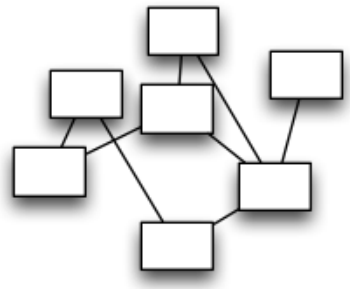
Governance in software ecosystems: achieving desirable architectures-in-use through strategies for insight into and influence over end-user development.

James Howison
jhowison@ischool.utexas.edu, School of Information, University of Texas at Austin

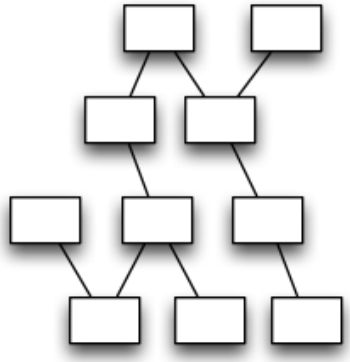
Keywords: product architecture, software ecosystem governance, socio-technical systems

Software ecosystems take advantage of the ability to combine different software components. Beyond simple re-use, recombination offers innovative potential as users discover the capabilities of new arrangements of components. However, the flip side of innovative recombination is the emergence of complexity: complexity that can overwhelm developers, leading to migration to new ecosystems (with initial lower complexity). The literature on product architectures highlights the usefulness of particular architectures—patterns of dependency among components—both for innovation and for dampening complexity from recombination. Yet since what matters is not abstract architectures but architectures-in-use, we need to know how end-user developers pursue recombination and we need to how to shape behavior towards desirable architectural patterns. In this paper I draw on qualitative examples from business, mobile, and scientific software ecosystems to highlight the way in which end-user developer behavior drives architectures-in-use and discuss approaches to governing that behavior, highlighting techniques for both insight and influence over how end-user developers pursue their innovative recombination.

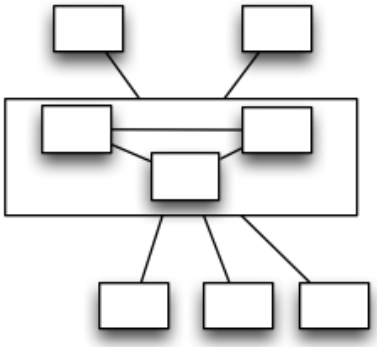
The **literature on product architectures** and complexity makes clear the advantages of particular structures of interconnection within a software ecosystem, a term introduced by Messerschmitt and Szyperski (2005). A great deal has been written about the value of modular architectures (e.g., Baldwin and Clark 2006; Langlois 2002) and platform architectures (e.g., Gawer and Cusumano 2008; Tiwana et al. 2010). In software engineering, the literature on product lines (e.g., Clements and Northrop 2002) has emphasized the value of hierarchical component architectures, especially sharing "upstream" components within firms, reducing maintenance costs and increasing the speed and quality of development.



Panel A



Panel B



Panel C

Figure 1: Structure of component co-use aggregated across software assemblies

(lines indicate components are used together in at least one software assembly)

Figure **1** depicts three idealized ecosystem architectures. In this illustration two components are connected if they are used together in at least one software assembly (whether that be as a dependency or a nearby complement). The lines represent potential paths of change impact, transferring through the software ecosystem and generating a need for adjustment at the

component. One can think of these diagrams as transmission paths, such that a change at a particular component acts like a "pulse" and can be transmitted along these lines. When that pulse reaches a connected component, adjustment work there might cause a new pulse, such that components connected to the newly adjusted component now may need to undertake adjustment work. It is easy to see that the more connections, the more potential paths for impact; yet there is more to be said than simple counts.

Panel A shows an architecture with random connections, including long-jumps and circular paths; such an architecture represents ungoverned complexity, with unpredictable and potentially cascading changes. Panel B shows a hierarchical architecture, eliminating circularity and arranging components into "layers."; such an architecture might be the goal of a "product lines" approach within a firm, or a cohesive open-source distribution, such as Debian or Redhat. Layers not only limit change impact, but allow adjustment work to be "pushed upstream" and shared out to end-user developers. Panel C shows a platform architecture, where components inside the platform have more complex interconnections, but components outside the platform connect only with the platform as a whole, the structure found on Apple's iOS and intended in many platform strategies.

Overall, then, the product architecture literature has clearly established the usefulness of recombination, identified the potential costs of complexity in patterns of recombination, and established the usefulness of particular architectures in dampening these costs. Yet the emphasis has primarily been on describing well-functioning, somewhat abstract and idealized architectures, with less emphasis on how these architectures are to be achieved. Unsurprisingly, given corporate settings, the literature on software product lines assumes a hierarchical organization able to simply assert a structure and to hold its employees and suppliers to that architecture; the literature on design rules and platform architectures considers governance, but does so assuming a central, dominant, firm (Baldwin and Clark 2006; Tiwana et al. 2010). Yet the true value of recombination occurs when the architecture facilitates widespread participation and ecosystems with a "platform owner" are only a single configuration, thus the general question of how to govern an ecosystem to achieve desirable architectures remains important.

Architectures and architectures-in-use. Software is incredibly malleable. Given a computer, code, and simple tools like compliers a developer can arrange and re-arrange software rapidly and at very low cost. This freedom at the edges is at the heart of the value of recombination, but it is also at the heart of complexity. In studying end-user developer's work one finds the "user-experience" of a software ecosystem. Qualitative vignettes from open source, mobile development, and scientific software illustrate this work as one of "bricolage" (e.g., Verjans 2005) in creating "software assemblages," groups of components linked together by patterns of compile-time dependencies but also data workflows of run-time complementarity, as components pass data between each other. Assemblages draw from components in ecosystems but usually extend beyond any one ecosystem. Holding these diverse assemblages together over time, as tasks and underlying technologies change, gives rise to frustrating work often described as "dependency hell," as developers attempt to keep their local assemblages working. Software, of course, does not decay, yet developers constantly speak of "bit rot," describing the importance of working with up to date components to take advantage of new techniques and performance requirements. Change acts like stones thrown in a pond, expanding ripples causing "adjustment" work at the edges and at components, changes that themselves potentially spark still more change. Given the active bricolage of end-user developers, the architecture that matters is the totality of connections in end-user developer assemblages; these, not abstract architecture diagrams, form the architecture-in-use that really matters to the performance and longevity of a software ecosystem.

Ecosystem governance, then, is about trying to achieve desirable architectures-in-use by understanding and influencing ecosystem participants' assemblages. Like other management tasks, then, two key needs are insight and the ability to influence.

Insight is surprisingly complicated, since many connections occur entirely in user-space and often as software is used together, rather than complied together (McConahy et al. 2012). Sources of insight can usefully be thought of in three classes. The first is attempts at prior, global, collection of requirements: this is dominant within large firms and in grant-funded scientific infrastructure projects. The second is central observation of the software ecosystem, represented in relatively novel form by the required submission of source code apps to central "app stores" in mobile platforms, but more traditionally by the activities of marketing engineers as well as in-app use/error reporting. The third insight strategy is via openness, including watching end-users download behaviors, and watching them talk and share code in public, as well as being open to end-user adjustment work, a strategy predominantly deployed by open source software ecosystems.

Influence over end-user developers is also crucial to obtaining desirable architectures-in-use. All the engineered layer diagrams in the world will not guarantee a rational architecture-in-use if end-user developers are motivated and free to recombine components in ways that generate patterns like those in Figure 1.A. The techniques available to influence end-user developer behavior are limited, especially outside the lines of traditional organizational command-and-control. A basic technique is published architectures and APIs, promising ongoing support for particular recombinations. This is often bolstered by "road-mapping" where a published plan for the future of the ecosystem promises future lines of development, motivating end-user developers to forbear immediate optimization for future support. A more aggressive strategy is the use of contract law to control access to ecosystem components: applications on Apple's iOS are forbidden to call a) each other, and b) unapproved platform APIs. Violators are removed from the App Store, losing access to customers. A third technique is to leverage "rational herding" (Devenow and Welch 1996; Oh and Jeon 2007) by making available information about how others are using the platform, encouraging coalescence on particular components and encouraging coordination between components frequently used together.

Open and uncontrolled recombination at the edges of software ecosystems is both extremely valuable and a recipe for rapid growth in ecosystem complexity. Ungoverned, this can lead to overwhelming, ongoing, multiplying, adjustment work, work that falls on the end-user developer attempting to keep their software assemblages working and up-to-date. Eventually, frustrated developers seek simplicity by migrating to a new, emergent, ecosystem currently tuned to their particular needs (the migration from the LAMP stack to the Ruby on Rails ecosystem is a great example), only for the cycle to begin again. By better understanding how complexity is managed in different ecosystems we can more reliably access their benefits, especially in domains like scientific software where the opportunity costs of creative destruction through ecosystem migration are considered too expensive.

References

Baldwin, C. Y., and Clark, K. B. 2006. “The Architecture of Participation: Does Code Architecture Mitigate Free Riding in the Open Source Development Model?,” *Management Science* (52:7), pp. 1116–1127.

Clements, P., and Northrop, L. 2002. *Software product lines: practices and patterns*, (Vol. 59) Addison-Wesley Reading.

Devenow, A., and Welch, I. 1996. "Rational Herding in Financial Economics," *European Economic Review* (40:3-5), p. 615.

Gawer, A., and Cusumano, M. A. 2008. "How companies become platform leaders," *MIT Sloan Management Review* (49:28).

Langlois, R. N. 2002. "Modularity in technology and organization," *Journal of Economic Behavior \& Organization* (49:1), p. -37.

McConahy, A., Eisenbraun, B., Howison, J., Herbsleb, J. D., and Sliz, P. 2012. "Techniques for Monitoring Runtime Architectures of Socio-technical Ecosystems," in *Workshop on Data-Intensive Collaboration in Science and Engineering (CSCW 2012)*, .

Messerschmitt, D., and Szyperski, C. 2005. *Software ecosystem: understanding an indispensable technology and industry*, .

Oh, W., and Jeon, S. 2007. "Membership Herding and Network Stability in the Open Source Community: The Ising Perspective," *MANAGEMENT SCIENCE* (53:7), p. 1101.

Tiwana, A., Konsynski, B., and Bush, A. A. 2010. "Research Commentary—Platform Evolution: Coevolution of Platform Architecture, Governance, and Environmental Dynamics," *Information Systems Research* (21:4), pp. 675–687 (doi: 10.1287/isre.1100.0323).

Verjans, S. 2005. "Bricolage as a way of life - improvisation and irony in information systems," *European Journal of Information Systems* (14:5), p. -506.