This article is a preprint of the article appearing in IEEE Software Vol. 29, Issue 1 (Jan-Feb 2012)
http://doi.ieeecomputersociety.org/10.1109/MS.2012.3

Conway's Law Revisited: The Evidence For a Task-based Perspective

Irwin Kwan, Oregon State University
Marcelo Cataldo, Robert Bosch Corporate Research
Daniela Damian, University of Victoria

A long-standing approach to deal with complex product development is to divide the system into smaller parts and assign responsibility for developing those parts to particular teams. Conventional wisdom suggests that by doing so, we reduce the communication overhead in the development project by making the separate development teams as independent as possible. This approach has been labeled in academic circles as "Conway's Law" or "the Mirroring Hypothesis". Conway (1968) argued that "development organizations... are constrained to produce designs which are copies of the communication structures of these organizations". Conway's arguments were later further developed and refined by other researchers such as Parnas (1972), Herbsleb and Grinter (1999), and Baldwin and Clark (2000). Furthermore, alignment between the technical and organizational structures results in improvements in product development productivity and quality for physical systems (Baldwin & Clark, 2000; Colfer & Baldwin, 2010). However, when it comes to developing software systems, the evidence about the "mirror" in Conway's law and its benefits to software projects outcomes is less clear.

Software systems differ from physical systems in many ways. Software interfaces are easy to change, and the impact of making a particular change is not easily understood especially in large-scale systems. Organizations also change, which complicates matters. The tacit software knowledge held by the development organization can easily depreciate or disappear as engineers leave the company or change positions. These differences require us to reconsider how we conceptualize the alignment between product and organizational structures and the implications of alignment for outcomes such as productivity and quality in software systems.

Not surprisingly, the evidence about Conway's Law in software projects is mixed. On the one hand, our own and others' work have identified socio-technical alignment in software projects. However, in a recent review of a large body of academic research, Colfer and Baldwin (2010) found that a number of the projects that did not show significant alignment were software development projects. Moreover, the authors also found no evidence that the lack of alignment between product and organizational structures was detrimental to those projects. Why is there this occasional lack of alignment and why does it seem not to affect these software projects? The answer may lie in how we conceptualize Conway's Law. The evidence suggests that achieving alignment at the level of development tasks and teams, instead of at the level of the architecture and organization, may benefit software projects the most.

Does Conway's Law matter in software development?

The short answer is yes. However, what also matters is how Conway's Law is examined in the context of software development projects. Software practitioners tend to conceptualize the structure of the product as "the software architecture" and the structure of the organization as the organizational chart, like they might in a physical system. This view is not sufficient for project success.

One issue with taking a high-level view of the organization and the software architecture is that the developers don't see their work from that perspective. They view their work usually as "a set of goals the system must accomplish" and thus, a developer's tasks involve modifying the system or arranging for others to modify the system so that it can accomplish these goals (Nordberg III, 2003). A second issue is that the architectural level misses the nuances of the developers' work. In some situations, the software architecture as a coordination mechanism can work in the organization's favor (de Souza et al, 2011), but we know of no evidence that suggests project-wide benefits (e.g. better productivity or quality) from achieving alignment. On the contrary, there are instances where problems occurred despite architectural alignment with the organization: Ovaska and colleagues (2003) as well as Bass and colleagues (2007) reported that achieving architectural-level alignment was not sufficient to enable sufficient coordination to avoid issues such as major quality problems.

This evidence, together with our past work, suggests that Conway's Law matches the unique characteristics of software development when we consider alignment at a task level rather than at the architectural level.

The Task-Level View of Conway's Law

The task-level view of Conway's Law conceptualizes a unit of work as one that developers are engaged with, such as a defect in the bug-tracking system, a feature, or a software build. When a developer works on a task, a number of units, such as files, are modified. The files that are modified together are considered to be inter-related (Gall et al. 1998). If multiple developers work on the same tasks then the rationale for the alignment is that those who work on the same tasks and files should coordinate their actions so they know what is being changed in the system.

Evidence illustrates that task-level alignment has benefits to a software project. Cataldo and colleagues (2006, 2008) found that maintaining congruence between communication and coordination action and task dependencies resulted in significantly shorter resolution times of development tasks. These results were also found in open source projects (Wagstrom et al, 2010). Sosa (2008) examining a software company found that 43% of communications were not predicted by the product dependencies at the architectural level, suggesting that the task-level may be more appropriate at which to seek such alignment. Kwan and colleagues (2011) also presented similar evidence, where teams that worked closely together were able to successfully build software when socio-technical alignment was high, but paradoxically this study also related low alignment to high performance during large integration builds.

Open source development is a special case in which research found evidence of low alignment without detrimental effects on project performance. These open collaborative projects (Colfer and Baldwin, 2010) involve independent and geographically-distributed contributors that work in the absence of a well-defined organizational structure and make highly independent contributions to the same design. Mutually compatible motivations, lack of major economic conflicts of interests and an established common ground about the evolving design are among the factors that could explain this exception to Conway's law.

How can we realize the benefits of Conway's Law in a software project?

The evidence clearly indicates benefits of achieving alignment between the dependencies developers face in their tasks and the developers' communication and coordination actions. The wealth of digital information available in software project repositories provides opportunities for researchers to further the understanding of Conway's Law at the task-level. In addition, there are amazing benefits for practitioners such as the possibility to increase development productivity or software quality. How can we realize the advantages of Conway's Law in practice? A stepwise procedure utilizing this information is outlined below.

Leverage software repositories for technical dependencies as well as developers' coordination patterns
Software repository data augmented with a team's knowledge of social interaction patterns are instrumental in providing valuable information to understand whether alignment exists and how it evolves over the lifetime of a software development project. Communication data in the form of bug reports exists alongside technical artifacts like code. They open the door for a range of analytics and tool support that can significantly improve the development organization's ability to coordinate efficiently and effectively, particularly in a globally distributed setting.

Identify task-based logical dependencies. Logical dependencies (Gall, 1998) provide valuable information about the technical relationships in a software system and complement the information that the more traditional syntactic dependencies provide. Using logical dependency information, developers can grasp a better understanding of the technical relationship between different source code files or modules, as well as the potential impact of a particular change. Tools that collect logical dependency information can display these dependencies to the developer in the appropriate context. However, it is also possible to use traditional infrastructure tools to inquire about other files that are affected whenever a particular file or artifact of interest changes.

Maintain awareness of task dependencies. This strategy consists of maintaining awareness of overlapping tasks on related components or modules, and especially of how the people involved work on these tasks. Design and implementation changes in inter-related software entities could have a significant impact in the area of developer responsibility. An approach commonly used to help developers maintain some sort of awareness of other development activities is to subscribe to the tasks of a particular team or component in the task or defect tracking systems. However, such an approach becomes ineffective pretty fast, particularly in large projects. Using information about specific dependencies, such as logical or particular work-related dependencies, helps manage the volume of information by filtering it, letting developers stay aware of those development tasks that are relevant (Costa et al, 2011) and thus facilitating socio-technical alignment.

Discover if team coordination and tasks align. Once the task dependencies are identified and people's coordination relationships are identified, a manager should examine and identify where alignment exists among team members. This means identifying whether people who are coordinating together are also working together on dependent tasks or vice versa. If there is a close match, then things are going well. If there are mismatches, there may be issues to look into. In most projects, a lack of alignment warrants further investigation. Are there gaps in knowledge among developers? Do they know which clients to talk to? When coordination involves developers you didn't expect to see, are developers contacting them because they are the experts on a specific part of the system? If such *emergent people* are being involved in communication, then you want to be aware of these hidden experts and ensure that this connection is not lost (Kwan and Damian 2011). Encouraging team members to inform you of people that they contact outside the scope of their current task or those who are not a part of your software team helps expand your awareness of critical sources of knowledge for future modifications.

Conclusion

The evidence about Conway's Law suggests that managers should take a step back from looking at a system with respect to inputs, outputs, and call graphs. They should examine instead the tasks that people have to do and how software modules influence these tasks. Coplien (2006) advises that the architect should "serve as the long-term keeper of architectural style", and suggests that the developer be in control of the process. By taking a task-level view of alignment, your organization can experience real benefits in the ability of a team to work together.

References

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

Bass, M., Mikulovic, V., Herbsleb, J., Cataldo, M. and Bass, L. (2007). Architectural Misalignment: An Experience Report. Conference on Software Architecture, Mumbai, India, Jan. 6-9.

M. Cataldo, J. D. Herbsleb, and K. M. Carley. Socio-technical congruence: A framework for assessing the impact of technical and work dependencies on software development productivity. In Proc. Empirical Software Engineering Measurement, Kaiserslautern, Germany, 2008.

M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley. Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In Proc. Computer-supported Cooperative Work, Banff, Canada, October 2006.

This article is a preprint of the article appearing in IEEE Software Vol. 29, Issue 1 (Jan-Feb 2012)
http://doi.ieeecomputersociety.org/10.1109/MS.2012.3

L. Colfer and C. Y. Baldwin. The mirroring hypothesis: Theory, evidence and exceptions. Working Paper, Harvard Business School, 2010.

M. Conway. How do committees invent? Datamation, 14(4):28–31, 1968.

J. Coplien. Organizational patterns. In I. Seruca, J. Cordeiro, S. Hammoudi, and J. Filipe, editors, Enterprise Information Systems VI, pages 43–52. Springer Netherlands, 2006.

J. M. Costa, M. Cataldo, and C. R. de Souza. The scale and evolution of coordination needs in large-scale distributed projects: implications for the future generation of collaborative tools. In Proceedings of the 2011 annual conference on Human factors in computing systems, In Proc. Computer-Human Interaction, pages 3151–3160

C. R. de Souza and D. F. Redmiles. The awareness network, to whom should I display my actions? and, whose actions should I monitor? IEEE Transactions on Software Engineering, 37:325–340, 2011.

H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In Proc. International Conference on Software Maintenance, Bethesda, USA, pages 190–198, 1998.

J. D. Herbsleb and R. E. Grinter. Architectures, coordination, and distance: Conway's Law and beyond. IEEE Software, 16:5, p. 63-70. 1999

I. Kwan, A. Schroter, and D. Damian. Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. IEEE Transactions on Software Engineering, 37:3, p. 307–324, 2011.

I. Kwan, D. Damian. The Hidden Experts in Software-Engineering Communication (NIER Track). In Proc. International Conference on Software Engineering, Honolulu, USA, 2011.

M. E. Nordberg III. Managing code ownership. IEEE Software, 20:26–33, 2003.

P. Ovaska, M. Rossi, and P. Marttiin. Architecture as a coordination tool in multi-site software development. Software Process Improvement and Practice, 8:233–247, 2003.

D. L. Parnas. On the Criteria to be Used In Decomposing Systems into Modules. Communications of the ACM 15:12, 1972

M. Sosa. A structured approach to predicting and managing technical interactions in software development. Research in Engineering Design, 19(1):47–70, 03 2008.

P. Wagstrom, J. Herbsleb, and K. Carley. Communication, team performance, and the individual: Bridging technical dependencies. Academy of Management Meeting, Montreal, Canada, August 2010.