changes $C\_IMPL\_CHANGES_{cis}$	The number of bugs, improvements, new features, and other issues implicitly associated with a component. Although bug-tracking systems do not explicitly associate these types of changes in the source code with a version, their entry date in the bug-tracking system occurs between the release dates of versions s and $s - 1$.

[2] Interface-type classes are "abstract" components, similar to "standardized interfaces" in hardware systems (Ulrich 1995), used to decouple two or more modules. As indicated by Martin (2002, p. 268), the "distance" 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. Such a recommended balance suggests that the fraction of interface-type components in a module should be proportional to the ratio of components outside this module that depend on this module over the number of components in the module that depends on other modules (Martin 2002, p. 264). In other words, because interface-type components are likely to handle dependencies across modules, they should be grouped into more stable modules.

Cumulative number of changes $C\_CUM\_CHANGES_{cis}$	The cumulative number of changes associated with a component *prior to* version s; controls for the cumulative workload generated by a component before current version s. One could argue that components that have generated significantly more tasks in previous versions are inherently more likely to attract bugs, yet one could also argue that the more bug-fixing workload a component 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 control for this variable.
Cumulative number of committers $C\_CUM\_$ $COMMITTERS_{cis}$	The cumulative number of committers associated with a component prior to a version. Committers have "write" access to the Apache code base. In addition to adding and modifying code themselves, they review and approve code submitted by other programmers, so as a result they are likely to influence the quality of a component.
Cumulative number of authors $C\_CUM\_$ $ATUHTORS_{cis}$	The cumulative number of authors associated with a component prior to a version. Authors contribute a code segment to the project, which is then reviewed and approved by the committers. Hence, while committers act as release managers who ultimately rule on (approve or reject) changes to the source code; authors merely propose those changes—not only to fix bugs but also to make functional improvements.
Interface_usage $C\_INTERFACE\_$ $USAGE_{cis}$	A variable ranging from 0 to 1, with higher values indicating greater deviation from the recommended usage of interface-type components to handle dependencies across modules (see a description of this variable at the version level). While interface-usage is a module-level variable, because each component is uniquely assigned to a module, we assign the same score to all components in a module.
Average component cyclomatic complexity $C\_AVG\_CC_{cis}$	Average cyclomatic complexity of a component's methods; a control for the internal complexity of a component (see a description of this variable at the version level).
Source lines of code of the component C_SLOC_{cis}	We measure (in "kilolines") the number of statements (excluding comments) in each method of each component (see a description of this variable at the version level as well).

References

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

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

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

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.

Martin, R. C. 2002. *Agile Software Development*. Prentice Hall, Englewood Cliffs, NJ.

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

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

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

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

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

Appendix B: Table B. Standard deviation of random-effects parameters and goodness-of-fit statistics of all regression models

	Model 1	**Model 2**	**Model 3**	Model 4a	**Model 4**	Model 5a	Model 5b	**Model 5**	Model 6a	Model 6b	**Model 6**	Model 6c	Model 7a	**Model 7**
Component-level random effect														
Random intercept per component	**1.147*** (0.029)**	**0.791*** (0.027)**	**0.718*** (0.026)**	0.711*** (0.026)	**0.693*** (0.026)**	0.713*** (0.026)	0.693*** (0.026)	**0.686*** (0.027)**	0.714*** (0.026)	0.693*** (0.026)	**0.687*** (0.027)**	0.687*** (0.027)	0.698*** (0.026)	**0.686*** (0.027)**
Application-level random effects														
Random intercept per application	**2.595*** (0.521)**	**2.767*** (0.558)**	**2.720*** (0.548)**	2.763*** (0.556)	**2.913*** (0.005)**	2.752*** (0.554)	2.906*** (0.584)	**2.900*** (0.581)**	2.764*** (0.557)	2.890*** (0.581)	**2.881*** (0.579)**	2.941*** (0.588)	2.712*** (0.547)	**2.781*** (0.562)**
Random slope for *CYCLE_SIZE*					**0.015*** (0.005)**		0.015*** (0.005)	**0.013*** (0.005)**		0.015*** (0.005)	**0.013*** (0.005)**	0.013** (0.005)		
Random slope for *IN_CYCLE_DEGREE*								**0.022** (0.010)**			**0.023** (0.010)**	0.025** (0.010)		
Random slope for *MODULES_CROSS_CYCLE*												0.083*** (0.026)		
Random slope for *IN_CYCLE*														**0.390*** (0.132)**
Goodness-of-fit statistics and parameters for model selection														
Log-likelihood (*LL*)	**-9329.46**	**-8745.89**	**-8623.93**	-8604.64	**-8564.72**	-8600.51	-8561.89	**-8557.61**	-8593.67	-8558.82	**-8554.50**	-8550.04	-8606.71	**-8590.83**
Deviance *(-2LL)*	**18658.92**	**17491.78**	**17247.86**	17209.29	**17129.44**	17201.02	17123.78	**17115.23**	17187.35	17117.64	**17109.00**	17100.09	17213.41	**17181.65**
Degrees of freedom (*d.f.*)	**22**	**31**	**33**	34	**36**	35	37	**40**	36	38	**41**	45	34	**36**
Deviance reduction		**1167.14**	**243.92**		**79.85**		77.24	**8.55**		69.70	**8.64**	8.91		**31.76**
Δ (*d.f.*)		**9**	**2**		**2**		2	**3**		2	**3**	4		**2**
Critical χ^2 (0.05 right tailed, Δ *d.f.)*		**16.92**	**5.99**		**5.99**		5.99	**7.81**		5.99	**7.81**	9.49		**5.99**
Selected Model (reported in Table 3)	**Yes**	**Yes**	**Yes**		**Yes**			**Yes**			**Yes**			**Yes**

- The standard deviation of each random parameter is shown with its corresponding standard error in parentheses.
- Standard deviation of correlation parameters between application-level random parameters are also estimated but not shown for simplicity of exposition.
- Models 1-3 are random intercept models shown in Table 3.
- Models 4's include *CYCLE_SIZE* as main cyclicality predictor.
- Models 5's include both *CYCLE_SIZE* and *IN_CYCLE_DEGREE* as main cyclicality predictors.
- Models 6's include *CYCLE_SIZE*, *IN_CYCLE_DEGREE*, and *MODULES_CROSS_CYCLE* as main cyclicality predictors.
- Models 7's include *IN_CYCLE* as main cyclicality predictor.

Random-coefficient model selection (Singer and Willett 2003, Rabe-Hesketh and Skrondal 2008):

- **Model 4** (which includes an application-specific random slope for *CYCLE_SIZE*) is preferred over Model 4a, which is the corresponding nested random intercept model that does not allow *CYCLE_SIZE* to vary across applications. The deviance reduction of Model 4 over Model 4a is significant (79.85 > Critical χ^2 (0.05 right tailed, 2) = 5.99).
- **Model 5** (which includes application-specific random slopes for both *CYCLE_SIZE* and *IN_CYCLE_DEGREE*) is preferred over Models 5a and 5b, respectively. The deviance reduction of Model 5 over Model 5b is significant (8.55 > Critical χ^2 (0.05 right tailed, 3) = 7.81).
- **Model 6** (which includes application-specific random slopes for both *CYCLE_SIZE* and *IN_CYCLE_DEGREE*) is preferred over Models 6a, 6b, and 6c. The deviance reduction of Model 6 over Model 6b is significant (8.64 > Critical χ^2 (0.05 right tailed, 3) = 7.81). However, the more complex Model 6c, which also includes an application-specific random slope for *MODULES_CROSS_CYCLE*, does not offer a significant deviance reduction and therefore is rejected in favor of Model 6 (8.91< Critical χ^2 (0.05 right tailed, 4) = 9.49).
- **Model 7** (which includes an application-specific random slope for *IN_CYCLE*) is preferred over Model 7a, which is the corresponding nested random intercept model. The deviance reduction of Model 7 over Model 7a is significant (31.76 > Critical χ^2 (0.05 right tailed, 2) = 5.99).

References

Rabe-Hesketh, S., A. Skrondal. 2008. *Multilevel and Longitudinal Modeling Using Stata*. 2nd Edition, Stata Press.

Singer, J. D., J. B. Willett. 2003. *Applied Longitudinal Data Analysis*. Oxford University Press, New York, NY.