Using Hubs and Cyclicality to Relate Software Architecture and Quality

Tyson R. Browning, Jürgen Mihm, and Manuel Sosa

June 18, 2013

Abstract

Recent studies of 17 open source applications have shown two salient characteristics of software architecture, hubs and cycles, to have strong relationships with software quality. Components in cycles were significantly more likely to contain bugs than other components, and architectures utilizing hub components tended to have fewer defects. Identifying hub and cycle components should therefore help software developers focus their quality control efforts.

Keywords: architecture, quality, cycles, hubs, degree distribution, modularity, empirical results

Software architecture matters. This is not a new revelation, since past studies have noted the significance of architectural characteristics such as coupling, cohesion, and modularity (e.g., Aggarwal *et al.* 2007; Briand *et al.* 1999; Burrows *et al.* 2010). Yet, our recent studies (Sosa *et al.* 2011, 2013) demonstrate that two additional characteristics, hubs and cycles, can also have a major impact on quality. These studies empirically link hubs and cycles to quality and identify particular ways these relationships play out. We found that components involved in cycles were significantly more likely to contain bugs and that architectures utilizing hub components were significantly less so.

Software architecture pertains to the structure of components (e.g., a Java class) and their dependencies (e.g., function calls). A cycle occurs when a component indirectly "calls" itself via a chain of other components. (Architectural cycles occur at a much higher level than the cycles measured by conventional metrics such as cyclomatic complexity.) A hub component is a relatively highly-connected component in the architecture—i.e., one with many dependencies linking it to other components.

Cyclicality Study

Our study of architectural cycles included 28,394 observations of 7,103 components across 111 major releases (versions) of 17 open source applications (an average of 6.5 versions of each application and 256 components per version) from the Apache Software Foundation in mid-2008. To collect architectural data, we downloaded the precompiled version (JAR file) of each major release of each application from the Apache archives or the application's website and used LDM, a tool developed by Lattix, Inc. (www.lattix.com), to build (instantaneously) a design structure matrix (DSM) representation from the source code and extract the module membership of components. To collect data on quality, we developed web crawlers to extract (patched) bug information on each component in each version from Apache's *Bugzilla* and *Jira* bug-tracking systems and SVN repositories. We also used each version's source code and release notes to determine a number of control variables. See (Sosa *et al.* 2013) for further details about the procedures of the study.

Two examples of the DSM representations are shown in Figures 1 and 2. The DSM is a square matrix (equal number of rows and columns) where the diagonal cells represent an element (here, a component such as a Java class) and off-diagonal cells represent directed dependencies of the component in column *j* on the component in row *i*. The DSM in Figure 1 shows the application Ant (version 1.1) with 62 components and 195 directed dependencies. (It is therefore one of the smaller applications in our study.) This DSM is called "flat" because it does not show the arrangement of the components into modules. Rather, the components in this flat DSM have been ordered to place as many of the

dependencies as possible below the diagonal. Any dependencies remaining above the diagonal are brought as close to the diagonal as possible, thereby grouping any cyclical components as closely together as possible (as indicated by the larger block shown along the diagonal). Ant 1.1 has only one cycle containing five components. The components in this group are designated as "in-cycle" components.

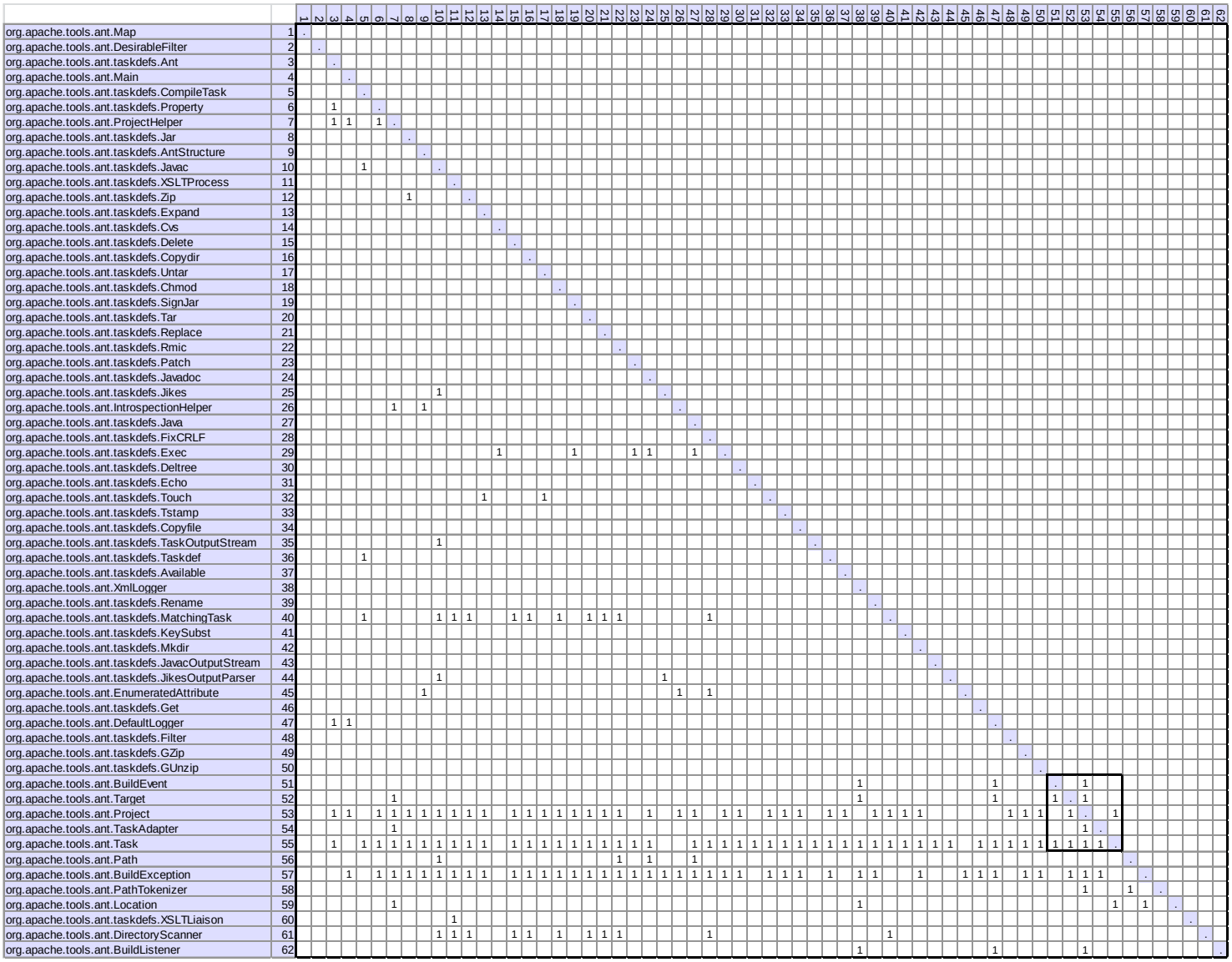

Figure 1: Flat DSM representation of Ant (version 1.1).

The DSM in Figure 2 shows the same application but with the rows and columns sorted according to the nested subdirectory structure of the code—i.e., showing the assignment of the components to the hierarchical modules designated by the architects. The modules are indicated by the differentiated shadings and blocks along the diagonal, as well as by the labeling to the left of the matrix. In this example, the five cyclical components reside inside a single module, but this is not always the case. Overall, DSMs allow us to visualize the dependency structure (or designed structure) of the architecture in various ways that help to highlight important patterns such as modularity and cycles.

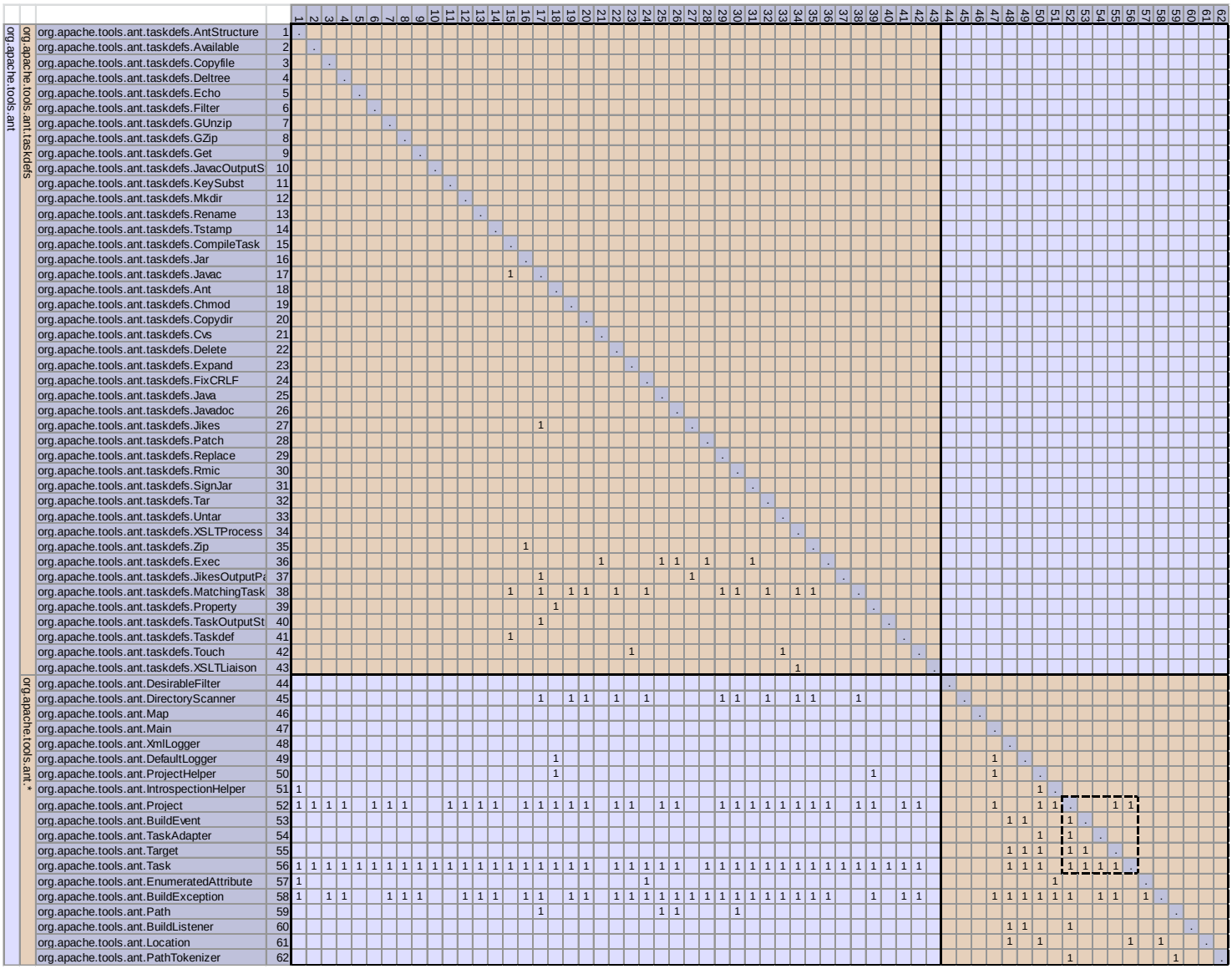

Figure 2: Hierarchical DSM representation of Ant (version 1.1).

The main dependent variable in our study of cyclicality was the number of bugs in a component, and our independent variables were component fan-in and fan-out, whether or not the component was part of a cycle, and the number of modules involved in each cycle. Component fan-out is the fraction of the other components on which the component depends, either directly or indirectly. Component fan-in is the opposite, the fraction of other components that depend on a component. Fan-in and fan-out have been studied extensively, with the latter having been shown to be a significant architectural factor.

We also included a number of system-level control variables for each version of each application, including the following: number of new and modified features since last major release, number of unpatched bugs, application age, time since last major release, application size (KSLOC), average cyclomatic complexity of all components, average fan-in and fan-out visibility of all components, and breadth and depth of the hierarchical module structure. Our component-level controls included the following for each version of each application: component age, number of non-bug changes to the component, cumulative number of changes associated with the component prior to this version, cumulative number of committers associated with the component prior to this version, cumulative

number of authors associated with the component prior to this version, average cyclomatic complexity of the component's methods, and component size (KSLOC).

Our cyclicality study yielded three main findings. First, cyclicality really hurts quality. Even when controlling for the aforementioned factors, components involved in cycles exhibited a significantly higher number of defects. Moreover, this effect increased with the size of the cycle: the larger the cycle, the greater the expected number of bugs associated each component therein. These effects were statistically significant at the 99% confidence level. The best fitting random coefficient regression model indicated that an in-cycle component had, on average, 58.3% more defects than a non-cycle component. In comparison, an increase of a single standard deviation in a component's fan-out corresponded to 35.3% more defects. Furthermore, in an analysis of a subsample of 6,064 components that were matched and balanced with respect to fan-in and fan-out, the in-cycle components averaged 80% more bugs than the non-cycle components. Overall, these results suggest that the average effect of cyclicality is of the same order of magnitude as the effect of component fan-out (modularity).

Second, the number of defects exhibited by a cyclical component increased with its centrality in the cycle. To understand centrality, consider all of the 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 circuit must exist, although some components will have more than one such path. Centrality is this number of circuits or paths. Central components will have to incorporate more potential changes propagated via their many connections to other components. Our study showed that increased component centrality correlated with more bugs. That is, components in two different cycles of the same size could exhibit different average levels of defects, depending on their centrality.

Third, failing to encapsulate cycles within a module hurts quality even more. The average number of defects exhibited by an in-cycle component increased with the number of modules involved in the cycle. In our sample 87% of the cycles spanned at least one module, which suggests that multi-module cycles are common and that, at least in an open source software development context, developers seemed to neglect the negative consequences of dealing with cyclical dependencies across modules, or were not aware of such cycles.

These results suggest some useful metrics and practices for software developers. System cyclicality indicates the percentage of components involved in cycles. It would be desirable to keep this number as low as possible, especially since the presence of cycles indicates one or more violations of the architecture's design rules. Cycles should be eliminated or at least reduced in size, especially when that reduction also reduces the number modules spanned by the cycle. At the component level, a component's membership in a cycle should be indicated, as well as its centrality in the cycle. Components in cycles, and especially those with the greatest cycle centrality, should receive extra attention and testing. Cycles can be broken by rerouting function calls and other dependencies. Cycles can be avoided in the first place by enforcing design rules that prohibit function calls to different modules, and by providing tools that show whether any potential function call would create or enlarge a cycle.

Hubs Study

Our study of hubs was similar to our study of cyclicality, although it looked at only 105 versions of 16 of the Apache applications used in the former study. See (Sosa *et al.* 2011) for further details about the procedures of the study. Our dependent variable was the number of bugs in an application version, and our independent variables were the skewness of the application version's degree distribution and its fraction of hub components.

A component's degree is its number of incoming and outgoing dependencies (the sum of its in-degree and out-degree), and a version's degree distribution is essentially a histogram showing the number of its constituent components with various degrees. It is often useful to normalize the degree

distribution by expressing its horizontal and vertical axes as percentages of the whole, and a further step can be taken to represent the cumulative version of the distribution. Figure 3 shows the cumulative, normalized, out-degree distributions for the 105 application versions in our sample. These distributions exhibit the Pareto principle in terms of a few components having a very large degree (the long tail to the right of the distribution) and the majority of components having a relatively small degree. Skewness measures this effect, where increasing (positive) skewness indicates increasing distinction among these two types of components (e.g., going from an "80-20 rule" to a "95-5 rule"). We developed a procedure for determining a degree threshold (which can differ for each application version) above which components can be identified as hubs. Note from Figure 3 the large variation in the fraction of hub components in different application versions. The average normalized degree of all components in our sample is 0.02, and the top 20% most-connected components have normalized degree 0.15 or greater. Thus, the normalized degree of a hub component (if 0.15 is used as the threshold) is more than seven times larger than that of the average component.

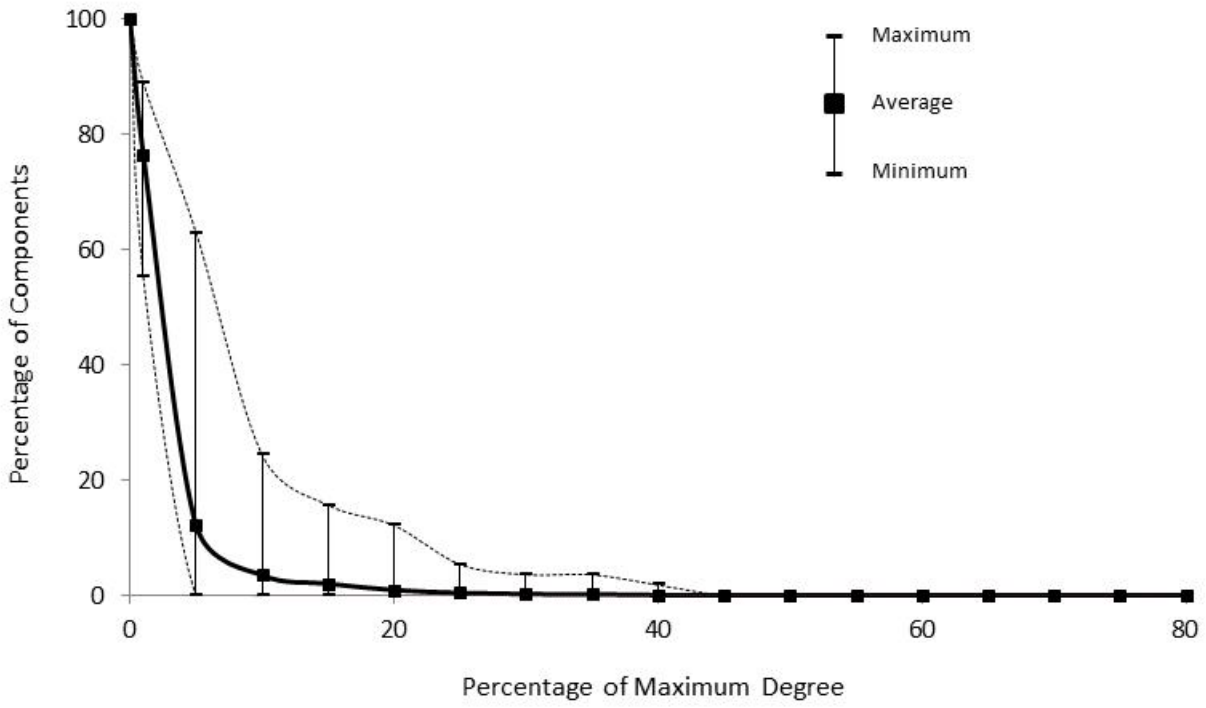

Figure 3: Cumulative, normalized, out-degree distributions for the 105 application versions in our sample (adapted from Sosa et al. 2011).

Two main findings emerged from our study of hubs. First, architectures containing hubs tended to be less defect-prone. The more right-skewed the degree distribution of an application version, the fewer defects it exhibited on average. In particular, increased out-degree skewness is significantly associated with fewer bugs (at the 99.9% confidence level).

Second, we compared different hub-designation thresholds (for both in-degree and out-degree) and found that there are optimal percentages of hub components that minimize the expected number of defects. A one standard deviation increase in the percentage of components with normalized in-degree greater than or equal to 0.15 correlated with 67% fewer defects, and in the case of out-degree it was 69% fewer defects. When the threshold is raised to 0.35, a one standard deviation increase in the percentage of components with normalized out-degree greater than or equal to the threshold correlated with 86% fewer defects.

However, it would be unreasonable to infer that merely increasing the percentage of hub components (e.g., by adding dependencies) would continually increase quality. We would expect points of diminishing and even negative returns. This is indeed what we found when we added a quadratic term to our regression models. When the in-degree threshold is set at 0.15 or greater for identifying hub components, then an application version is likely to have the fewest number of defects if 9.0% of its components are involved in hubs. The application versions in our sample had an average of 2.3% hub components at this in-degree threshold, with a maximum of 13.5%. Thus, to have fewer defects on average, an application version should have more hub components than did the average application version (9% vs. 2.3%), but applications with an even higher percentage of hubs (up to 13.5%) started to have a higher expected number of defects. We found similar results in testing higher in- and out-degree thresholds. Overall, we found substantial empirical evidence that systems with in- and out-degree distributions with "thicker than average" right tails (i.e., an above-average fraction of hub components) were more likely to have fewer defects, but that there are points past which this benefit subsides.

Previous research has focused on modularity as the most salient architectural characteristic in the architecture-quality relationship, yet these two recent studies have demonstrated that additional architectural characteristics, cycles and hubs, are similarly important. By analyzing their architectures in light of these characteristics, software developers should be able to increase their return on investment in verification and testing.

References

Aggarwal, K.K., Yogesh Singh, Arvinder Kaur and Ruchika Malhotra (2007) "Investigating Effect of Design Metrics on Fault Proneness in Object-Oriented Systems," *Journal of Object Technology,* **6**(10): 127-141.

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

Burrows, Rachel*, et al.* (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.

Sosa, Manuel E., Jürgen Mihm and Tyson R. Browning (2011) "Degree Distribution and Quality in Complex Engineered Systems," *Journal of Mechanical Design,* **133**(10): 101008.

Sosa, Manuel E., Jürgen Mihm and Tyson R. Browning (2013) "Linking Cyclicality and Product Quality," *Manufacturing & Service Operations Management,* forthcoming.

Acknowledgements

The authors are grateful to Neeraj Sangal and Frank Waldman of Lattix Inc. for use of the LDM tool and for insightful feedback on the research. The first author is grateful for support from a grant from the U.S. Navy, Office of Naval Research (grant N00014-11-1-0739). The last two authors are grateful for financial support from the INSEAD R&D committee (grants 2520-360 and 2520-519).

Biographies

Dr. Tyson R. Browning is Associate Professor of Operations Management in the Neeley School of Business at Texas Christian University, where he conducts research on managing complex projects and teaches courses on project, operations, and risk management. He has previous work experience with Lockheed Martin Aeronautics Company and other companies and has also consulted for several organizations. He earned a B.S. in Engineering Physics from Abilene Christian University and two Master's degrees and a Ph.D. from MIT. He is a member of INFORMS, INCOSE, and POMS academic and professional societies and serves on the editorial boards of *IEEE Transactions on Engineering*

Management and *Systems Engineering*. Address: TCU Box 298530, Fort Worth, TX 76129. www.TysonBrowning.com

Jürgen Mihm is an associate professor of technology and operations management at INSEAD. His research interests are concerned with all management aspects of large engineering projects. Recently, he has turned to understanding the management of design. He holds a doctorate in technology management from Wissenschaftliche Hochschule Koblenz (WHU) and a joint degree in business and electrical engineering (Dipl. Wirtsch. Ing.) from Technische Universität Darmstadt. Prior to his position at INSEAD, he was a long standing consultant with McKinsey & Company, Inc. in Frankfurt, serving mainly semiconductor and automotive clients and specializing in technology and operations management. He is a member of INFORMS and POMS and serves as a senior editor of the *Production and Operations Management* journal. Address: INSEAD, Boulevard de Constance, 77305 Fontainebleau, France.

Manuel Sosa is an associate professor of technology and operations management at INSEAD and the Director of INSEAD's Heinrich and Esther Baumann-Steiner Fund for Creativity and Business. Prof. Sosa's research efforts are applied to improving product development systems. He is particularly interested in studying coordination and innovation networks in complex product and software development organizations. His work experience includes systems engineering in the petrochemical industry and development and deployment of computer-aided engineering software applications in the automobile and aerospace industries. He received his BS degree in mechanical engineering from Universidad Simón Bolívar (Caracas, Venezuela) and his SM and PhD degrees in mechanical engineering from MIT. He is a member of INFORMS, AOM, and ASME academic societies and serves as senior editor of the new product development department of the *Production and Operations Management* journal. Address: INSEAD, 1 Ayer Rajah Avenue, Singapore 138676.