The Durability of Software

Christopher Kelty and Seth Erickson

Software is neither material nor immaterial but durable, entrenched and scaffolded. In this article we suggest that services and software should be understood through the diverse forms of durability and temporality they take. We borrow concepts from evolution and development, but with a critical eye towards the diagnosis of value(s) and the need for constant maintenance. We look at examples from diverse cases—infrastructural software, military software, operating systems and file systems.

Keywords: software, durability, maintenance, development, evolution, generative entrenchment, scaffolding.

To appear in: I. Kaldrack and M. Leeker (2015). *There is no Software, there are only Services*, Lüneburg, Germany: Meson Press.

Our goal this week is the conversion of mushyware to firmware, to transmute our products from jello to crystals.

-- Alan J. Perlis, In the Keynote speech to the 1968 NATO conference on Software Engineering, p. 138.

A Software Coelacanth

In April 2014, a *60 Minutes* report made a brief splash when it revealed that the United States live nuclear weapons arsenals are using "antique" software and hardware, such as floppy disks, microfiche and radiograph data and software written in the 1970s. The Internets mocked the hopelessly outdated technology; John Oliver's studio audience for *Last Week Tonight* audibly gasped when he showed them the image of a missileer holding an 8-inch floppy disk. Oliver's commentary: "Holy Shit! Those things barely look powerful enough to run *Oregon Trail*, much less earth-ending weaponry."

Many people accustomed to constant updates, rapid release cycles, beta-testing and automatic upgrades found the story shocking—viscerally so since it concerns the deadliest weapons on earth. The "silver lining," as a *Vice* article put it, quoting Major General Jack Weinstein, was that "cyber engineers [who analyzed the network last year] found out that the system is extremely safe and extremely secure in the way it's developed" (Richmond 2014). The dramatic tension is thus driven by something unstated (that newer technology is always safer, better, more efficient than old, *legacy* systems) in conflict with something intuitive (that it makes very good sense *not* to connect these weapons to the Internet).

The software and hardware systems that run these 1970s era Minuteman launch control systems are a kind of technological *coelacanth*: a living fossil. Isolated, highly engineered, rigorously (one hopes) maintained, but never upgraded or changed. Contrast this with what we might think of as the *cichlids* of contemporary software: mobile apps, games, websites, APIs and *services* that appear hourly, where updates are constant and the rate of extinction equally rapid. [1] The rise of "Software as a Service", "service-oriented architecture" or the "cloud" seems to suggest that a qualitative shift towards a kind of hyper-instability is taking place: instead of a stable "program" nothing but a temporary relationship of "queries" across interfaces and devices, rendering something that was immaterial even

[1] Cichlids are common, rapidly diversifying fish, comprising between 2000 and 3000 species, including things like Angelfish and Tilapia, and exhibiting a stunning diversity in morphology and behavior.

more airy and vaporous. It would seem to follow that our economy and culture are also becoming similarly cloudy—precarious, uncertain, distant, contracted.

The apparent transition—from software to services—raises a question: are they different? What is the difference, and how best is that difference described? On the one hand, one might assert that there is no difference at all because the concept of service was built into software from the very beginning. Indeed, before the word or the object *software* existed, there were programming services.[2] Software had to be "unbundled" and "productized" to achieve a stability and singularity we colloquially attribute to things like Microsoft Word or Adobe Creative Suite. Whether it be the computer utility of the 1960s or the thin clients and netPCs of the 1980s when Sun declared "the network is the computer," services have been a constantly desired goal all along. On the other hand, services today appear quite different: the ease of reconfiguration, the openness of their accessibility, the standardization of their functioning, and the reliance on a data-center-as-computer model all seem to turn software, databases, archives, indeed even whole companies into ephemeral conduits of information, query or control. Stable "productized" software disappears in place of unstable, contractual arrangements—Adobe Creative Suite becomes Adobe Creative Cloud, Microsoft Word becomes Office 365—replete with a shift from a sense of ownership to one of servitude.

But the desire to fix the difference between the two falls into an ontological trap—demanding that the difference between the two be an abstract one of properties and kinds (and rights) rather than one of temporality and use amongst humans. Software studies occasionally suffers from a philological fantasy that the conditions of operation of software are territorialized by programmability, rather than the programmability of software being *terrorized* by time. Software and service are thus an entangled set of operations which are better viewed from then perspective of duration and temporality, and in particular that of a evolutionary frame, than from the perspective of code, conduit, circuit, network, or other aspects that privilege a spatiality or an intellectual abstraction that relies on spatiality to make sense of it.

So in between the coelacanth of the Minute-Man missile software, and the cichlids of the Apple App Store lies a whole range of software existing at different temporalities and with different levels of *durability.* An evolutionary approach makes sense here, but not simply in order to describe this diversification, but to critically analyze where and how value and values—novelty, most centrally, but

[2] On "software services" see (Campbell-Kelly and Garcia-Swartz 2011; Campbell-Kelly 2009). Chun (2011) also makes this point.

also security, safety, freedom, health or risk—are structuring these temporalities. “Evolution is not just any change and stasis, but particular patterns of change and stasis, *patterns that tend to preserve ancestry*” (Wimsatt and Griesemer 2007:283).

We are far from alone in turning to the ideas of evolution-both those who create software and those who study it frequently do so. For instance, within the field of software engineering, the language of software evolution often replaces that of repair and maintenance.[3] And so-called “artificial life” researchers have long fallen prey to the fantasy that because a program evolves it must be *alive* (Helmreich 1998; Riskin 2007). More recently, Lev Manovich, among others, has adopted a loose language of evolutionary theory—but only, he insists, *as a metaphor*—to explain change over time in the domain of media-production software (Manovich 2013).

Our exploration of evolutionary theory is not metaphorical, but critical and analytic, *viz.* how to analyze populations of software differentially, and in order to diagnose the values, ideologies or cultural technologies at work in and through software. Our focus is not on code or the program, but on the *population* of software—as engineers might say, the “installed base” of software, which necessarily implies an ecology of users, designers, maintainers, as well as organizations and physical facilities that must be kept running: made durable.

The durability of software is not an internal feature of a particular software program or service, nor a feature of an abstract programmability or mathematical facet, but instead a feature of its insertion into a social, economic and cultural field of intention and expectation where it *becomes* differently. The Minute-man silo stays stable for reasons that are different than the "stability" of the Linux kernel (which changes often, in the name of a stability that maintains an unknowable range of possible uses). The becoming of a "service" such as Facebook Connect is much different than the simple query API provided by the Oxford English Dictionary. Both are services, both depend on money and humans who care about them—but the dynamics of their evolution and stasis are much different from each other.

[3] See for example the *Journal of Software* Maintenance, so called until 200, when it was renamed the *Journal of Software Maintenance and Evolution*, until 2012, when it merged with *Software Process and Improvement* to become *The Journal of Software: Process and Evolution.* There are countless examples of the colloquial use of the term “evolution” in software engineering, but there are also more precise attempts to characterize software evolution, primarily as an analysis of the *internal* evolution, or ontogeny, of a program (facilitated by the technology of versioning control systems) such as (Mens and Demeyer 2008). There is also a ubiquitous “phylogenetic” obsession amongst software programmers visible in the array of trees documenting the descent of different software (e.g. Levenez’s Unix chart, http://www.levenez.com/unix/).

Evolution therefore is not just a theory of change or duration-- it is also about how aspects of the past are preserved differentially in different ecologies. Software does not evolve the same way everywhere—like life it is constantly *diversifying*. Recognizing variation, heterogeneity, and the preservation of the past in the present can serve an important analytical and critical function: to identify the values, ideologies and cultural technologies that keep some systems stable and slowly changing while demanding that others seem to change "at the speed of thought."

Software is not immaterial—this much is clear to anyone who studies it. But nor is software a *substance*. The replacement of *software* by *services*, if such a replacement is actually occurring, may be interpreted less as an ontological or material shift, and more as a shift in the relationships of concurrency, dependency and durability—software too has "modes of existence" (Latour 2013).

In this article we borrow two notions from developmental-evolutionary theory in order to think about the patterns of change and stasis in software: *generative entrenchment* and *scaffolding*. Wimsatt and Griesemer use these terms in order to argue for a *developmental* understanding of cultural and biological evolution, as opposed to a strictly gene/meme centered (á la Dawkins) one or a "dual-inheritance model" (Richerson and Boyd 2005). This is felicitous given the concrete fact that software is always paired with the word 'development' – though we ought to be careful distinguish a "developmental biology" of software from software development as an established methodology. We argue here that durability—perhaps even "enduring ephemerality" as Chun (2011) calls it—is a result of robustness and generative entrenchment—*viz.* when software becomes foundational or otherwise locked into a network of uses and expectations, signaled by *maintenance*—another key term in our analysis—and driven by particular cultural and economic value(s). Maintenance of software, as software professionals often recognize, is not quite the same as maintaining a bridge or freeway: it is not about wear and tear or the failure of particular bits of software. Rather it is about keeping software in synch with changes and dependencies made in other software and systems (Ensmenger 2014).

Layers, Stacks, Entrenchments, Scaffolds

In most engineering textbooks, information systems are "layered" into "stacks" – often a pyramid—with material, physical layers on the bottom and an increasing ephemerality as one ascends.[4] Such layers do exist, but they are hardly ever so clean. In fact, it can sometimes be harder, more expensive or more dangerous to change a bit of software than the hardware or the infrastructure on which it is supposedly "layered" or stacked. *Generative entrenchment* is a real feature of developmental entanglements, one that generates innovations by virtue of the very necessity of the entangled part or function.[5] How these entanglements came about is a not pre-ordained or mechanical: it is matter for historical research into the development of a project, the spread of software, the standards guiding them (or failing to), and the reliance on expectations about the future of other components in a system, and the values organized in lines of force around a given software system.

Scaffolding as a concept serves a related analytical purpose. In building, scaffolding is necessary but ultimately disappears when a structure is complete (thought it often reappears for maintenance). In developmental psychology, scaffolding happens when people provide boundaries within which others can learn and develop skills. As they repeat these skills, the boundaries become less necessary. In the process of software testing, something similar happens: tools representing these boundaries (use cases, testing suites, different software environments like browsers, or common failure scenarios) are constructed around the software to test how it responds—as it is revised and improved these testing systems are torn down and disappear. As the software stabilizes and becomes more robust, it becomes generatively entrenched amongst other software systems and tools. Something of this process is captured by the process known ubiquitously in software engineering by the name of "bootstrapping": the use of one software system to design or construct another that either supplements or replaces it. Similarly, "beta testing" might also be interpreted as using real users (or early adopters) as scaffolding.

The appeal to these developmental evolutionary concepts is not proposed simply in order to provide a description of pure dynamics, complex or simple. Rather, by

[4] There are numerous meanings of the term "stack" in the history of software. Sometimes it refers to an abstract data type in a programming language (adding something to a memory stack); sometimes it refers to a layering of different technological features, as in a "protocol stack"; and a more recent, more colloquial usage (e.g. "solution stack") includes the range of tools—programming languages, package managers, database, libraries—that make up a particular web framework used for rapidly building and deploying apps in different contexts. What they share is the attempt to capture how "software" is always stacked, layered or interconnected in progress. No software is an island, etc.

[5] Blanchette (2011) discusses the example of modularity's effects on "cross-layer" innovation.

identifying dynamics and patterns, we can show how the values and the logics operate: some entrenched software is maintained and some is not, and *maintenance* implies a set of *values* that require critical interpretation (Jackson 2014; Orr 1996). Not all software is maintained because it is *economically* valuable—Minuteman III missile systems, for instance, or the software that runs a power grid. Failing to maintain it may have economic effects, but it is maintained primarily in virtue of other values: security, safety, health, mobility, secrecy, etc.[6] Even “archived” software must be maintained, and represents particular values: preservation, recovery, evidence (Kirschenbaum 2008).

Beyond Old and New

It should not come as a surprise that there is great diversity in the world of software. What is surprising is that we have no good way of taxonomizing it—or studying it—other than the language of old, outdated, or obsolete vs. “cutting edge” or new. The language of innovation privileges the linear and the incremental over the spread of diversity or the interaction of different temporalities. The supremacy of the *value* of novelty or innovation is a peculiarly modernist and Western notion: novelty at all costs! And it implies a similar and opposite mistake: to think of the old as similarly linear and incremental--as deposited, archived, forgotten and in need of constant renewal. In fact, the perspective of evolution demands a perception of newness everywhere and in many different forms that persist: the past is not superceded, but preserved, differentially and in response to a changing ecology (consisting of other things that are similarly new and old at the same time).

The key critical or analytic moment therefore is not the identification of the new, but the identification of a distinct *population*—a kind of curatorial maneuver—the drawing of boundaries around a set of instances of the same kind such that diversity and differentiation are made to appear. A few examples might indicate a different path for how to study software.

To begin with: particular populations of operating systems are arguably the most entrenched—and most generative—aspect of our software ecology. They come in many forms, from the consumer-focused iOS and Android mobile OS (which is on the order of 10 years old) to UNIX-derived operating systems (which are on the order of half a century old). Add to this the various populations that are in some

[6] In fact, there is a relatively robust economic niche where “obsolete” software is maintained, e.g. The Logical Company (http://www.logical-co.com/) which re-creates “hardware, software and diagnostic compatible” versions of DEC’s 1970s PDP computers, giving that software a new temporality and durability

ways both old and new. The OpenVMS and Alpha operating systems were originally designed in 1970s for DEC-VAX computers, but are still in use in old, new, updated, emulated and migrated forms; OpenVMS runs India Railways' reservation system and the Paris Metro's automated, driverless line 14.[7]

Similarly, entrenched programming languages (Cobol and FORTRAN) were at the heart of the Y2K hysteria. Although the predicted apocalypse did not come, it did reveal the problem of maintaining software—both its costs, and the kinds of values (in this case, fear of apocalypse) that are necessary to *disembed* entrenched software. Military systems, public infrastructure, factory process control (SCADA) systems, all contain various forms of entrenchments and dependencies—some of which are revealed dramatically (e.g. the case of the Stuxnet virus), some of which are revealed only slowly through maintenance or breakdown.

Entrenchment and scaffolding can also make sense of the variety of basic tools in use by software programmers—from compilers like gcc to programming languages, libraries and their bindings. The latter—language bindings—are a good example of generative entrenchment. Libraries of commonly used code in an operating system are often written in particular languages, such as C, C++ or assembly, often to facilitate re-use, and sometimes to make code more efficient (an algorithm in C can be made to run much faster than one in Perl, for instance). But because these libraries are "entrenched" in the operating system, they "generate" the need for bindings: bits of code that access and sometimes recompile a library for use with another programming language. Old technologies "scaffold" new ones: stories of programmers' need to re-write a program in another language (whether for efficiency or elegance, or to access parts of an entrenched system) are everyday evidence of the scaffolding process.

Indeed, in 2015, the range of new programming languages and frameworks for rapidly building and deploying software have created vast but fragile webs of entrenchment and interdependency. Web programming frameworks like Drupal and Ruby on Rails are rapidly evolving – the underlying programming languages (e.g., Ruby and php) are relatively new, the frameworks themselves are evolving as their developers refine their approaches to the web, and (perhaps most importantly) the individual modules and plugins for extending these frameworks lead to a kind of "dependency hell." One commentator (Hartig 2014) reflecting on this historical difference in software said "compiling a C program from more than 20 years ago is actually a lot easier than getting a Rails app from last year to work,"

[7] See for example: http://h71000.www7.hp.com/openvms/brochures/indiarr/
and http://en.wikipedia.org/wiki/Paris_M%C3%A9tro_Line_14

a clear indication, as evolutionary theory predicts, that innovations are abundant, but not necessarily advantageous.

Some kinds of software are *not* generatively entrenched, even if they persist in time or remain durable. The Minuteman missile base is an example—no other software or hardware depends on the software created to control those missile launch facilities, but it is nonetheless durably maintained as a closed system.

Other software is maintained *because* it is entrenched—both technically and culturally. Take for instance the whole system of software that makes an abstraction of “a file” possible: file systems, memory allocation, attributes and permissions, and directory hierarchies. As the authors of a Microsoft Technical Report (Harper et al. 2011) point out, the concept of “file” as a unit of data with associated attributes (e.g., ownership, permissions) and canonical actions (copy, edit, delete) has proven to be remarkably robust, changing little over the last forty years. Most operating systems are built around files, which manage allocation of memory and access to data; “pipes” and “files” were central to the design of UNIX, which treats the *everything* as a file, including external devices like printers (accessed through “device files”). Humans are also built around files: we expect them to function in particular ways, to be stable and findable, to be *ownable* and *sharable.*

Although the file is a seemingly essential concept, it is challenged by “service oriented computing” or “cloud computing” where new a kind of “social” data is associated with files, and where files exist simultaneously on multiple devices. In this case it is not so much a particular piece of software code, but an essential “abstraction” (and an implied set of interoperable components) that is entrenched both in the hardware, and in the expectations of users. It is *generative* because “the file” cannot simply be replaced *in toto,* but rather must be “piecewise” re-engineered, guided by particular values.

Blanchette’s example (2011) of the Google File System demonstrates that even if “the file” is not what it used to be, we still need the abstraction as a way to get the “file” to appear manipulable and stable on a set of virtualized servers (preserving it, and further entrenching it). Engineers might agree that there are “better” ways to do things, but the file cannot be so easily disembedded from both human and machine consciousness.

But: *it is changing*. “Scaffolding” can help us see how. iOS and Android operating systems both “hide” files from users. They are not yet gone—the OS still relies on them—but are embedded inside an “app” which has very quickly become the primary mode of interacting with software on most devices (Apple 2014). It is very hard to “open a file” on a phone or iPad, because the system is designed to hide files and metadata about files inside an app—which is now intended to be the primary abstraction for humans. For most purposes, however, "apps" do not require users to open or close important files, and they solve the question of "where is my stuff?" by putting it inside the app (and "in the cloud"). This creates a kind of scaffold whereby users can change from an understanding of apps that open files, to one where apps have data and resources tied to users, accounts and devices. Some populations change faster than others.

This transition, however, is not simply an evolutionary fact. Rather, by understanding the generative entrenchment and scaffolding of files and apps, we can turn a more critical eye on what are otherwise simply dubbed technological or engineering concerns. Among other things, the file abstraction supports a particular model of property rights in which digital objects are literally designed around stable property ownership: files must have owners and permissions. Apps, by contrast, are designed around a different field of rights and laws: contracts and terms of service—specifically non-negotiable contracts in which the app provider has significantly more rights than the app user.

This is the "cultural technique" at the heart of the transition from "software" to "services": 20th century intellectual property rights law was designed for intangible property fixed in "tangible media" and the myriad ways in which media was so fixed in the era of "film, gramophone and typewriter." Contract law, by contrast is not about a relationship between the intangible and the tangible, it is about the fixed duration of a relationship of trust, and a way of structuring the future in terms of liability and responsibility. It is not an either/or situation, but as more and more users enter into contracts, instead of purchasing property, the software itself changes to support this cultural practice.

Apodosis: Legacy

The word "legacy" is one with a precise meaning in the history of information technology. Legacy systems are every IT manager's bogeyman; they are the cause of "lock-in" they are the emblem of the evils of "proprietary" software; they are the cause of Y2K bugs and the scourge of cyber-security, they represent the evils of corporate capitalism, the domination against which "free software" and "open source" are often pitched in battle.

But if evolution is "particular patterns of change and stasis *that preserve ancestry*" then there is no way out of a legacy. Not all legacies are equally momentous, however, just as not all inheritances are equally large. We would do well to develop a better understanding of how ancestry has been and is preserved in software systems, if we want to make any claim that innovations like "software as a service" actually represent some break with the past. On the contrary—some "services" will become entrenched; the seemingly flexible "solution stack" of today is the legacy system of tomorrow. Even more importantly, there is no single legacy, but a pattern of differences: a diversification with respect to environment. And if we want to analyze the difference these differences make, we must move away from treating software as substance—whether material substance or thought substance: program, code, algorithm.

Actor Network Theory makes a simple point here: we do not live in a world with humans as the foundation, nor in one simply run by the automaticities of machines, but in a world of relations and modes. The difference that software makes depends on how it is inserted into the relations amongst our associations—but it is not inserted the same way everywhere. The effect of software—the difference it makes—depends on the "patterns of change and stasis which preserve ancestry" at play in any given case.

Thinking in terms of scaffolding and generative entrenchment might be an antidote to the relentless anti-humanist teleology so common in both popular and scholarly thinking. That teleology—a kind of neo-Spencerianism—is driven by punditry and criticism that demands of software (and technology generally) that it obey a law of ever-complexifying, ever-accelerating progress towards either the domination of some imagined all-powerful capitalism or the liberation-destruction of some fantasized autonomous artificial intelligence.[8] This Refrain of Constantly Accelerating Change contains a grain of truth—software has enabled new patterns, new durabilities—but it misses the *existence* of diversity in the world, and the ways in which it preserves ancestry. To view software evolution as an institutionally and culturally heterogeneous object might allow us to critically diagnose its real effects, rather than running ahead to the next new thing in order to declare its sudden dominance, and the irrelevance of all the rest.

We thank the Part.Public.Part.Lab members for valuable conversation and feedback, Irina Kaldrack and Martina Leeker for incisive comments, and the Digital Cultures Research Lab of Leuphana University for the invitation to contribute.

[8] See, for example, Ian Bogost's op-ed on the subject "The Cathedral of Computation" *Atlantic Monthly,* 15 Jan 2015; http://www.theatlantic.com/technology/archive/2015/01/the-cathedral-of-computation/384300/

Bibliography

Apple. 2014. “File System Basics.” *iOS Developer Library*. http://goo.gl/6icJ4u.

Blanchette, Jean-François. 2011. “A Material History of Bits.” *Journal of the American Society for Information Science and Technology* 62 (6): 1042–57. doi:10.1002/asi.21542.

Campbell-Kelly, Martin. 2009. “Historical reflections: The Rise, Fall, and Resurrection of Software as a Service.” *Communications of the ACM* 52 (5): 28. doi:10.1145/1506409.1506419.

Campbell-Kelly, Martin, and Daniel D. Garcia-Swartz. 2011. “From Products to Services: The Software Industry in the Internet Era.” *Business History Review* 81 (04). Cambridge University Press: 735–64. doi:10.2307/25097422.

Chun, Wendy. 2011. “Programmed Visions Software and Memory.” Cambridge, Mass.: MIT Press,.

Ensmenger, Nathan. 2014. “When Good Software Goes Bad: The Surprising Durability of an Ephemeral Technology.” In *MICE (Mistakes, Ignorance, Contingency, and Error) Conference*. Munich. http://homes.soic.indiana.edu/nensmeng//files/ensmenger-mice.pdf.

Harper, Richard, Eno Thereska, Siân Lindly, Richard Banks, Phil Gosset, William Odom, Gavin Smith, and Eryn Whitworth. 2011. *What Is a File? Microsoft Research Technical Report MSR-TR-2011-109*. Redmond, WA.

Hartig, Pascal. 2014. “Building Vim from 1993 Today.” http://passy.svbtle.com/building-vim-from-1993-today.

Helmreich, S. 1998. “Recombination, Rationality, Reductionism and Romantic Reactions:: Culture, Computers, and the Genetic Algorithm.” *Social Studies of Science* 28 (1): 39–71. doi:10.1177/030631298028001002.

Jackson, Steven J. 2014. “Rethinking Repair.” In *Media Technologies: Essays on Communication, Materiality and Society*, edited by Tarleton Gillespie, Pablo J. Boczkowski, and Kirsten A. Foot, 221–39. Cambridge, MA: MIT Press.

Kirschenbaum, Matthew. 2008. *Mechanisms: New Media and the Forensic Imagination*. Cambridge Mass.: MIT Press.

Latour, Bruno. 2013. *An Inquiry into Modes of Existence: An Anthropology of the Moderns*.

Manovich, Lev. 2013. “Software Takes Command Extending the Language of New Media.” London: Bloomsbury Publishing,.

Mens, Tom, and Serge Demeyer. 2008. “Software Evolution.” New York;;London: Springer.

Orr, Julian E. 1996. *Talking About Machines: An Ethnography of a Modern Job*. Ithaca, N.Y: ILR Press/Cornell University Press.

Richerson, Peter J., and Robert Boyd. 2005. *Not by Genes Alone: How Culture Transformed Human Evolution*. University of Chicago Press. http://books.google.com/books?id=dU-KtEVgK6sC&pgis=1.

Richmond, Ben. 2014. “America’s Nuclear Arsenal Still Runs Off Floppy Disks.” *Motherboard (Vice Magazine)*, April. http://motherboard.vice.com/read/americas-nuclear-arsenal-still-runs-off-floppy-disks.

Riskin, Jessica. 2007. *Genesis Redux: Essays in the History and Philosophy of Artificial Life*. Chicago: University of Chicago Press.

Wimsatt, W. C, and J. R Griesemer. 2007. “Reproducing Entrenchments to Scaffold Culture: The Central Role of Development in Cultural Evolution.” In *Integrating Evolution and Development: From Theory to Practice*, edited by Roger Sansom and Robert N Brandon, 227–323. Cambridge, MA: MIT Press.