PREPRINT: *Complexity*, Nov/Dec 2007.

Putting Complex Systems to Work

Russ Abbott
The Aerospace Corporation
and
California State University, Los Angeles
Russ.Abbott@GMail.com

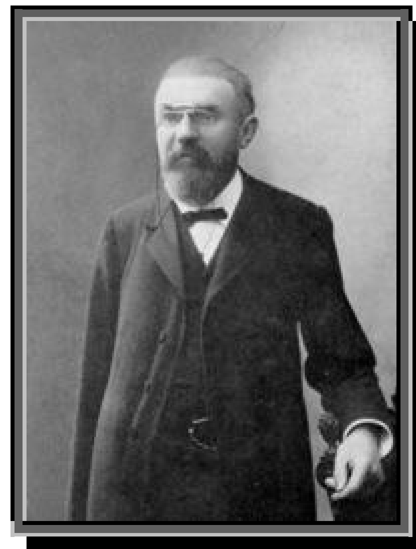
Abstract. This paper explores how concepts from the field of complex systems can be applied to systems engineering.

1 Introduction: complexity

This paper explicates certain perspectives developed in the study of complex systems, and it discusses how those perspectives can be applied to systems engineering. It includes new material as well as material from existing sources.

Fundamental to both complex systems and systems engineering are the questions: (a) how do certain kinds of systems work—typically large, multi-scalar systems—and (b) how should we describe how these systems work. The second question raises the issue of what it means to describe a system. This question in turn leads to the issue of what we intend by the term *complexity*. I'd like to start there.

Let's call a model of a system—the system could be either man-made or naturally occurring—forwardly predictable if it enables one to project the current state of a system into the future through a closed-form formula or other computationally simple computation,. Not all models are forwardly predictable. A model may either be theoretically unpredictable or it may be so computationally intractable that it isn't practically useful for projecting into the future. Newton's law of gravity is an example of the second kind. As Poincaré showed, there is no closed-form formula for predicting the future state of three or more bodies under mutual gravitation. At best all one can do is to incrementally propagate the current state into the future. In contrast, Newton's law is forwardly predictable for two bodies.



Henri Poincaré
Photograph from the frontispiece of the 1913 edition of 'Last thoughts.' Retrieved from Wikipedia Commons: http://commons.wikimedia.org/

Computability theory provides an example of the first kind of unpredictability. It is well known that there is no closed-form formula that will yield the end result of a computation even if given complete knowledge about the program perform-

ing the computation and the current state of the computation. Again, the best one can do is run the computation and see what happens.

It is characteristic of systems that are intuitively considered complex that they tend not to be forwardly predictable. But some systems have properties that seem to be both forwardly predictable and not forwardly predictable at the same time. An example is the Game of Life. It is known that the Game of Life is Turing complete. It is possible to use it to simulate a Turing machine. So the Game of Life is not forwardly predictable. Yet there are many completely predictable Game of Life patterns. For example one can trivially predict the velocity and future positions of a glider on a Game of Life grid—assuming that there are no live cells in its way.

So here we have what may look like a minor paradox. We have both (a) *complexity:* there is no general way to predict how a Game of Life configuration will turn out, and (b) *simplification:* a higher level construct such as a glider may be described and predicted much more simply than the lower level on top of which it was created.

Gliders are so trivial, though, that probably very few people would think of this as a paradox. Of course a glider is predictable. So what? In [1] I explained that a glider is an implementation of a level of abstraction. As such it obeys laws that are independent of the underlying system that may implement it. These laws may be much more predictable that those of the underlying system.

In that same article, I also noted that because of the seeming inversion of control—a higher level of abstraction appearing to control a lower level, e.g., the glider turns cells on and off as it goes by—there may be a temptation to look for an explanation in terms of downward causation. But of course this isn't the case. Instead, gliders illustrate what I called *downward entailment*.

Along similar lines, may people experience surprise when they understand how flocking behavior can be produced by simple agent rules. Just as a gliders result from rules that are capable of arbitrary Turing computations, agents that flock obey rules composed of primitive operations, which if arranged differently could produce much more complicated behaviors. But once those rules are in place, the resulting flocks behave in ways that are much simpler to explain at the flock level than would be required were that flock-like behavior to be explained in terms of the rules of the individual agents.

The popular literature has come to use the term *emergence* to describe situations such as these. Shalizi [2] captures the algorithmic complexity sense of this understanding of *emergence* as follows.

> One set of variables, *A*, emerges from another, *B* if (1) *A* is a function of *B*, i.e., at a higher level of abstraction, and (2) the higher-level variables can be predicted more efficiently than the lower-level ones, where "efficiency of prediction" is defined using information theory.[1]

[1] The extract is from Shalizi's website: http://www.cscs.umich.edu/~crshalizi/notebooks/emergent-properties.html. Accessed January 10, 2007.

1.1 Constraints, the hierarchy of the sciences, and the principle of emergence

Although Shalizi's definition nicely characterizes the algorithmic complexity issues, it misses an important point. Emergence occurs when a system is constrained in some way. The Game of Life rules that create a glider could compute any computable function. But they were constrained by the states of the cells to create a glider. A collection of agents could behave in any of a very large number of ways. But they were constrained by the arrangement of primitives operations in their rules to behave in a way that produced flock-like behavior.

If one thinks about it, how else but through constraints is it possible for the algorithmic complexity of a system to be reduced? A description of a system is a description of the system. If one description—in terms of "higher level variables"—is simpler than another—in terms of lower "level variables"—it can only be because the simpler description takes advantage of some structure that the more complex description ignores. In other words, the higher level results from some constraints imposed on the lower level.

Physicists use the term *broken symmetry* for certain kinds of constraint imposition. In his landmark 1972 paper "More is Different" [3] speculates that the hierarchy of the sciences may result from symmetry breaking. Indeed the paper's subtitle is "Broken Symmetry and the Nature of the Hierarchical Structure of Science." I suspect—but am not in a position to argue—that there is a significant relationship between the physicist notion of broken symmetry and the sort of constraint imposition that occurs when a level of abstraction is implemented.

Constraint imposition is the day-to-day activity of software development. Whenever we program a computer—or the Game of Life—we break the symmetry of all the possible ways in which the computer may operate by constraining it to operate in some particular ways. The more complex the software, the more constrained the computer.

It shouldn't be surprising that levels of abstractions give rise to new laws. To implement a level of abstraction is to impose constraints. A constrained system almost always obeys laws that wouldn't hold otherwise. The *principle of emergence* helps describe what exists. Extant levels of abstraction—naturally occurring or man-made, static (at equilibrium) or dynamic[2] (far from equilibrium)—are those whose implementations have materialized and whose environments support their persistence. It is as a result of the principle of emergence that the hierarchy of the sciences comes into being.

[2] Statically emergent structures (for example, atoms, molecules, and solar systems) are held in place by energy wells. Dynamically emergent structures (for example, living organisms, social and political organizations, and, strikingly, hurricanes) are held in place by imported energy. Dynamically emergent structures occur far from equilibrium and depend on the environment for a continuing supply of energy. Static and dynamic emergence are discussed below.

1.2 The reductionist blind spot

Gliders—like all Game of Life patterns and like levels of abstraction in general—are epiphenomenal. They have no causal power. It is the Game of Life rules that turn the causal crank. Why not reduce away these epiphenomena?

Reducing away a level of abstraction results in a *reductionist blind spot*. No equations over the Game of Life grid can describe the computations performed by a Turing machine—unless the equations themselves model a Turing machine.

This is a second aspect of emergence that a purely information theoretic view misses. The relationships between lower and higher level variables are much less important than the emergent properties themselves. In fact, the higher/lower relationships are typically a matter of convenience rather than necessity.

To capture this aspect of the issue, I defined *emergence* as the situation in which one can describe properties of a system in terms that are *independent* of its implementation. Consider the hardness of diamonds,[3] a simple example of static emergence. The higher-level property *hardness* appears *ab initio* at the level of the diamond. A diamond is hard because of how its component carbon atoms fit together. But the notion of a collection of carbon atoms fitting together is expressible only at the collection level, i.e., only at the level of the diamond itself.

A concept that comes into being at a given level of abstraction is typically inexpressible at lower levels of abstraction. This is the case for any formal system. Consider Peano's Axioms as a definition of the Natural numbers. Note especially the italicized terms.[4]

1. *Zero* is a *number*.
2. If x is a *number*, the *successor* of x is a *number*.
3. *Zero* is not the *successor* of a *number*.
4. Two *numbers* of which the *successors* are *equal* are themselves *equal*.
5. (Induction axiom.) If a set *S* of *numbers* contains *zero* and also the *successor* of every *number* in S, then every *number* is in S.

These axioms define the terms *zero, number, successor*, and *equal*, terms that are not defined in terms of lower level concepts. They are defined as primitives. The level of abstraction in which one works with the Natural numbers is autonomous.

---

[3] A diamond offers a nice example of downward entailment. Because the carbon atoms of a diamond implement a rigid lattice, the position and orientation of the diamond as a whole downwardly entail the positions of its components.

This is a good illustration of the significance of multi-scalar phenomena. At one level, the diamond as a whole moves through space. At another the diamond is maintained as a rigid lattice structure. We have two almost independent collections of phenomena that operate on different scales but that are sufficiently linked to produce the effect of downward entailment.

[4] This version of is from Wolfram's MathWorld (http://mathworld.wolfram.com/PeanosAxioms.html), italics added. Accessed August 30, 2007.

Definitions of this sort formalize levels of abstraction. For any level of abstraction, if sufficiently well understood, it should be possible to formulate similar axioms that relate the abstractions (the types and operations) that exist at that level of abstraction. As a practical matter most levels of abstraction are far too complex for such formalizations. Furthermore, some levels of abstraction may not be as well pinned down as the Natural numbers. But the point is that even if one could axiomatize a level of abstraction at best all one could hope to accomplish would be to define its primitive terms with respect to each other. One shouldn't expect new terms that come into being when representing concepts defined at a level of abstraction to be definable in any other way. That's why reducing away a level of abstraction—epiphenomenal though it may be—will result in a reductionist blind spot.

1.3 Emergence and requirements

Systems engineers are familiar with emergence as the requirements that a system must satisfy. Consider what we would now consider a simple system such as an automobile. A primary requirement is that it can be driven from here to there. This property is emergent. It is not meaningfully applied to any of the components of the automobile. Nor is it expressible as a closed form mathematical function of the automobile's components. Thus systems engineering (in fact, engineering in general) may usefully be understood as the design and development of systems that have desired emergent properties. As Rechtin put it,[5]

> A system is a construct or collection of different elements that together produce results not obtainable by the elements alone.

Virtually any engineered product illustrates emergence in this sense. Consider the fact that computers perform binary arithmetic. Binary arithmetic (or any other kind of arithmetic) has no relevance as a property of any of the components that we use to build computers. There is no sense in which one can say that properties of arithmetic apply to logic gates. Binary arithmetic is defined autonomously at the computer level and is not applicable to the components of which the computer is constructed.

This becomes especially clear when one considers software. Consider virtually every software system, e.g., a word processor, a payroll system, an image processing system for photographs. The concepts the user understands himself to be manipulating when working with those programs—words, salaries, sharpness—don't exist within the computer except at the level of abstraction defined by the software. They certainly are not meaningful at the computer hardware level.

2 Emergence and entities

In the preceding, emergent properties were defined in terms of a higher level entity—a diamond, a flock, or an automobile—generally using language that is not even applicable to the compo-

[5] From the INCOSE website: http://www.incose.org/practice/fellowsconsensus.aspx. Accessed January 3, 2007.

nents of the entity. If one thinks about it, this is quite strange. What are these higher level entities? On what ontological grounds do we permit ourselves to speak about them? I have already acknowledged that they are epiphenomenal. Are such higher level entities objectively real in any meaningful way? Is a diamond, a flock, or an automobile (or any other system) a real entity? Or is it simply a collection of its components?

In [4] I define an entity as an instance of a level of abstraction and conclude that entities are objectively and recognizably real for two reasons. (a) They have either more or less (but not the same) mass as the combined mass of their components considered separately. (b) They bind their components together in a form that reduces entropy.

There are two kinds of entities: static and dynamic. Static entities (for example, atoms, molecules, and solar systems) maintain structure because they exist in energy wells—and hence have less mass as an aggregate than their components. Dynamic entities (for example, living organisms, social and political organizations, and, strikingly, hurricanes) maintain structure by using energy they import from outside themselves. Because of the flow of imported energy, they have more mass as an aggregate than the combined mass of their components without the energy flowing through them.

Entities have emergent properties that are defined at the level of abstraction of which the entity is an instance, i.e., at the level of the entity itself. That a government is democratic or that a diamond is hard are properties defined at the level of the government or the diamond. They are not properties of the components of a government or a diamond.

2.1 The wonder of entities

If entities are objectively real, one might wonder whether they spring into existence fully formed. How might that be possible? Because this seems so mysterious, one may be tempted to speak of mechanisms for self-organization. I see this as a distraction. There is nothing mysterious about how entities form. Static entities form as a result of well understood physical laws: atoms are created from elementary particles; molecules form from atoms; etc. Dynamic entities also form as a result of natural processes. Governments form when people create them—either explicitly or implicitly. Hurricanes form when the atmospheric conditions are right. Admittedly it is still an open question how one might form a biological cell "from scratch." Currently there is no known mechanism for producing a cell other than through cell division, i.e., from an existing cell. How did the first cell form? We don't yet know. But I am confident we will soon be able to create a cell "from scratch." Self-organization is not the point.

The marvel of entities is not in some seemingly magical process of self-organization; the marvel is that entities exist at all and that they have properties and behaviors that in some sense may be described autonomously. How is that possible? How can something that is altogether new and that has new properties appear apparently from nowhere?

The answer is that the question does not have as much content as it seems. The "new properties" we attribute to entities are really nothing more than ideas in our minds. Properties as such don't exist in nature. Entities are what they are no matter what properties we attribute to them. This is not say that an

entity's "new properties" are fictitious. Hemoglobin, for example, can transport oxygen. But the property of being able to transport oxygen, while true of hemoglobin, is not a label one finds attached to hemoglobin molecules. The conceptualization of the ability of hemoglobin to transport oxygen as a property of hemoglobin is an idea in our minds. So there is not new able-to-transport-oxygen property that mysteriously springs into existence. Because of its structure hemoglobin is able to transport oxygen. But hemoglobin also has a certain mass and other properties we may (or may not) chose to characterize. Hemoglobin's ability to transport oxygen simply is.

2.2 Mechanism, function, and purpose

A fundamental question with regard to dynamic entities is how to use the terms *mechanism*, *function*, and *purpose* when speaking about them. In his talk at the 2006 Understanding Complex Systems Symposium Eric Jakobsson made the point that biology must be equally concerned with both (a) what biological organisms do (i.e., their functionality) and (b) the mechanisms that allow them to do it. Although equally important, Jakobsson didn't mention purpose, i.e., why organisms perform the functions they do—what benefit does it bring them. This section attempts to clarify these concepts.

2.2.1 Mechanism

I define a mechanism as a closed collection of elements whose interactions are well understood and whose workings can be predicted in terms of that understanding.[6] This definition intentionally ignores issues of non-determinacy and chaos. The sorts of mechanisms I have in mind are any of the standard models of deterministic computation or other intuitively mechanistic constructions. Non-determinacy can be added if necessary.

2.2.2 Function

I define the functionality of a mechanism (for an entity that embodies that mechanism) to be the change the mechanism produces in the entity's environment or in its relationship with its environment.

A simple input-output relationship is a special case of a change to the environment caused by a mechanism. For example, if one conceptualizes a Turing Machine as a finite state mechanism that interacts with an environment that consists of a tape, the functionality of

[6] It is striking how little help standard dictionaries provide when attempting to define *mechanism*. The current philosophical literature is not much more help. The two currently most widely used philosophical definition of mechanism are as follows.

> A mechanism for a behavior is a complex system that produces that behavior by the interaction of a number of parts, where the interactions between parts can be characterized by direct, invariant change-relating generalizations. [5]
>
> Mechanisms are entities and activities organized such that they are productive of regular changes from start or set-up to finish or termination conditions. [6]

It would seem that a mechanism is a level of abstraction that cannot be defined in lower level terms.

the Turing machine is characterized by the function it computes.

More interesting are the sorts of functionality biological organisms perform. A lovely example is *E. coli* locomotion. When placed in a non-homogeneous nutritive medium, *E. coli* bacteria tend to move in the direction of the greater concentration of the nutrient. In doing so *E. coli* realize a certain functionality: they move up the nutrient gradient. Through lots of good scientific work, we now understand the mechanisms that produce this result. Here is how Harold [7] explains it.

> [*E. coli*] movements consist of short straight runs, each lasting a second or less, punctuated by briefer episodes of random tumbling: each tumble reorients the cell and sets it off in a new direction.
>
> Cells of *E. coli* are propelled by their flagella, four to ten slender filaments that project from random sites on the cell's surface. … Despite their appearance and name (from the Greek for whip), flagella do not lash; they rotate quite rigidly, not unlike a ship's propeller. … A cell … can rotate [its] flagellum either clockwise or counter-clockwise. Runs and tumbles correspond to opposite senses of rotation. When the flagella turn counter-clockwise [as seen from behind] the individual filaments coalesce into a helical bundle that rotates as a unit and thrusts the cell forward in a smooth straight run. … Frequently and randomly the sense of the rotation is abruptly reversed, the flagellar bundle flies apart and the cell tumbles until the motor reverses once again.

So this is the first step in understanding *E. coli* locomotion: it engages in a random walk. But we know that *E. coli*'s motion is not random; it moves up nutrient gradients. How does it do that? Here is Harold again.

> Cells [which happen to be] moving up the gradient of an attractant … tumble less frequently than cells wandering in a homogeneous medium: while cells moving away from the source are more likely to tumble. In consequence, the cell takes longer runs toward the source and shorter ones away. [7]

How can a cell "know" whether it is traveling up the gradient or down? It measures the attractant concentration at the present instant and "compares" it with that a few milliseconds ago.

> *E. coli* can respond within a millisecond to local changes in concentration, and under optimal conditions readily detects a gradient as shallow as one part in a thousand over the length of a cell. [7]

In other words, *E. coli* has a memory of sorts. It has an internal state, which is set by the previously sensed concentration of attractant and which can be compared to the currently sensed concentration.

Given this work, we now understand the *mechanism* whereby *E. coli* performs the *function* of moving up a nutrient gradient.

2.2.3 Purpose

That *E. coli* is capable of performing a function that moves it up a nutrient gradient is an important element of the means whereby it perpetuates itself. Like all dynamic entities, *E. coli* must acquire energy from the environment. Its movement toward greater concentrations of nutrients facilitates that process. This all seems very straightforward.

It is an explanation of this sort that I refer to as *purpose*. If *mechanism* refers to the internal operations that an entity performs and *functionality* refers to the change in the relationship between an entity and its environment, I use the

term *purpose* to refer to the consequence for the entity of the change in its environment or its relationship with its environment.

If one were to ask what the purpose is of *E. coli*'s gradient climbing functionality, the answer would be to put itself in a position in which it is able to acquire more energy. Purpose is related to future functionality. *E. coli* moves up a nutrient gradient so that it will be in a position to make use of another bit of functionality, i.e., it ability to acquire energy from an external source, the nutrient.

In some cases, purpose is related to group rather than individual future functionality. This is especially clear in insect colonies. Individual insects behave the way they do, i.e., they accomplish some functional interaction with their environment, so that[7] the colony as a whole (or other members of their colony) will be in a position to exercise certain functionality. Bees, for example, bring nectar back to the hive, and ants leave pheromone trails not primarily for their own use but for the use of other members of their colonies.

D. S. Wilson discusses group evolution in [8]. The basic mechanism of group evolution involves behavioral rules that members of the group adhere to (such as bees bringing nectar to the hive) that help the group as a whole to persist. So it is the group behavioral rules that really persist rather than the group or individuals. This applies to cultural and societal groups as well as to insect and animal groups.

My use of the term *purpose* is not intended to convey intelligence, planning, intention, anticipation, or deliberation. As such it is contrary to most standard definitions. For example, the WordNet entry for *purpose*[8] (which is similar to most) has as its first entry "an anticipated outcome that is intended or that guides your planned actions". That is not my intent. I am not intending *purpose* to require an entity that can anticipate an end result to be achieved. *E. coli* doesn't run and tumble with intelligence, planning, intention, anticipation, or deliberation. It doesn't imagine itself fat and happy in a bath of high density nutrients. Its running and tumbling isn't a deliberate attempt to achieve an imagined end state. Presumably *E. coli* is not capable of anticipating end states at all.[9]

2.3 Self-persistence

Systems engineers tend to build special kinds of entities which are intermediate between static and dynamic entities. Prigogene coined the term *dissipative system* (see, for example, [9]) for a static entity that exhibits regularities when energy is pumped through it. Most of the widely cited examples of dissipative sys-

---

7 "So that" is an intentional invocation of group purpose.

8 See http://wordnet.princeton.edu/perl/webwn?s=purpose. Accessed September 10, 2007.

9 An entity may "anticipate an end state" if it has the ability to form an internal representation of its external world. To anticipate an end state would mean to have an internal representation of the that state. Presumably the internal representations of the current and future states would differ. The "purpose" of a bit of functionality would be to perform actions that modify the current state so that it will conform to the anticipated future state. Section 7.3 discusses this issue further.

tems consist of relatively unstructured static entities that exhibit somewhat surprising structures—e.g., Rayleigh-Benard convection patterns—when they are forced to respond to energy inputs.

But virtually any static entity will exhibit some response to an energy flow—especially when that energy flow is both sufficient to have some noticeable effect on the entity and moderate enough not to destroy it. Much of what engineers build, e.g., automobiles and computers, are static entities whose (necessarily dissipative) responses to energy flows are in some way useful to us.

A dissipative system is intermediate between a static entity and a dynamic entity in that it consists of a static entity skeleton (which is more or less stable without an energy flow) through which one pumps energy. Dynamic entities do not have such stable static skeletons. Dynamic entities depend on their own ongoing processes to maintain their structures. A living organism, a hurricane, or a government would not persist even as a skeletal structure without a continual flow of externally supplied energy.

By working primarily with dissipative static entities engineers save themselves from having to build systems that are continually rebuilding and repairing themselves. But the price for that convenience is that these systems are not self-persistent. To date, we don't know how to build systems that persist on their own.

To the extent that we try, our approach tends to be backwards: design a dissipative static entity and then add features to it that might allow it to repair itself. That isn't how naturally occurring dynamic entities work. Most naturally occurring dynamic entities are built to be self-persistent from their core.

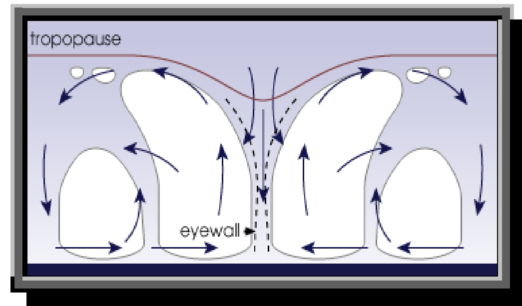


Figure 1. A hurricane
Original image from NASA

A hurricane, for example—which generates its own winds as a result of condensation—maintains its structure simply because of how it is organized. The same is true for a government and a living cell.

Much of what goes on in a cell is cell maintenance. Here's how Harold [7] describes this aspect of a cell's functioning.

> Is the cell as a whole a self-assembling structure? ... Would a mixture of cellular molecules, gently warmed in some buffer, reconstitute cells? Surely not, and it is worthwhile to spell out why not. One reason is [that] assembly is never fully autonomous, but involves [pre-existing] enzymes or regulatory molecules that link [developing elements] to the larger whole. But there are three more fundamental reasons … First, some cellular components are not fashioned by self-assembly, particularly the … cell wall which resembles a woven fabric and must be enlarged by cutting and splicing. Second, many membrane proteins are oriented with respect to the membrane and catalyze vectorial reactions; this vector is not specified in the primary amino acid sequence, but is supplied by the cell. Third, certain processes occur at particular times and places, most

> notably the formation of a septum at the time of division. Localization on the cellular place is not in the genes but in the larger system. Cells do assemble themselves, but in quite another sense of the word: they grow.

It would seem that if we are to build self-persistent systems, the first step is to learn how to build minimal dynamic entities that have as their core (a) a means for converting available raw materials into the substances needed to create and maintain their physical structures and (b) mechanisms for using those generated materials for self-persistence.

This is quite a trick. A cell manufactures its own building blocks and then uses those building blocks to keep itself in good repair. A hurricane doesn't manufacture anything, but it does use the raw materials at hand (water vapor, rain drops, air, etc.) to maintain its structure. Since it doesn't manufacture anything, and since the materials at hand are fairly insubstantial as building blocks, a hurricane's structural framework is itself insubstantial.

After we learn how to build dynamic entities that have the ability to convert available materials into structural building blocks, we can then move on to adding additional functionality. The example of *E. coli* offered two example of additional functionality. *E. coli* have the ability to move about in their environments. They also have a mechanism that biases their movement and makes it—and I'll use the word intentionally—purposeful. But these bits of functionality are added on to a basic self persistent cell structure. Nature didn't start with a requirement of locomotion up a nutrient gradient. It started with self-persistence.[10]

More generally, if one compares naturally occurring systems to those built through an engineering process, we find a significant difference. Human-built systems are often functionally fragile; their emergent properties don't hold up as robustly as we would like. Naturally occurring systems tend to be more robust, flexible, and adaptable. Why is that?

We tend to think of our system designs hierarchically. Systems designed and developed according to the standard systems engineering process are typically built using a top-down design methodology. This is quite different from how nature designs systems. When nature builds a system, existing components (or somewhat random variants of existing components) are put together with no performance or functionality goal in mind. (Nature doesn't have a mind.) The resulting system either survives in its environment or it fails to survive.

This approach doesn't necessarily make nature a brilliant designer. Some of nature's designs are wonderful—and some suck.[11] But significantly, nature never

---

[10] Of course, like Theseus' ship, discussed briefly below, most of the systems we build are embedded within dynamic social entities that provide for their maintenance—although we too rarely conceptualize our systems that broadly.

[11] Way and Silver's paper [10] in this issue points to "the panda's thumb, the placement of the windpipe in front of the esophagus (so that food can go down the wrong tube), traversal of the urethra through the prostate gland (so that if the prostate becomes inflamed and swells,

has to satisfy a requirement.[12] Systems engineers don't have that luxury. But is there anything we can learn from how nature develops designs that we can apply in our work?

The software approach to design in terms of objects, libraries, and levels of abstraction is a useful alternative. Service oriented architectures (SOA) generalize that approach to networks. This approach can be generalized to systems in the form of layered architectures. It is likely that these new approaches to system design will result in systems that are both more robust and more flexible. This issue is explored further in section 5.

2.4 Wikipedia as a case study of a complex system

Wikipedia[13]—the dynamic social entity consisting of both the software and the people who keep it going—is a very public example of what may not at first seem like a traditional complex system. Yet it has all the properties of a complex system. It relies on externally supplied energy for its persistence. It is multi-scalar. It includes (at a very broad level) the MediaWiki platform on which it runs, the Wikipedia conventions and Templates that give it some overall consistency, the users who contribute content, the users who maintain and regulate its content, and the users who make use of that content. Its overall structure is best understood as a layered architecture.

Like most other dynamic entities Wikipedia is both deployed and under development simultaneously. Like a biological organism, Wikipedia exists and functions in the world at the same time that it is undergoing development and self-repair. The MediaWiki software is open source software that is continually being modified. Similarly, Wikipedia content itself is also continually being extended at the same time that it is being used. Like most dynamic entities Wikipedia has very effective mechanisms for self-repair. Editors and other users continually monitor pages for damage, which they repair very quickly whenever it occurs.

3 Externalizing our thoughts

A useful way to think about the difference between systems designed to satisfy requirements and naturally occurring systems is that requirements-based systems typically result from an attempt to externalize our thoughts. We think, "I want a system that does this, this, and that—i.e., with these properties and behaviors."

Goals of this sort, no matter how dressed up and legitimized in terms of formal requirements are still nothing but ideas in our minds. Use of the somewhat deprecatory term *nothing but* is intentional. Ideas by their nature can exist only in the mind of someone who is thinking them. That's all an idea is and can ever be, a subjective experience in the mind of the idea's thinker. (See [4].)

Yet when our ideas involve systems, we want more than just pretty mental pictures. When want the systems we build to be material embodiments of our ideas. We want the ideas in our heads converted into physical reality. We want to externalize our ideas and to make them materially concrete. And we often

it becomes difficult to urinate)" as examples of bad natural design.

[12] Nor does nature have to work within budget and schedule constraints.

[13] See http://www.wikipedia.org.

succeed—spectacularly. Much of what we experience in our post-modern 21st century lives is the result of successfully externalized thought.

But let's consider what it means to externalize a thought. There is no *externalize button* on our foreheads which, when pressed, causes our ideas to materialize as physical objects. One cannot simply imagine something and expect a material embodiment of it to spring into existence. Furthermore, even when we do build something that reflects our ideas, it is impossible to create an external replica of a thought. Anything outside our heads is different from something inside our heads. Nothing outside our heads is an idea. The best we can ever do in externalizing a thought is to create something that we can understand as representing—or embodying—that thought.

3.1 Molding reality to resemble thoughts

Consider a word processing computer program. We design word processors to (appear to) operate in terms of characters, words, paragraphs, etc. Characters, words, and paragraphs are ideas. Word processors operate (when described at one reasonable level of abstraction) in terms of character codes, sequences of character codes bounded by white space character codes, and sequences of character codes bound together as what the word processor may internally refer to as a paragraph data structure.

What we do when we attempt to externalize an idea is to mold elements of physical reality into a form onto which we can project the idea we want to externalize. That's all we can ever do. We can never do more than mold existing reality. But even though we cannot incarnate our ideas as material reality, we can mold physical reality in such a way that it has—or at least appears to have—properties a lot like those of the ideas we want to externalize.

Thus there is always a tension between (a) building something out of real physical substance (even if that substance involves bits) and (b) externalizing one's thoughts about what one wants. This tension is easiest to describe with respect to software—but it is true of every constructive discipline, including systems engineering. When one writes software, one is writing instructions for how a computer is to perform. That's all one can ever do: tell a computer first to do this and then to do that. The this and that which the software tells the computer to do are the computer's primitive instructions. But what we want in the end is for the computer's *this-ing* and *that-ing* to produce a result that resembles some idea in our heads.

Thus in software (as in any engineering discipline) our creations always have two faces: (a) a reality-molding face whereby the software tells the computer what to do and (b) a thought externalizing face which represents our ideas about what we want the result of that molding process to mean. The eternal tension is to make these two faces come together in one artifact.

3.2 Thought externalization in computer science

Because software can be about an extraordinarily wide range of possible thoughts, computer science has been forced to face the reality-vs.-thought confrontation more directly than any other human endeavor. And possibly because software as text seems to be

the only example of an artifact that directly embodies both aspects of this tension, computer science has been relatively successful in finding ways to come to grips with this problem.

An enduring goal of computer science is to develop languages that have two important properties. (a) The language may be used to externalize thought. (b) Expressions in the language can act in the material world—that is, the language is executable. This is remarkably different from anything that has come before. Human beings have always used language to externalize thought. But to have an effect in the world, language has always depended on people. Words mean nothing unless someone understands them. Software acts without human intervention.

Computer science has developed languages in which we can both express our thoughts and control the operation of a computer. We invented so-called higher level programming languages (Fortran being one of the earliest) in which one could write something like mathematical expressions which the computer would evaluate. We invented declarative languages (Prolog is a good example) in which one could write statements in something like predicate calculus and have the computer find values that make those statements true. We combined Prolog and Fortran when we invented constraint programming (which has not been as widely appreciated as it deserves) in which one can write mathematical statements of constraints which the computer ensures are met.

We invented relational databases in which one can store information about entity-like elements—along with their attributes and their relationships to each other. We invented languages that allow one to query those databases more or less on the level of that conceptualization.

We invented object-oriented programming languages—which led naturally to agent-based and now service-oriented environments—in which one writes programs that consist of interacting entities.

At the application level, virtually every computer program—from a payroll program to a word processor to an photo processing program—embodies an ontology of the world to which that application applies.

To help us write programs we invented tools and frameworks that define meta-ontologies within which one can create a desired ontology.

We did all this by writing programs that tell computers first to execute this instruction and then to execute that instruction. The gap between the underlying computer and final application is often enormous. But that doesn't mean that we can forget about the computer. No matter what else it is, and no matter how well our programs express the thoughts in our heads, a program is nothing unless it tells a computer which instructions to execute and in what order. In the end, that's all a computer program is: a means to tell a computer what to do.

3.3 Computer science uses emergence to link ideas to reality

Computer Science has been called applied philosophy:[14] one can think about

[14] Fred Thompson, one of my early mentors, is now Emeritus Professor of Ap-

virtually anything as long as one can express those thoughts in a form that can be used to control the operation of a computer. I like to think of the computer as a reification machine: it turns symbolically expressed abstract thought into concrete action in the physical world.[15] As a reification machine, the computer's interface between thought and action is the computer program. When we write in a programming language we are expressing our thoughts in the programming language—to the extent allowed by the language. When a computer reads what we have written, it takes our writings as instructions about what operations to perform. We have developed programming languages that allow us to express something close enough to our thoughts that the resulting programs, when executed, can be identified with those thoughts.

The primary technique computer scientists use to build programming languages that allow us to externalize our thoughts is emergence. Recall that I defined emergence as a situation in which a property can be described independently of its implementation. That's exactly what a program specification is. Whenever we specify the desired behavior of a computer program independently of the means by which that behavior is implemented, we are asking for the creation of an emergent phenomenon.

Emergence of this sort has a long history. Both axiomatic semantics [11] and denotational semantics [12] offer approaches to providing declarative specifications for computer programs—which by intent are independent of the program's implementation. Hübler[16] has suggested that what is most important about a level of abstraction are the properties that it conserves. Axiomatic semantics is exactly that: a formulation of computer software in terms of conserved properties.

More generally, workers in the fields of functional and logic programming attempt to create programming languages (e.g., Haskell[17] and Prolog[18]) in which the programs one writes may be understood as a declarative statement of one's intentions rather than as instructions to a computer.[19] The ACM's series of conferences on declarative programming[20] explores the even more general question of what one can say about programs that is independent of the steps the program instructs the computer to take.

plied Philosophy and Computer Science at Cal Tech.

15 With virtual reality we complete the cycle: generating real physical signals with the intention of producing particular subjective experiences.

16 Personal communication.

17 See http://www.haskell.org/.

18 For example, http://www.swi-prolog.org/.

19 Practitioners in both paradigms find, however, that most "real" programs written in functional programming and logic programming languages generally cannot be understood in a fully declarative way. Virtually all real programs no matter what the language must be understood operationally. This is understandable since one can never escape the fact that a computer program is necessarily a means for instructing a computer to take certain actions that mediate between input and output.

20 The International Conferences on Principles and Practice of Declarative Programming (PDPP). See the website, http://pauillac.inria.fr/~fages/PPDP/.

The prototypical example of emergence in software is the application program interface (API). An API characterizes what software—generally the elements of a program library—will do when invoked in certain ways. A good API does not describe how the software will accomplish that result, just what the result will be. This is standard good practice about software design and specification.

What is not often mentioned—perhaps because we take it so much for granted—is that an API is generally explicated in terms of an ontology that may have nothing to do with the means by which the API is implemented. As I indicated above, virtually every computer application is intended to implement a conceptual model, i.e., some externalized thought. The same is true for APIs. In other words a software system creates an emergent ontological domain that can be accessed and manipulated in ways specified by its API.

One of the primary threads in the history of computer science is the development of increasingly powerful ways to create new levels of abstraction. By providing ourselves with the means to create new ontological domains—which can then be used to build other ontological domains, etc.—computer scientists have used the power of emergence to create models of an extraordinarily wide range of thoughts. Although these externalized thoughts are far removed from low-level computers operations, they are nonetheless still grounded by real computers executing one real physical instruction after another. In perhaps more familiar words, software development is both a top-down (thought externalization) and a bottom-up (emergence) endeavor.[21]

3.4 Thought externalization in systems engineering

Systems engineering is just beginning to focus on this issue. Model-based development, e.g., SysML, attempts to allow systems engineers to think in a language that both expresses thoughts and represents how to mold reality. But systems engineering is at a significant disadvantage. In computer science we write in languages that control real computers.[22] There are no systems engineering languages that generate real physical systems.

When software developers write a computer program, load it into a computer, and press the Start button, the computer *becomes* the program we have written. There is nothing comparable for systems engineers. We don't have a systems engineering language and a device into which descriptions written in that language can be loaded that will *become* the system the language is describing. The closest systems engineering can come to this dream is to write in a language that represents a model of a physical system. But models aren't reality. Programming languages succeed

[21] Of course the "real" "physical" instructions that ground computer software are themselves emergent phenomena built on top of still lower level phenomena. Computer science owes its existence to the ability of electrical engineers to create an emergent digital world—of bits and instructions that manipulate them—that we use as a platform on which to build our emergent creations.

[22] UML is an unfortunate step back from computer science's traditional loyalty to executable languages.

because they are grounded in the reality of an actual computer executing actual instructions. Models, in contrast, are always divorced from reality. One can't ever model all aspects of a system. So one chooses what one considers a system's most important aspects and models those. But that's always dangerous. See the discussion in section 7.1 about the difficulty of looking downwards for additional discussion of this point.

4 Multi-sided platforms

As discussed above, a level of abstraction encapsulates and embodies a specialized ontology (i.e., a conceptual model) of one sort or another. An extraordinarily important kind of level of abstraction is the multi-sided platform. Hagui characterizes a multi-sided platform (from the perspective of the platform as a business[23]) as one in which the platform provider must

> get two or more distinct groups of customers who value each other's participation on board the … platform in order to generate any economic value. … Examples are pervasive in today's economy and range from dating clubs ([the two sides are] men and women), financial exchanges [such as a stock market], real estate listings, online intermediaries like eBay (buyers and sellers), ad-supported media (ad sponsors and readers/viewers), computer operating systems (application developers and users), videogame consoles (game developers and geeks), shopping malls (retailers and consumers), digital media platforms (content providers and users), and many others.

When considered more generally, i.e., not necessarily as an business, a multi-sided platform is a level of abstraction that provides a means, mechanism, or set of conventions for structuring and enabling interaction among parties—especially parties that expect to benefit from the interaction. As Hagiu indicated, men and women in a dating club are able to interact because they belong to the same club. The same is true of merchants and shoppers in a mall and buyers and sellers on eBay.

In general, a multi-sided platform results from the factoring out of an aspect of an interaction. In malls and dating clubs, what's factored out is (a) the process of finding the other party and (b) the formalism of making contact. By factoring out an aspect of an interaction and providing it more efficiently, the platform makes the interaction more efficient for the parties and at the same time generally makes money for itself.

Browsers are multi-sided platforms—mediating between web sites and web surfers. Other examples include online interest groups (such as YahooGroups) and bulletin boards. It is common wisdom that mailing lists and bulletin boards have created communities—and hence interactions among members of those communities—that never would have come into existence otherwise. The same is true of multi-person environments such as Second Life. Systems such as MySpace and FaceBook provide another sort of community building platform. The interactions that occur on these platforms would almost certainly never had occurred were it not for the existence of these platforms.

4.1 Platform ownership and control

Once a commercial platform becomes established, conflicts may arise when

[23] Interview with Andre Hagiu, *Working Knowledge*, Harvard Business School, March 13, 2006. http://hbswk.hbs.edu/item/5237.html.

the interests of the platform owner differ from those of the platform users. Pressure may develop among platform users to de-commercialize the platform and to move its governance out of the commercial realm and to bring it under the control of the users.

Our regulated utilities—such as power and telephone services—illustrate a successful combination of user governance and private ownership. Other platforms, e.g., our road and highway system, are owned and operated directly by the government. We find these platforms so essential that we want to ensure that the interest of the platform users take precedence over the interest of the platform providers.

Platforms such as these, along with the rest of our community-wide platforms (such as our transportation, package delivery, and mail systems[24] and others), define what we refer to generically as a community's infrastructure.

Organizations that are able to establish a multi-sided platform as a widely used standard (explicit or *de facto*) are often able to profit from it. Familiar examples are the Microsoft Windows operating system and eBay. This has led to the notion of what has been referred to as a network effect, namely that the value of a network increases more than linearly with an increase in the size of the network.[25] The literature on network effects seems not to identify platforms as the source of the value: networks hogs the spotlight. Nonetheless, it is the establishment and ownership of platforms that has economic value. Thus platforms become very important to commercial organizations, who will fight to establish the dominance of their platform in a certain realm.

There are (at least) three countervailing forces to private/commercial control of platforms. The first, as mentioned above, is government regulation and in some cases control.

The second is the adoption of neutral standards, i.e., standards that are neither controlled by nor tailored to the interests of any particular vendor. When a vendor-neutral standard is defined for a platform, the platform functionality is defined independently of any specific implementation of that functionality. This is to the benefit of platform users because vendors must then compete to provide better implementations of a platform with a well-defined specification. Thus the most ephemeral of multi-sided platforms is the standard. Users of systems/components that adhere to a standard are able to interact with each other only because they both conform to the standard.

The third force that mitigates the commercialization of platforms is open source software. Many commercial software products depend on platforms for their operation. The most widespread case is the dependence of software application programs on operating sys-

[24] The platform that facilitates the electronic version of such interchanges is the collection of Internet email standards. See the Internet Email Consortium website (http://www.imc.org/mail-standards.html) for a list of email standards. We discuss below the importance of standards as platforms.

[25] See [13] for an argument that the rate of growth is typically *n log(n)* rather than n^2.

tems. Consider the position of the developer of such a software application product. He is essentially at the mercy of the platform owner. Should the platform owner decide to enter the same market, that owner has an enormous advantage. The most widely known example is the way in which Microsoft destroyed the Netscape browser. No company wants to be that vulnerable. The developer of a software application will be motivated to support alternative platforms for his product. The most attractive alternative platform is one that is neither controlled by a commercial entity nor regulated by the government. Hence it is not at all surprising that many commercial companies provide significant support to the development of open source platforms.

In a draft article, Iansiti and Richards [14] analyze open source systems. They find that one can group open source software into two categories: the "money driven cluster," which receives 99% of corporate funding and the "community driven cluster," which receives very little corporate funding.

The big four in the money driven cluster are Linux, Firefox, OpenOffice, and MySQL. All four are platforms that are central to how computers are used. The first three compete with platforms that are owned and controlled by a single for-profit company. No corporation wants to see the platforms on which its products depend subject to the profit calculations of some other commercial entity. It's no wonder that corporations are willing to spend money to strengthen publicly and openly controlled alternative platforms.

4.2 Platforms governance

The preceding were all examples of specialized platforms that support important but limited kinds of interaction. The more significant community-level multi-sided platforms are those that structure economic interaction itself. The two most important are (a) the monetary and banking system and (b) the laws of commerce.

By factoring out the economic notion of value, the monetary system is the multi-sided platform that allows economic value to be abstracted, stored, exchanged, and transformed.

Similarly, the laws of commerce (and its associated judicial and police system) provides the multi-sided platform that enables economic agreements to be made and transactions to occur—both with an increased level of confidence and security. This latter multi-sided platform has factored out what would otherwise be the need on the part of the participants to establish their own enforcement mechanisms.

As a society we clearly believe that these platforms should be controlled by the government and not by commercial organizations.

When platforms are considered a common resource, the question of governance must be addressed. Elinor Ostrom has pioneered the study of governance for common resources. In [15] she developed a framework of eight design principles.

1. Well defined boundaries that clearly identify both the resources to be shared and the user community among whom they are being shared.

2. Proportionality between costs and benefits in the sharing mechanisms.
3. Mechanisms that allow for the participation of the group members in the decision making process.
4. Effective monitoring by accountable monitors.
5. Graduated sanctions for participants who do not respect the rules.
6. Conflict-resolution mechanisms which are inexpensive and easy to access.
7. Recognition by higher level authorities of the right of the group to make at its own rules.
8. In case of large collections of common resources, nested layers of governance so that rule making and rule enforcement can occur at the appropriate level.

Research in the governance of common resources is ongoing. (See, for example, [16].) It seems likely that this work will produce approaches that will help solve the governance problem.

4.3 Natural language as a platform

An even more pervasive platform is language itself. Our natural languages provide us means to interact. Language as a platform is different from most of the other platforms we have discussed in that it is implemented by each of us individually.

Natural language is also like a standard in a number of ways. For one thing, no one entity provides an implementation. Secondly we have dictionaries and grammar books and "standard English" reference implantations. But clearly natural language is not a regulated standard whose precepts change only with the approval of the standardization committee—other, perhaps than French. Natural language is more like an open source system in that its evolution depends on a large community of contributors.

As indicated above, the natural language platform is implemented by each of us individually. We each spend the first six years of our lives learning how to do that. Even though we are certainly not completely consistent about our private implementations, it is really quite amazing that we all do it as consistently as we do and that it is as successful a platform as it is.

4.4 Multi-sided platforms and systems engineering

Like levels of abstraction in general, multi-sided platforms should be central to systems engineering. Unfortunately, they tend not to be. In systems engineering we tend to focus on pair-wise communication among systems components. Often, that pair-wise communication is hierarchical; sometimes it is horizontal. But in either case, we don't think about factoring out any of the functionality built into that communication. When working on interaction we often write what are called interface control documents (ICDs). But like the interfaces they specify, these documents are defined on a pair-wise basis.

Recently, the notion of net-centric operation has gained significant traction. That notion is built on the idea of the network as a platform. This, of course, is

a very powerful idea—one that was inspired by the Internet. But the network as a platform is just one example. In software platforms abound. It's time for systems engineering to begin to conceptualize systems in terms of (a) levels of abstraction in general and (b) platforms in particular. Doing so will result in systems that are designed more in terms of layered architectures and less in terms of functional decomposition. The next section continues this theme.

5 Service-oriented design

Much of what succeeds in nature consists of processes that build on other processes. Food web analysis illustrates how species depend on other species. Ecologies are built on seasonal cycles and resources flows—energy from the sun being the most basic but ocean and river currents being other examples. A species, a seasonal cycle, and a resource flow can all be understood as emergent phenomena. In other words, nature builds new emergent phenomena on existing emergent phenomena.

When this happens in an ecological system, we call it *succession*[26]—a territory proceeds though a series of relatively stable stages.[27] At each relatively stable stage, the species that populate that stage depend on each other and on the other aspects of the environment. Progression occurs either because something disturbs the *status quo* and destroys some of the structures on which some of the participants depend or because there is an inefficiency in the system that can be exploited by some new mechanism.

This is pretty much the same picture one sees in a market-based economy. A collection of products, services, and community-supplied infrastructures (such as a monetary system, a postal system, a judicial system, etc.) develops into an ecology of mutual dependencies. Such a system remains stable until either a disturbance destroys something on which part of the system depends or a new way is found to use some of the available energy.

Natural ecologies and market economies are both examples of what I call innovative environments—which are discussed in section 8. This section focuses on how such environments work and how the principles underlying how they work may be applied to system design. The fundamental principle is that new things are built on top of existing things. Because we have a well-developed international transportation system, for example, we can produce products in one location and move them to other locations to be consumed. One doesn't have to develop a transportation system from scratch in order to establish a production facility in a low-wage part of the world.

5.1 Products and services evolve

Even though most marketed products and services tend to originate as externalized thought, well-managed companies are always looking for new applications of their products—even applications that have little to do with the origi-

[26] See, for example, http://www.mansfield.ohio-state.edu/~sabedon/campbl53.htm.

[27] This resembles what on an evolutionary scale we refer to as punctuated equilibrium. The difference is that in succession outside species replace existing species in a habitat. But the outside species are not generally created as new species.

nally conceived market. In other words products and services evolve to fit their environments.[28]

Products and services that survive over the long term are not stuck attempting forever to implement the original vision of what they were intended to be. A product or service may have been born of externalized thought, but the original externalized thought is not considered a constraint on the evolution of the product or service. It's the environment that determines how a product or service will evolve.

In order for a product or service to evolve, its design must support change. If a system is designed in such a way that modification of that design is not feasible, it will die. Thus any system that is expected to survive over the long term must have evolvability as a primary design consideration.

Unfortunately, we tend not to build systems this way. Customers often want a set of functional requirements satisfied as inexpensively as possible. Normally that entails sacrificing design flexibility and evolvability for a rigid focus on specific functionality.

This is one reason why it is a bad idea ever to buy a major system. If the system developer has a financial interest in seeing the system flourish over the long term, the developer will (presumably) design it to allow it to evolve. In contrast, a customer, who has no idea about these sorts of things, cannot require evolvability as a system property. (We don't know how to do that in any case.) Even if the customer could require evolvability, he or she would probably not be in a position to exploit it. After all it is the developer who is on the lookout for new uses of the system, not the individual customer.

5.2 Products and services are built on top of an established base of other products and services

The second and perhaps more significant lesson to be learned from the web-of-interrelationships perspective is that when building something new it's a good idea (actually more than just a good idea) to make use of existing products and services. This is quite different from how most of our systems are designed.

As discussed above, we tend to build systems hierarchically. We formulate a top-level design that meets top level requirements and then determine what components we need to implement it. We then decide how to build the components in terms of sub-components, etc. This approach fails to take advantage of existing products and services—except when we use standard parts, which we too rarely.

A hierarchical design approach has (at least) two disadvantages. First, it tends to produce what have been called stove-piped systems—systems that may work successfully on their own but that are very difficult to use in conjunction with other systems. That such a consequence will occur is quite understandable. When a system is built from the top-down without regard to what else exists, it is likely to be incompatible with other

[28] We have all encountered the now familiar version progression among software products. Version 5.0 is frequently quite different from version 1.0. It might even serve a significantly different customer base.

systems—except for those with which it is explicitly designed to interact.

Second, the internal design of such systems tend to be rigid in the same way. Just as a hierarchically designed system isolates itself from other systems, the system components of a hierarchically designed system isolate themselves from each other. Hierarchical design results in stove-piping both inside and out.

The alternative is to take advantage of what exists and build on top of it. In software there are now innumerable tools, frameworks, components, and libraries (both open source and commercial) that serve as the basis for further development.

The prototypical example of building on top of existing products and services is a service-oriented architecture (SOA).[29] Service oriented architecture is a nice example because it illustrates how systems can be built on top of existing elements at both the system-to-system and internal design levels.

Through an SOA, systems can provide services for other systems. Similarly system components can provide services for other system components through an SOA. In both cases, one is building on two foundations: (a) the network itself as a service (e.g., a level of abstraction) that all elements that reside on it use and (b) the design principle whereby elements provides service for each other. It's a positive development that service-oriented and net-centric architectures are becoming desirable attributes within the systems engineering world.

It's important to remember that SOA and net-centricity are examples not principles. The principles are (a) build on top of existing capabilities and (b) conceptualize whatever one builds as a service that others will use, not as an end in itself.

5.3 Dynamic entities need energy to persist

This principle captures one difference between most systems engineered systems and systems that appear either in nature or in a market-based economy. Most of the systems with which we are concerned are dynamic entities. They generally do something as a result of energy flows. But even static objects, such as a bridge, require maintenance. The real system is not just the static bridge. The real system is the bridge along with the maintenance process that keeps the bridge in good repair.[30] When understood from that broader perspective, it's clear that most of the systems

[29] This is a design fad that has staying power. It's useful to think of our entire economic system as a service oriented architecture: every economic transaction is essentially a service transaction. The SOA nature of our economic system is one of the reasons it is both strong and agile.

[30] This perspective explains the Theseus' ship paradox. Is a ship that has been maintained in port so long that all its parts have been replaced "the same ship" as the original? The answer is that the ship maintenance process is the same persistent entity (even if it involves numerous people cycling though it—a property of entities). The physical ship is just a component of that social entity in much the same way as our (replaceable) cells are a component of ourselves as entities and the officers of a corporation or a government are component of those continuing dynamic entities.

that systems engineers build are dynamic entities.

Dynamic entities persist only as long as the energy that flows through them continues to flow. For a business, which is also a dynamic entity, money is a proxy for energy. A business exists only while the money flowing into it is at least as large as the money flowing out of it.

Unfortunately, it very rare that we ask ourselves about the energy flows needed for the persistence of the systems that we as system engineers are asked to build. Long term energy flow considerations (i.e., funding) should be fundamental to any system development project. But it generally isn't. Because it isn't we don't think about systems in terms of what it would take to make them self-persistent.

This is not to say that every system must be profitable in the traditional sense of profitable. Many of our system, e.g., our judicial and monetary systems are both so central to the functioning of our society and so ill suited to be funded by their direct customers that we properly treat them as a commons. But whether the system we are building is expected to be a commons or self-sustaining, we must understand from the start how the energy flow required to sustain it will be provided.

In business the answer to this sort of question would be recorded in a business plan. In systems engineering we don't have a name for where we record answers to these questions because we rarely ask them.

6 Feasibility ranges

Emergence occurs within feasibility ranges. A visible and tragic illustration of this is the Challenger disaster in which the O-rings lost their (emergent) sealant property because the temperature was too low.

Since there are *always* feasibility ranges for emergent properties we should make it standard practice to identify and determine the feasibility ranges of each emergent property we expect our system and system components to display. For each emergent property we should explain why its feasibility range won't be violated—and what happens if it is. Had this been done for the Challenger, we would not have lost our astronauts.

Computer science doesn't worry about feasibility ranges. A level of abstraction remains a level of abstraction as long as the software that implements it continues to run. In contrast, an illustration of the difficulty engineering has with ontologically independent constructs is that engineering designs are buttressed by safety factors; software uses assertions.

Every physical implementation of a level of abstraction does indeed have feasibility conditions. Safety factors should be understood as ways to ensure that those feasibility conditions are met rather than as an insurance policy against improbable events.

Although we tend not to think about them this way levels of abstraction have feasibility range concerns. But these aren't ontological. They typically involve such issues as data rates, access rates (for quality of service issues), data storage demands, assumed data (and other input) ranges and limits, computational demands, accuracy assumptions, and precision needs. It is these sorts of issues that may cause the software that implements a level of abstraction to cease to run. In software these concerns are often lumped together as per-

formance (as distinguished from functionality) issues.

7 Modeling and Simulation

Modeling and simulation provide powerful tools for systems engineering. But they have significant fundamental limitations.

7.1 For want of a nail …

An important characteristic of complex systems is that they are multi-scalar. Every system that exhibits emergence exists on at least two scales, the scale at which the emergent property appears and the scale at which the emergent property is implemented. Often there are many more scales. This is especially true when emergence is built upon emergence, as it should be. The poem telling the story of how a missing horseshoe nail led to the loss of a kingdom illustrates the potential significance of multi-scale phenomena.

Much of the work in systems engineering relies on the results of simulations. We build models of possible system designs, and we run them, watching what happens as we vary the parameters. But even with our advanced modeling and simulation capabilities it would be virtually impossible for us to model all the nails in all the horseshoes on all the feet of all the horses ridden by all the men in King Richard's army. Certainly we can't do anything remotely like that if we were to model today's massively larger systems.

The fundamental problem is that simulations are software, and software is built on a foundation of bits. Bits prevent computer science from working with the full richness of nature. Every software model has a fixed bottom level—making it impossible to explore phenomena that require dynamically varying lower levels. This means that every simulation has a defined floor. For some problems we either do not know what the correct floor is, or when we do know, simulation at that level is not feasible.

A good example is a biological arms race. Imagine a plant growing bark to protect itself from an insect. The insect may then develop a way to bore through bark. The plant may develop a toxin—for which the insect develops an antitoxin. There are no software models in which evolutionary creativity of this richness occurs. To build software models of such phenomena would require that the model's bottom level include all potentially relevant phenomena. But to do so is far beyond our currently available computational means.

This is an example of a situation in which engineering—as a discipline that works with physical materials—has a profound advantage over computer science—a discipline that builds its models on a foundation of bits. Engineering (like science) has no floor. It is both cursed and blessed by its attachment to physicality. It is cursed because one can never be sure of the ground on which one stands—raw nature does not provide a stable base. It is blessed because one can decide for each issue how deeply to dig for useable bedrock.

Systems engineering depends profoundly on modeling and simulation. But we are unable to write simulations with dynamically varying bottom levels. So we are not able to model our intended systems at the many scales at which we must investigate them. What are we to do? This is a major research issue and one for which I have no answer. In [1] I called this the difficulty of looking down-

ward. The first step is to recognize that we have a serious problem.

7.2 For want of imagination …

Imagine (unrealistically) that we were able to simulate our air transportation system and everything else relevant to how airplanes are used and maintained. Would that capability have helped prevent 9/11? My answer is "No." The problem is that we have no idea how to build simulations that can identify emergent phenomena that occur within themselves.

Earlier we urged that new systems be built on top of existing capabilities. That's exactly what the 9/11 terrorists did. They used the capability provided by the airlines of carrying and delivering large amounts of explosive material. All the terrorists had to do was to take over the planes at the critical times—a brilliant example of using an existing capability to produce a new capability.

We know how to write simulations in which emergence occurs. Any agent-based model is capable of exhibiting emergence. But we don't know how to write simulations that will recognize that emergence has occurred and issue a report about it. In [1] I called this the difficulty of looking upward.

This is a nice illustration of the difficulty of recognizing emergence. Let's return to our fully accurate simulation of our air transportation system. Suppose it included (a) airplanes accidentally crashing into buildings and (b) air hijackings. Perhaps in such a virtual world, a hijacked airplane accidentally crashed into a building, destroying it. Even so, it is difficult to imagine that the simulation would be able to predict the intentional hijacking of an airplane for the purpose of crashing it into a building.

A system might be able to make such a prediction if it were (a) programmed to look for instances of significant destruction (and categorize the accidental crash as such an instance), (b) able to conclude that the accidental crash could also be caused intentionally, and (c) aware of the possibility of suicide actions. The ability to make those observations, draw those inferences, and predict a 9/11 type of attack goes significantly beyond what one would normally find in an air transportation simulation—and probably far beyond any system yet developed.

7.3 Externalizing the idea of an idea

Let's explore what it might take to build a system that could generate the idea of attacking the World Trade Center with hijacked airplanes.

I know of no system that is capable of generating new ideas. For all the advances we have made in externalizing thought, we do not yet know how to externalize the idea of an idea in anything like its full richness.

In saying this I'm assuming (a) that ideas themselves exist only in the minds of their thinkers, i.e., only as subjective experience and (b) that computers don't have subjective experience. An immediate consequence of this is that computers don't have ideas. So the best we can possibly hope for with current technology (and with any technology that we can currently envision) is that we might be able (a) to externalize the process of generating new ideas and (b) to execute that externalized process as software.

To do this we would have to find a way (a) to represent the idea of an idea and (b) to generate new ones artificially. To use the example from the previous section, we would have to develop a computer systems that could come up with an idea such as, “let’s use a commercial airplane as a weapon to be wielded by a suicide hijack crew.” To accomplish this, four technologies would have to be brought together: knowledge representation, ontology, modeling and simulation, and exploratory search.

Knowledge representation. The question of how to represent ideas in general has long been a subject of study within computer science. Brachman [17] provides a survey of the current the state of the art of knowledge representation. Most of the material in that book has been well known for quite some time; it’s surprising how stale it seems. As well as we have done in building computer systems that externalize particular realms of thought, we have done surprisingly poorly at externalizing thinking as such. There is nothing in Brachman (or elsewhere) of which I am aware that suggests that we have any new ideas about how to write a computer program that can represent ideas in general.

Ontology. If knowledge representation is something like the structure of a database for representing ideas, we need a way to populate such a knowledge representation database. That’s the subject matter of ontology. Ontology, the study of what the world is composed of and how those pieces fit together, is as old as philosophy. What is needed is an ontology that is (a) rich enough to provide the raw material that when combined will result in new ideas, (b) flexible enough to be able to incorporate new ideas as they are generated, and (c) detailed and clear enough to allow that knowledge to be make operational.

The two most active streams of research in this area are the various Semantic Web projects and Cyc[31]. These show promise, but none seem anywhere near mature enough to be put to work in generating new ideas.

Ultimately, no ontology will be sufficient unless it includes all of science. There is no current hope for building a computer system that incorporates all of science—and not just science in the form of scientific papers but science in the form of executable models.

Modeling and simulation. Once one has an ontology captured by some knowledge representation formalism, to make use of it, one needs more than static information.[32] But executing generic ontologies is far beyond our current state-of-the-art. One might identify as a fundamental goal of the computational treatment of complex systems the ability to model (and simulate) multi-level ontologies.

To take a simple example, we are not currently able to simulate a multi-level model of a diamond—a simple static entity. On one level the simulation would illustrate how the diamond is held together as a lattice by atomic forces. On a second level the simulation would illustrate how the diamond as a whole

[31] See http://www.w3.org/2001/sw/ and http://cyc.com/ respectively.

[32] Cyc contains lots of static information about its subject matters. But it has no way to execute operations that the elements of its database are able to perform. Perhaps for that reason Cyc seems particularly weak in its catalog of verbs.

moves through space. A third and even more difficult combination of these levels would show how a diamond can be used to cut glass. Since a diamond is a relatively simple static entity, imagine how far we are from being able to build adequate multi-level simulations that involve dynamic entities. Or consider again the evolutionary arms race discussed earlier. We simply are not able to represent the multiple levels of information required to model and simulate reality with anything near that degree of richness.

Exploratory search. Exploratory search—e.g., genetic algorithms and genetic programming—is needed to allow a system to explore various possibilities and come up with ones that achieve its objectives. Recall the discussion of purpose in section 2.2. In order to do exploratory search—i.e., to model systems that are capable of acting with intention—one must be able to build entities that include models of the world within which they are situated and then use those models to guide their actions.

Work in all four of these areas is ongoing, but I am not aware of any current projects that attempt to integrate these areas in a system that would be powerful enough to generate an idea such as the one that resulted in the destruction of the World Trade Center. To do so would be to achieve one of the original grand dreams of artificial intelligence. We are still far from that goal.

8 Innovative environments

Emergence is associated with new levels of abstraction, i.e., innovation. Four environments that are justifiably celebrated for an outpouring of emergent phenomena are the Internet, the market-oriented economic system, our system of scientific research, and biological evolution. Although quite diverse, all four have served as incubator for an ever-broadening flow of innovative products, services, and other elements.[33] This section asks what, if anything, these environments have in common and whether other organizations and environments may make themselves similarly innovative.

8.1 Evolution in a nutshell

Fundamental to all innovative environments is that like dynamic entities they depend on externally supplied energy. The impetus driving most innovation is a rivalry—generally implicit but sometimes explicit—to find more effective ways to access that energy.[34]

---

[33] Transformation in the Defense Department—including net-centric operations, and service oriented architectures—has been motivated at least in part by a desire to produce similar benefits within the DoD.

[34] Darwin [18] thought that evolution was driven by Malthusian competition.

> The struggle for existence inevitably follows from the high geometrical ratio of increase which is common to all organic beings. … More individuals are born than can possibly survive. … As the individuals of the same species come in all respects into the closest competition with each other, the struggle will generally be most severe between them; it will be almost equally severe between the varieties of the same species, and next in severity between the species of the same genus. … The slightest advantage in one being, at any age or during any season, over those with which it comes into competition, or better adaptation in however slight a degree to the sur-

With that in mind, let's begin with a brief review of biological evolution. Evolution by natural selection depends on (a) the possibility of heritable variation and (b) the effect of an entity's environment on the entity's ability to survive and reproduce. The more successful an entity is at surviving and reproducing, the more likely it is that the features that define the entity's relationship to its environment will be passed on to the entity's offspring. Feature variations that enable their possessors to survive and reproduce more effectively will propagate. Since the environment "selects" the features to be preserved—by making it harder or easier for entities with those features to survive and reproduce—Darwin referred to this process as evolution by natural (i.e., environmental rather than artificial) selection.[35, 36]

In *Darwin's Dangerous Idea* [19], Dennett argues that the evolutionary mechanism—random variation and environmental selection—is the source not only of biological creativity but of all creativity, including our own. When we have a creative idea (according to Dennett), it is the result of a random mutation and recombination of ideas in our mind along with our mind's selection of one or more of the resulting ideas because it has some property we find valuable. In other words, we don't design and fabricate new ideas to meet specific requirements; random variations and recombinations of existing ideas (some-

---

> rounding physical conditions, will turn the balance.

I disagree. In my view evolution is primarily a rivalry for better ways to access energy. Malthusian competition is but one element of that more general rivalry.

[35] In *The Origin of Species* [18] explained evolution as a form of breeding.

> [M]an can and does select the variations given to him by nature, and thus accumulate them in any desired manner. He thus adapts animals and plants for his own benefit or pleasure. He may do this methodically, or he may do it unconsciously by preserving the individuals most useful to him at the time, without any thought of altering the breed. It is certain that he can largely influence the character of a breed by selecting, in each successive generation, individual differences so slight as to be quite inappreciable by an uneducated eye. This process of selection has been the great agency in the production of the most distinct and useful domestic breeds. …
>
> There is no obvious reason why the principles which have acted so efficiently under domestication should not have acted under nature. … Why … should nature fail in selecting variations useful, under changing conditions of life, to her living products? … I can see no limit to this power, in slowly and beautifully adapting each form to the most complex relations of life. The theory of natural selection, even if we looked no further than this, seems to me to be in itself probable.

[36] Recall that computability theory applies to a level of abstraction implemented in the Game of Life even though it is independent of the Game of Life rules. Like computability theory evolution also is similarly autonomous. It neither depends on nor is derived from lower level laws of nature. Although reproduction and feature transmission are implemented by DNA, Charles Darwin and Russell Wallace didn't know about DNA. They didn't have to. Evolution occurs in any environment that includes heritable variation and environmentally influenced survival and reproduction.

how) come to us, and we select the ones that work.

It's important to understand how the success of a variant manifests in an evolutionary environment. For the most part, it's not the triumph of one partisan over another. Here's one of the most widely cited examples.[37]

> Originally, the vast majority of peppered moths in England had light coloration, which effectively camouflaged them from predators since they blended into the light-colored trees and lichens which they rested upon. Due to widespread pollution during the Industrial Revolution, many of the lichens died out, and the trees which peppered moths rested on became blackened by soot, causing most of the light-colored moths to die off due to predation. At the same time, dark-colored moths flourished because of their ability to hide on the darkened trees. Since then, with improved environmental standards, light-colored peppered moths have again become common.

This example illustrates the evolution of pepper moths from light-colored to dark and back. In this, as in all examples of evolution, a successful variant must exist before it can be favored by an environment. (Some dark-colored moths had to exist before they could flourish. Some light-colored moths had to exist before they could make a comeback.) Variants must exist or be created to get the evolutionary ball rolling.

As the evolutionary process proceeds, less favored variants don't change to adopt the properties of the more favored variant. The less favored variants die out, and the more favored variants reproduce more successfully. The light-colored moths didn't "notice" how well the dark-colored moths were doing and "decide to become dark." Nor was there a debate among the moths about the best survival strategy. Nor did the dark moths force the light moths to become dark. The light-colored moths simply died back, and the dark-colored moths reproduced more successfully.[38] In sporting terms a contest between competing variations is more like a high jump competition than a boxing match. It's a matter of *outdoing* a rival rather than *subduing* him or her.

Evolution generally involves what has been called *creative destruction*. As successful new variants enter the environment, older variants die out. This is not to say that evolutionary environments are rife with violence. As the moths example illustrates, there were no epic battles between light- and dark-colored moths. Creative destruction is often a strikingly peaceful process. Nonetheless, creative destruction generally involves a reduction in resources for some elements of an environment as others get more.

8.2 Application to organizations

If Dennett is correct—and I believe he is—the only way to ensure innovation in an organization is for the organization (a) to encourage the prolific creation and

[37] This explanation is adapted from Wikipedia: http://en.wikipedia.org/wiki/Peppered_moth_evolution. Accessed August 7, 2007.

[38] In the free market economic system, "battles" between competing products are similarly circumscribed. It's ok for Coke and Pepsi to compete for the same customers. It's not ok for either to use its economic power to attempt to force retailers not to carry the competing product.

trial of new ideas and (b) to allow new ideas to flourish or wither based on how well they do. I call these stages of innovation (a) *creation and trial* and (b) *reaping the rewards*.

This sounds pretty straightforward. It might seem that it shouldn't be too difficult for organizations to develop processes that support both stages of innovation. But serious obstacles arise. Let's look at what happens to a new idea (a) in biological evolution, (b) when tried out by a private entrepreneur, and (c) in a bureaucratic organization. For concreteness let's imagine that the new idea is something like a networking website like *MySpace* or *FaceBook* but before these sites had become established web phenomena. (See table 1.)

Creation and trial. Most organizations claim to favor innovation. They may say that the door is always open for new ideas. It is likely that even in bureaucratic organizations, people will remain creative. So it is likely that there will always be a supply of new ideas. The first real difficulty is the step from idea to trial.

- **Initial funding.** It generally requires capital—i.e., time and equipment—to develop a new system. This is capitalism in the small but at its purest. It takes capital to create something new.
  - *Biological evolution.* Capital is always available. The natural course of reproduction creates an endless stream of experiments. We don't think of this as capital, but it is. Every reproductive act is a capital investment, and nature continually invests a portion of its capital in experimentation.
  - *Entrepreneur.* If an individual doesn't have (or have access to) the capital or the equivalent, he or she is out of luck. One reason the Internet has been the source of so much creativity is that it typically takes only a relatively small amount of capital to try out an idea. Many people have the skills and the equipment to build new systems on their own or with a few friends.
  - *Organization.* Does the organization provide funding to try out new ideas? If so what procedures are required to get that support? Must one write a proposal? Is there competition among large numbers of proposals? Are the prospects of getting funded so remote that many people give up before they start? How are proposals evaluated? Inevitably, research funding follows fads. Some years some topics are considered hot; other years those same topics are dismissed as unworthy. If a proposal is required must it include a cost-benefit analysis? That's generally not a good idea. After all, if the proposed work is truly new, it is unlikely that one will be able to estimate its expected value. How, for example, would one estimate in advance the possible value of a networking site like *MySpace*—either for intra-organizational networking or as a service to be offered to the public? With many new ideas, all one can say is that one suspects it may be useful, but it's very difficult to quantify anticipated results.[39]

[39] Interestingly, Google models its support of new ideas on nature—although much more lavishly. There are no proposals and no funding requests. Google has all its employees spend 20% of their work hours on creation. Presumably some people will work on their own ideas; others will get together and work on a joint

- **The prospect of failure.** To what extent does the prospect of failure limit experimentation?
  - *Biological evolution.* Failed natural experiments generally result in death—and most experiments in nature fail. If the prospective subject of one of natures experiments were aware of the odds and had the option not to participate, it is unlikely that mutations would ever occur. But that's not how nature works. So even though nature is particularly hard on failed experiments, experimentation in nature abounds.
  - *Entrepreneur.* Failure is never fun. But when one fails in a private endeavor, the failure has definite boundaries. An individual may have invested some of his or her time and money. He or she may feel embarrassed in front of acquaintances. But for the most part, one knows in advance what one is risking. Failures of this sort may hurt, but they tend not to cast too long a shadow. Experimentation for private individuals certainly requires a certain type of personality—or desperation. Fortunately, there are enough such individuals that experimentation abounds among individuals.
  - *Organization.* How discouraging is the prospect of failure for projects run under a corporate umbrella? Organizations may say that they expect a certain percentage of experimental projects to fail. But are the people who run the failed projects rewarded for their efforts? Does anyone really want a failed project in his or her personnel file? Unlikely—especially if one works for an organization that expects "100% mission success," where "failure is not an option," and that values "can-do" people who can "execute" and "get the job done." It may take a lot of spin doctoring to rescue one's reputation after a failed experiment.
- **Approvals.** Once a new idea receives a preliminary go-ahead, how difficult it is actually to try it out? Is there an approval gauntlet to be run?
  - *Biological evolution.* There is no approval process in nature. Nature just does it.
  - *Entrepreneur.* For an individual, once the hurdle of initial development is surmounted, there are few additional problems. One makes the website available and hopes it succeeds.
  - *Organization.* Normally it requires approval to install something on an organizational server. The new pages may have to be written in a certain programming language and conform to certain technical and documentation standards. What about the content? Is approval needed for the content? Will there be issues regarding organizational policies? Most organizations are concerned enough about their public images that web pages will be required to be approved by the organization's public communication vetting system. The organization may also be concerned about having its name associated with some strange new idea. Even if the organization's name is not on the new website, the organization may be concerned that the website can be traced back to the corporation. There may be visual

---

idea. Since most Google employees are technically capable, this solves most of the capitalism-in-the-small problem.

and graphics style guidelines that make it difficult or even impossible for the new idea to be tried out as originally conceived. What about intellectual property issues such as organizational proprietary and copyright? What about other legal issues? Will the organization require that it's lawyers review the new website before allowing the project to go public? If the new website is intended for internal use, might the organization be concerned about how people use it and what legal liabilities certain forms of misuse creates. What if people insult each other? What if someone posts something with sexist or racist overtones? How will pictures or videos posted to the site be monitored and approved? The organization may require that a formal review process be instituted for each posted item, thereby killing the immediacy and spontaneity upon which such sites depend.

Reaping rewards. What happens once an experiment is tried and succeeds? Can the experiment reap the rewards of its success? In environments in which resources are allocated bottom-up, it can. In top-down environments, it's a lot harder.

- *Biological evolution.* Success in nature is defined by whether or not the experimental population grows and flourishes. There is no distance between the experimental result and the reaping of rewards. They are one and the same thing. One should imagine the landscape as consisting of resources (primarily energy) in various forms. Success typically means either finding a better way to acquire known resources or finding an under-exploited resource niche. Access to resources within such a framework is bottom up. There is no top-down assignment and allocation of resources.
- *Entrepreneur.* When an entrepreneur runs an experiment, s/he wants the experiment to succeed. If it does, the entrepreneur typically does whatever her or she can to help it grow and prosper. This is usually not an issue. Again, resource allocation is bottom-up. It's the customers who allocate the resources.
- *Organization.* In an organizational context, even if an experiment succeeds, the organization may not reward it with resources that match its success. There are two underlying reasons why this may be problematical. One reason is that decisions about organizational direction and resource allocation are made by people—typically management. Management may simply decide that it isn't interested in pursuing the experiment even though it was a success—that the new direction is not consistent with the organization's strategic plan or mission. Of course that's why managers are hired, to make such decisions. But these sorts of decisions are capable of killing off successful experimental ideas.

  In organizations, resources are not disbursed within the environment. They are pooled at the top and distributed downward. Decisions about how resources are distributed are made by people whose job it is to make them. Of course that's how we want our organizations to function. It is the responsibility of management to determine how best to use available resources to achieve their goals. But the goal of a manager is not necessarily to see to it that successful experiments grow and flourish.

A well-known example of a system that was particularly notable for its lack of innovation was the command economies of the former Eastern block Communist countries. Their failure was not a matter of insufficient initiative. There were many people with lots of initiative. The most successful rose through the ranks and wound up with dachas on the Black Sea. Nor was their failure a matter of technological incompetence. Eastern block countries had excellent educational systems and well educated people. The problem was that the structure of a command economy has no place for innovation except as a top-down phenomenon. Furthermore, even when the system created successful products, there was no direct way to tie success to increased production. Good products did not reap their own rewards. Decisions on what to produce were made top-down, creative destruction was resisted, and failed products and methods received a disproportionate share of resources.

Many hierarchical systems erect barriers—sometimes unintentional—to innovation. It is often difficult to experiment in the first place, and successful experiments may not get the resources they deserve. Innovation stalls.

Innovation is easy. All it takes is lots of experimentation and ensuring that success reaps its rewards. What's difficult is building organizational and environmental frameworks that enable that process.

9 Summary

This paper has taken a brief tour of the landscape of complex systems and explored how its concepts may be useful to systems engineering.

References

[1] Abbott, Russ, (2006) "Emergence Explained: Abstractions," *Complexity*, 12/1 (September-October).

[2] Shalizi, Cosma R. (2001), *Causal Architecture, Complexity, and Self-Organization in Time Series and Cellular Automata,* Ph.D. dissertation, Physics Department, University of Wisconsin-Madison.

[3] Anderson, Philip W. (1972), "More is Different: Broken Symmetry and the Nature of the Hierarchical Structure of Science", *Science* 177: 393-396

[4] Abbott, Russ, "If a Tree Casts a Shadow is it Telling the Time," *International Conference on Unconventional Computation*. To appear.

[5] Glennan, Stuart S. (2002), "Rethinking Mechanistic Explanation", *Philosophy of Science* 69 (Supplement): S342-S353.

[6] Machamer, Peter, Lindley Darden, and Carl F. Craver (2000), "Thinking About Mechanisms", *Philosophy of Science* 67: 1-25.

[7] Harold, Franklyn M. (2001) *The Way of the Cell: Molecules, Organisms, and the Order of Life*, Oxford University Press.

[8] Wilson, David Sloan (2007), Evolution for Everyone, Delacorte Press.

[9] Prigogine, Ilya and Dilip Kondepudi, *Modern Thermodynamics: from Heat Engines to Dissipative Structures*, John Wiley & Sons, N.Y., 1997.

[10] Way, Jeffrey C. and Silver, Pamela A., (2007) "Why we need systems biology," *Symposium on Complex Systems Engineering*.

[11] Hoare, C. A. R. (1969) "An axiomatic basis for computer programming". *Communications of the ACM*, 12(10):576–585, October 1969.

[12] Tennent, R.D. (1976), “The Denotational Semantics of Programming Languages,” *Communications of the ACM*, 19(8):437-453, August 1976.

[13] Briscoe, Bob, Andrew Odlyzko, and Benjamin Tilly, (2006) “Metcalfe's Law is Wrong,” *IEEE Spectrum*, July 2006.

[14] Iansiti, Marco and Gregory Richards (2007), “The Business of Free Software: Enterprise Incentives, Investment, and Motivation in the Open Source Community,” (draft) Available online at: http://www.hbs.edu/research/pdf/07-028.pdf as of June 15, 2007.

[15] Ostrom, Elinor, (1990) *Governing the Commons: The Evolution of Institutions for Collective Action*, Cambridge University Press.

[16] Dietz, Thomas, Elinor Ostrom, Paul C. Stern3, (2003) “The Struggle to Govern the Commons,” Science 12 December 2003: Vol. 302. no. 5652, pp. 1907 - 1912.

[17] Brachman, Ronald and Hector Levesque (2004) *Knowledge Representation and Reasoning*, Morgan Kaufmann.

[18] Darwin, Charles (1859), *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*, John Murray.

[19] Dennett, Daniel (1996), *Darwin’s Dangerous Idea*, Simon & Schuster.