To Main Proceedings Document

The Source of Poor Policy: Controlling Learning Drift and Premature Consensus in Human Organizations[1]

Jim Hines
MIT
JimHines@Interserv.com

Jody House
Oregon Graduate Institute
jhouse@ece.ogi.edu

Abstract. As system dynamicists, we spend our days finding and patching up faulty policies. We give surprisingly little thought, however, to the origin of these faulty policies. And yet, if we understood their origin, we might be able to attack the problem of poor policy at its source.

This paper presents a theory of policy formation that is consistent with what is known about evolutionary processes and human psychology. The theory is translated into a computer simulation model, which is used to illuminate several "handles" on policy creation. The handles influence two potential failure modes in policy creation: (1) "learning drift", a process in which people learn without making progress, and/or (2) "premature consensus", a process in which managers agree on a policy before the "best" one has emerged.

1. Introduction.

Poor policies in companies can produce stagnating sales, falling morale, oscillating production, excessive attrition, poor quality, slow product development and a host of other undesirable behavior patterns. Poor policies in non-commercial organizations can be similarly devastating. Where do all of these bad policies come from? If we knew the answer, we might be able to eliminate policy flaws at their source.

In this paper we present a theory of policy formation that is consistent with what is known about evolutionary processes and what is known about human psychology. We then translate the theory into a simulation model of policy creation in order to explore several ways in which the creation process can go awry.

The importance of policy. From the beginning, "policy" has occupied a central place in system dynamics. Jay Forrester spent a chapter on the topic in *Industrial Dynamics* and fingered "policy" as something beginners were likely to have difficulty grasping. Jay used "policy" in a very specific sense.

> "As used here, a 'policy' is a *rule* that states how the day-by-day operating decisions are made". (Forrester 1961, p. 93).

A decision rule at a produce market might guide a grocer in pricing tomatoes on a particular day, in software company a decision rule might guide a computer programmer

[1] This work is supported by a grant from the National Science Foundation (award # 997594) and our corporate partners: Eastman Chemicals, PriceWaterhouse Coopers, Hewlett Packard, Pugh Roberts Associates, The Lincoln School, and General Motors.

in deciding how late to work. The grocer might consider whether he had tomatoes left over the day before, and whether today's tomatoes are large or small. The computer programmer might consider how interesting the programming task is, her level of fatigue, how close the deadline is, and whether her friends are throwing a party that same night. A decision rule specifies which information to take account of and how to combine the information to arrive at a decision.

In emphasizing the importance of policy, Jay Forrester was careful to point out that most policies are not written down in manuals and that some rules that *are* in manuals are not *policies*. A rule is a *policy* if it gives rise to a stream of decisions. Other characteristics of a decision rule– including whether it's written down or even articulated at all – are irrelevant to the issue of whether the rule is a policy.

Why the emphasis on policy? As system dynamicists, we are interested in what creates behavior patterns in prices, morale, unit sales, technology and other variables. Behavior patterns-- oscillation, growth, collapse, and adjustment – come from regularity in decision making; that is behavior patterns come from what remains after stripping away erratic quirks and random urges. What remains are "policies" – everything that is systematic about decisions.

From a system dynamics viewpoint, a company *is* a system of policies. A company's particular collection of policies gives the company its unique identity – an identity that survives employee departures, spatial relocations, and corporate reorganizations[2]. A company that tends to create "good" policies will tend to have desirable behavior – rising morale, rising sales, better and better products.

2. Where policies come from: Theory.

Humans, bacteria and policy. A long time ago, one of us (Hines) believed he would find true career happiness as a banker. He describes his earliest experience:

> "I took myself and my freshly-printed MBA degree to the venerable Mellon Bank. Like the fifteen or so other new trainees, I was assigned a company along with the task of assessing whether the company merited a Mellon loan. I quickly realized that the usual procedure for assessing credit worthiness was unscientific, based on mere tradition. Consequently, I set to work performing linear regressions and Box-Jenkins time series analyses on the performance of various companies and corresponding time-histories of various financial measures. In contrast, my friends – the other trainees – went to work trying to imitate the "mere tradition". My friends aped the way our more senior colleagues assessed credit worthiness without a critical thought as to whether the process was likely

[2] Of course employee departures, reorganizations, and relocations *can* affect company's behavior patterns, but only if these changes also affect policies. For a discussion of reorganizations in the organizational chart vs. reorganizations in policies see Morecroft and Hines 1985 and Forrester 1968, 146 (Forrester 1968; Morecroft and Hines 1985).

> to produce truth and understanding. The result? My friends' work contributed to the lending decision, while mine did not. At the time, I was a bit surprised that my friends' work was valued so highly. And, I'm afraid I became a bit skeptical about the intellectual curiosity, not to say horse-power, of my most senior colleagues."

Regardless of Hines' surprise, those long-ago trainees, *by imitating the older analysts*, were learning the explicit and implicit rules by which people at Mellon decided to give money to, or withhold money from, prospective borrowers.

The trainees were also behaving like most people. Humans (perhaps with the sad exception of Hines) are gifted at high-fidelity imitation (Whiten and Custance 1996); in fact this is a big part of how we learn (Bandura 1986, Zantall 1996). Imitation is a typical, perhaps even unique[3], trait of our species (Meltzoff 1996). Infants a few months old not only imitate well, but also are also adept at figuring out the goals of adult actions (Meltzoff 1995). This latter innate skill is important, allowing humans to imitate the "deep" structure of what they are seeing, ignoring random, surface distractions.

The trainees' approach to figuring out how to reach a complex judgment should be familiar to most people who have worked in (or who have worked with people who work in) large organizations: A large fraction of time is spent figuring out the mindset of the boss (Hines 1998). People's innate orientation to sniff out intent means that they inherently strip away what is unintended, random or undirected in the behavior, to concentrate on what Forrester called *policy*.

The innate ability to learn by imitation is complemented by an innate human desire to be imitated; that is to teach (Meltzoff 1996). The result of both good imitation and good teaching is a relatively efficient one-way transmission of policy from one person to another.[4]

One-way transmission of policy is intriguingly reminiscent of a rather different form of one-way information transfer: A "donor" bacterium gives DNA genes to a recipient bacterium in a process known as "conjugation" where plasmids – DNA-encoded extrachromosomal genes, in a sense, biological "policies" – flow from donor to recipient[5] (see for example, Leach 1996; Summers 1996). Among the bacterial "policies" that spread in this way are those for resistance to antibacterial drugs.

[3] Parrots and, perhaps, dolphins and apes also imitate, though not as well as humans (Heyes and Galef 1999).

[4] We don't mean to imply that a person who is "teaching" cannot also be a learner later, or even roughly at the same time. We say "one way" to emphasize the fact that the act of imitating does not necessarily affect the knowledge of the person who is imitated.

[5] The transfer of DNA from one bacterium to another can also be mediated by viruses in a process known as "transduction". A bacterial cell can also take up "free floating" DNA from the environment in a process known as "Transformation". Conjugation, transduction, and transformation all have the same effect – the one-way transfer of genetic material from one bacterium to another. Although we use the term conjugation, our analogy holds also for these other two mechanisms of bacterial genetic transfer. A two way transfer process known as retrotransfer has also been reported (Summers 1996).

It is widely appreciated that this one way transfer of DNA has played a crucial role in bacterial evolution." (Summers 1996). That is, conjugation in addition to being imitative is also creative (Leach 1996). We would like to suggest that policy transmission from one person to another is a creative process as well – in fact, a creative process that accounts for the evolution or origin of most policies in most companies (Hines and House 1999; House, Kain et al. 2000).

Evolution of Policies. Consider the process those trainees at Mellon were part of. At the start, each trainee had in mind some notion of how to assess the credit worthiness of a company, perhaps vague and probably acquired through a learning/imitative process that predated joining Mellon. Each one (except Hines, of course) then asked one or more senior analysts or lending officers how to proceed and also poured over their assigned company's prior credit evaluations. The trainees tried to strip away what was idiosyncratic, random, or tied to the peculiarities of a past time or other company – and tried to identify the underlying principles, the policies, that guided the credit evaluation process. In the process, they no doubt made mistakes. And, perhaps, even tried to make small improvements. The result was a combination of the newly extracted policies, perhaps slightly modified, and their existing understanding of what they were supposed to do.

The process described above is analogous in every important respect to the process by which biological evolution proceeds in bacteria; evolution depends on passing a high-fidelity *copy* of information. The creative part of evolution comes in the two separate processes of *mutation* and *recombination*.

Mutation. High fidelity genetic copying in organisms is accomplished through a complex molecular process of DNA replication that corresponds in the organizational realm to imitation of policy. Biological mutation covers a variety of genetic processes in which a segment of DNA or a single nucleotide is randomly inserted into or deleted from a DNA molecule (See for example Li 1991). Biological mutation corresponds to mistakes made when imitating (that is, learning) policies. For example, a senior analyst might say that the one thing that for sure disqualifies a company from getting a loan is any scent of illegal dealing; management's every action must be governed *by law*. An analyst might hear incorrectly and proceed to always read the corporate *bylaws* of companies seeking loans. The analyst's mistaken innovation – for better or worse – would be a kind of mutation.

Intentional innovation can also be seen as a kind of mutation. Certainly, people *within* the system often adopt this view. No matter what Hines *intended,* the crusty old credit officers of Mellon Bank viewed Hines' effort as an (unwanted) mutation. But, the similarity between random mutation and intended innovation also holds at a deeper and broader level.

Like mutation, the global result of intentional change is highly unpredictable. Our experience over many years coaching people through the beer game (Sterman 1989;

Senge 1990); building and then leading people through management flight simulators, and working with clients on system dynamics models, suggests to us that improvement is difficult even in simulations, which are much less complicated than real companies. Clients and students *routinely* are wrong in their beliefs about how a parameter change will impact model behavior. Conversations with other system dynamicists suggest that our experience is hardly unique. Experimental results suggest that people are *not* very good at intuiting the feedback impacts of their attempts at improvement (Kampmann and Sterman 1992; Diehl and Sterman 1995; Sterman, Repenning et al. 1995; Sterman, Repenning et al. 1996,).

Both consulting experience and scholarship suggest that an improvement attempt succeeds only by *chance* when the critical, but poorly perceived, feedback structure does *not* interfere. Our working hypothesis is that *from the viewpoint of global organizational improvement*, intentional change has little more probability of success than accidental change.

Recombination. It is unlikely that the trainees were learning *entire* policies. Instead they probably were taking *pieces* they understood and fitting those pieces into their own pre-exiting understanding. This process in genetic systems is termed recombination (Leach 1996).

Recombination in bacteria occurs when genes from the donor are integrated into the recipient's own genes (Summers 1996). Occasionally this results in two similar but not identical genes (one from the donor, one from the recipient) getting spliced together: a part of a gene from the donor replaces the corresponding part in the recipient. The result is a new gene that might have the front end of the donor's DNA and the back end of the recipient's original gene. If the donor had a superior front end, and the recipient a superior back end, the resulting gene will be better than either the donor's or the recipient's prior to recombination. Work with genetic algorithms (Goldberg 1989; Holland 1992) and genetic programming (Koza 1992) suggests that recombination is even more important to evolution than mutation. .

To get a sense of the potential power of recombination, consider a simple example of numerical recombination. Let's say that bigger numbers are better. And, lets say that we start with two numbers, say **2,000** and *1,555*. Say **2,000** "donates" its first two digits (**2** and **0**). The donated digits replace the first two digits (*1* and *5*) in the "recipient" whose new number become **2,0***55* – larger than the donor and larger than the recipient before recombination. If by chance, only the donor's first digit were donated, the recipient would swell to **2**,*555* – bigger yet.

Recombination, remarkably enough, can result in combinations that are "better" than what went before. The "trick" is that the recombination is *not* an averaging process. Obviously if we averaged 2,000 and 1,555 we would get a number 1777.5 which is not larger than *both* "parents".

Recombination occurs routinely in policy imitation, because the learner usually will imitate only part of the teacher's policy, integrating that part into his pre-existing policy. For example, say that in order to lend money, I consider the ratio of a company's profits to its loan-service burden, and grant the loan if the ratio is above some number X. *You*, on the other hand, look to see if cash flow has been positive and steady over the past five years. I understand (without even thinking about it) that our intent is broadly similar – we are looking for a stream of un-fettered money with which the company can service debt. However, profits reflect a lot of non-cash items, such as depreciation, which are actually "unfettered". Net cash flow is a purer reflection of the ability to pay back a loan. Perhaps, I learn from you, and switch to a ratio of *net cash flow* to debt service. In this case, I've combined a part of your policy with my own and have created a new policy. It's possible (though by no means certain) that this new policy is superior to the one you are following and to the one I previously followed.

In summary, policies arise from an evolutionary process of imitation and alteration in which a part of one person's policy is incorporated into another person's policy with innovation occurring along the way. Essentially, the process is piecing together good policy components.

3. Modeling Policy Creation. We would like to illuminate the above theory with computer simulations. Accordingly, we turn now to the problem of representing the above theory in a computer. The representation is challenging from a system dynamics perspective because the inherent "discreteness" of recombination is not well served by system dynamics' usual preference for continuity.

In system dynamics models, combinations usually involve smooth blending or accrual. To get productivity Barry Richmond takes a weighted average, blending the productivity of rookies and with that of pros. To track the energy efficiency of housing, John Sterman creates a co-flow that blends the energy efficiency of a time-steam of housing. George Richardson and David Andersen accumulate a population of sick people by smoothly accruing them into a stock. These kinds of combinations, simple and expressive though they may be, are all combinations in which individual identity is lost. The process is one in which all characteristics of an individual are added to or averaged with others; and what is added can never be separated again.

Recombination of genes or policies is different. In recombination, "addition" involves adding some part of a thing, while other parts are not added. Furthermore, the added part remains untouched and identifiable. You combine the idea of a lunchtime sandwich with an egg from breakfast to get an Egg McMuffin, something new and something that maintains its constituent parts so that they could be re-combined with something else (Hines and House 1998). To make the recombination of an Egg McMuffin, you can*not* simply puree lunch and breakfast in a Cuisinart and pour off another meal.

In order to explore the evolution of policies we will need to augment the "normal" system dynamics modeling process with a technique that retains the separate identity of individuals in a population. More specifically, we will combine a traditional system dynamics model with an agent-based representation of the population of managers who determine (some of) the policies in the system dynamics model.

The system dynamics portion represents a number of business units. In what follows, our model represents a software developer that has a number of products – word processor, spreadsheet, C++ compiler, Internet browser, personal information manager, operating system, etc. Each product is managed separately and is represented by a project model (Cooper 1980; Abdel-Hamid and Madnick 1991; Cooper 1993) that simulates repeatedly. Each time a project model completes, a new version of the corresponding product is released. Then, the project model starts up again as programmer get to work on the next version. The "whole" system dynamics model is composed of all the project models.

The agent-based portion of our model is composed of a population of (usually fifty) managers. Here we can *not* simply pour all the managers into a stock, because we need to keep individual managers' policies separate. Instead, we represent each manager as an object that has a policy[6] and we keep track of each manager and his policy individually. The manager's policy controls a decision in one of the project models. Each manager's policy changes through the life of the run as he learns (via recombination) from other managers and as he innovates. All of our experiments to date have represented policies as numbers, so learning is very similar to the recombination of **2,000** and *1,555* that we looked at previously.

The algorithm in brief is (1) simulate each project model using the policy of the appropriate manger(s), (2) if using teams, mix the managers and reform the teams, (3) let managers learn from one another, (4) let managers innovate.

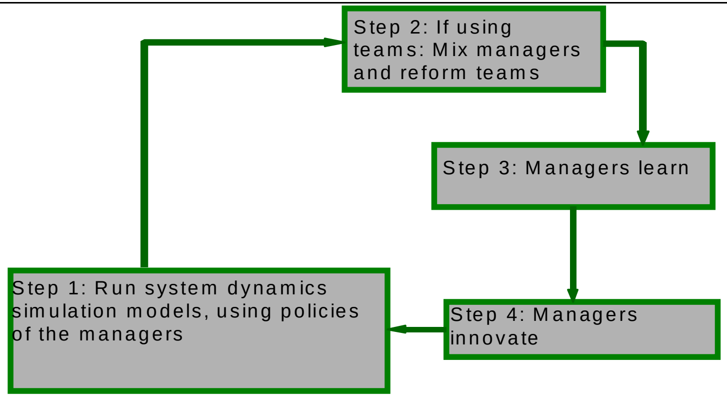

Step 2: If using teams: Mix managers and reform teams
Step 3: Managers learn
Step 1: Run system dynamics simulation models, using policies of the managers
Step 4: Managers innovate

Figure 1. Flowchart of a simplified learning algorithm.

There are actually two additional steps, which we will describe shortly. But for now the steps are

1. **Simulate the model**. Simulating the model means to simulate simultaneously all of the constituent project models. The particular person managing a given release

[6] The objects also have other characteristics, as we will discuss below.

determines the parameters of the project model for that release. In some of our runs, teams of managers manage projects. In this case, the project model's parameters are determined by taking a weighted average of the policies of the team member, with the weights determined by seniority.

2. **If using teams, mix managers**. In simulations involving teams, managers only learn from teammates, reflecting our original thinking that people are most likely to imitate people with whom they are currently working. Recent interviews with our partner companies suggests that in fact this is not true – people tend to learn from people with whom they have worked in the past in addition to people they are working with currently. This suggestion will alter the nature of this step in our future work. At the moment, however, we need to mix managers from different releases so that managers get to learn from other managers who have used different policies in past releases.
3. **Learn.** Every manager has a certain probability of learning (perhaps 100%), after his project completes. If the manager is going to learn, he chooses a "teacher". For the moment, we will say that the teacher is chosen randomly. After choosing a teacher, the learner randomly substitutes part of the teacher's policy for the corresponding part of his own policy. More specifically the algorithm randomly picks a position in the policy number of the teacher (e.g. if it were a 6 digit number, we could pick the 1st, 2nd, 3rd, 4th, 5th, or 6th position). The teacher's number then consists of two parts – the part up to and including the randomly chosen position, and the part that comes after. The algorithm then randomly picks one of these two parts and uses it to replace the corresponding part of the "recipients" policy. At this point, the recipient has learned from the donor.

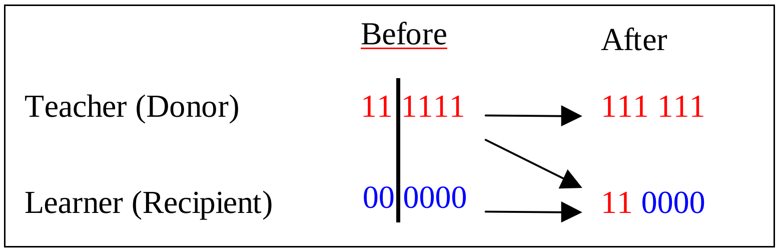


Figure 2: Learning by recombination

4. **Innovate.** As discussed above, we think of innovation broadly to include both accidental and intentional changes to policy. In our models, the learner has some probability (perhaps 0) of innovating after learning. If he does innovate, he will randomly choose a position on his new policy number and change the corresponding digit – in a binary run the manager simply switches a 1 to a zero.

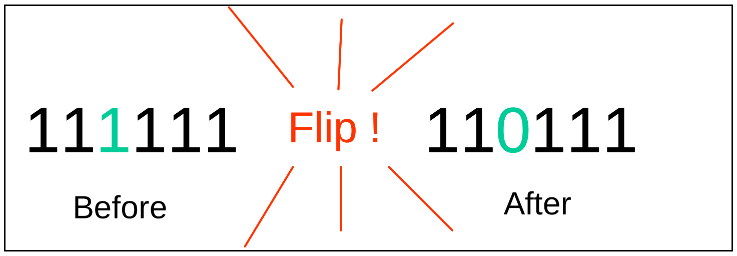


Figure 3: Innovation (mutation)

We have investigated the evolution of several policies in project models using this algorithm (or a slightly more complicated algorithm described in the next section). Policies we have investigated include time to hire, time to change schedule, and, most simply, just the number of programmers to have on the project. In this paper we present simulations using the simplest policy – the number of programmers on the release. And, we let managers decide on any integer number between zero and 15.[7] The reason we use this very simple rule is entirely for clarity; we would get similar results using any of the policies investigated to date.

The project model corresponding to our very simple policy is also simple and is diagrammed below. Equations follow. Note that the equation for Final Time means that the simulation will end when 95% of the code is correctly written, or after 15 months, whichever comes first.

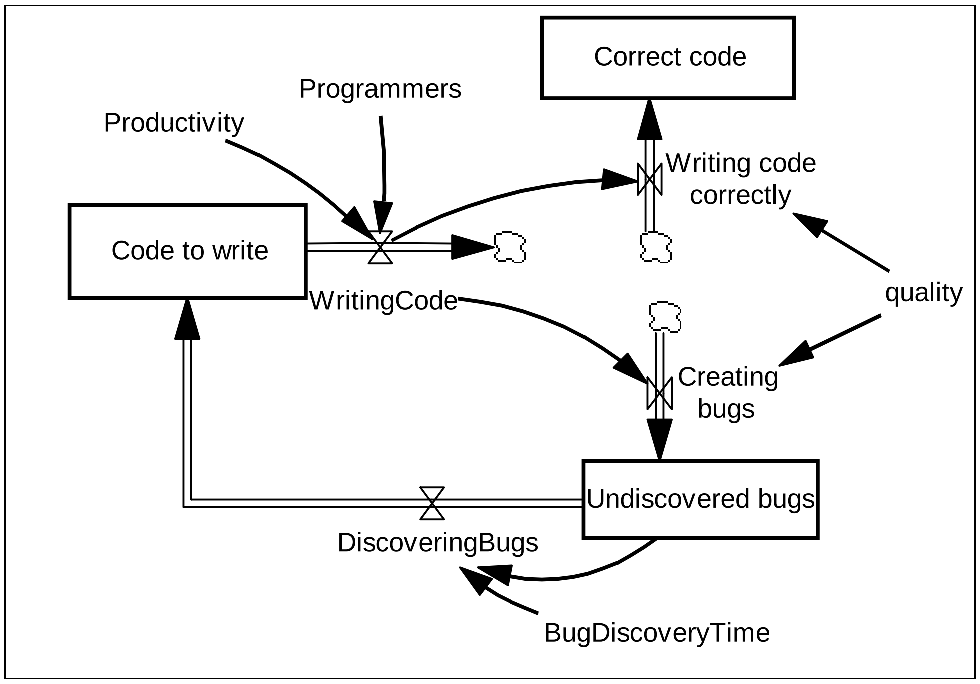


Figure 4: Simple project model for simplest policy

CodeToWrite = Integ(DiscoveringBugs – WritingCode, InitialCode)	lines
InitialCode = 100	lines
WritingCode = Programmers*Productivity	Lines/Month
Productivity = 2	lines/programmer/month

[7] When simulating other policies we use floats with some maximum precision.

Programmers = {determined by manager at the beginning of the run}	people
DiscoveringBugs = UndiscoveredBugs/BugDiscoveryTime	lines/month
BugDiscoveryTime = 0.25	months
UndiscoveredBugs = Integ(CreatingBugs – DiscoveringBugs,0)	lines
CreatingBugs = WritingCode*(1 – Quality)	lines/month
Quality = 0.7	fraction
CorrectCode = Integ(WritingCodeCorrectly, 0)	lines
WritingCodeCorrectly = WritingCode*Quality	lines/month
Final Time = if then else(CorrectCode >= 0.95*InitialCode,0,15)	Months

4. Learning drift, pointing and pushing.

Selection. We were surprised by the model's performance. We had anticipated that models incorporating the above learning algorithm would perform very well. There is no cost to having more programmers – that is, there is no labor cost or effect on productivity or quality from over-staffing. Accordingly, we anticipated that eventually *all* of the managers would learn that the best number of programmers was the maximum number, fifteen. We were wrong.

Our initial run involved 50 managers each of whom managed one of 50 products (i.e. 50 project models). To keep things simple, we eliminated innovation, so that only learning (i.e. recombination) was active. We initialized the model by assigning to each manager a random number between 0 and fifteen, representing his initial desired number of programmers. Once the simulation began, managers learned from one another after each release as described in the algorithm above.

Below we show a run like the one that we saw originally. We have plotted each manager's desired number of programmers over the life of the run. We can do this because we keep track of each manager individually. Clearly, the managers are *not* learning that fifteen programmers is best.

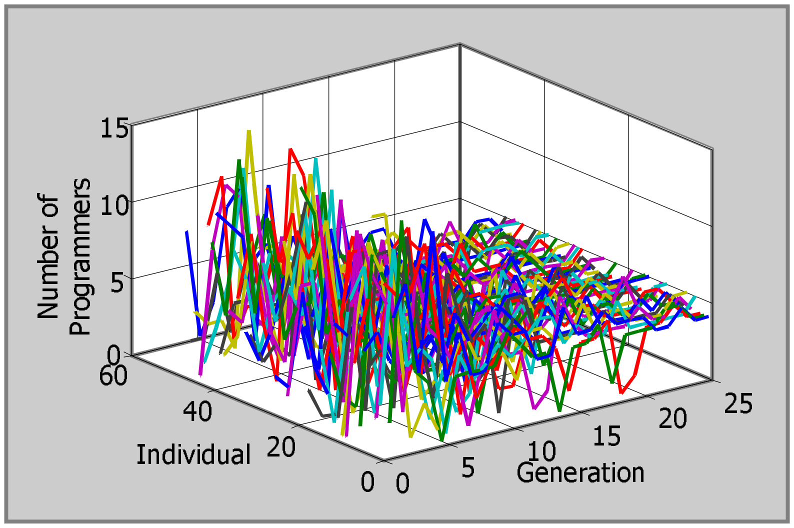


Figure 5: Number of programmers on each release in a typical simulation of initial algorithm

To get a sense of what is happening overall, we plot below the average number of programmers per release through the life of the simulation. In this run, the average programmers per release meanders downward. In some runs it will meander up, in other runs it will meander around the point it started. In fact, the average number is drifting randomly.

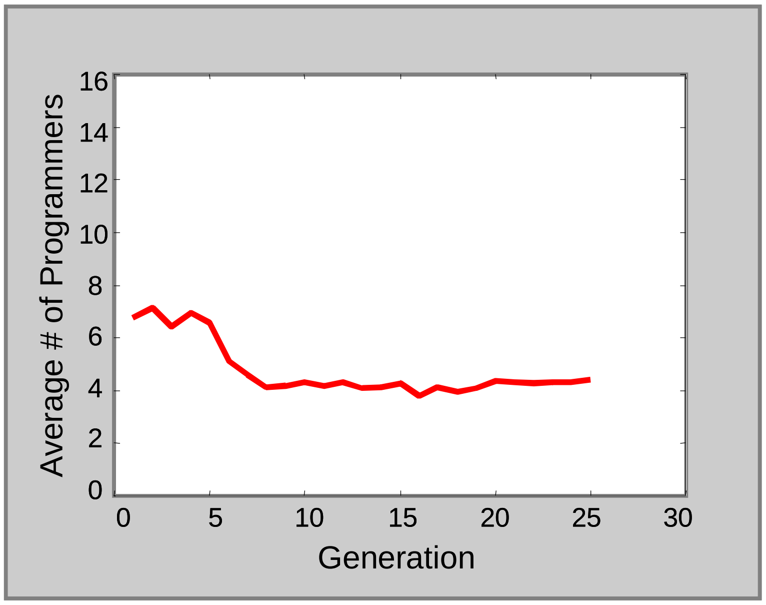


Figure 6: Average programmers per release for typical run of initial algorithm

Interestingly, though, *something* is happening, even if the average is just drifting randomly. Figure 5 shows that variance declines, that is the managers are moving toward consensus, a consensus that is "wrong", but a consensus nonetheless. Of course a pattern in a single stochastic run needs to be treated skeptically. At the time of this writing, we have made hundreds of runs. Variance always declines. Managers always moved toward agreement, even though *what* they agreed on is random.

We are witnessing learning drift, something similar to genetic drift in natural populations (Nei 1987; Smith 1989; Li and Graur 1991; Gillespie 1998). Genetic drift occurs when natural selection is weak with respect to some particular genetic variation – that is when survival rates are the same for animals having any of a number of different variants (alleles) of a gene. In such cases, allele frequencies will drift randomly and in finite population one of the variants will randomly come to dominate, or become "fixed", in the population.

Learning drift occurs because the modeled managers have very weak, in fact no, guidance concerning which policy is better. The best number was obvious to us, and we had blithely assumed it would be obvious to the managers in the model. But our modeled managers are truly stupid. *They* don't know that 15 is best. Our starting point was that real managers are ignorant with respect to the complexity of the real world and we created our *modeled* managers so that they were analogously limited with respect to their *modeled* world. Our model reminded us that the correct policy is not obvious to real managers in the real system. Simply being able to imitate, innovate, and recombine other people's policies will not lead to anything other than learning drift.

Without some kind of direction, managers in the real world can make whatever argument they desired to one another about what policy is desirable. But because, the real system is much more complicated than any manager's understanding, what a manager argues for can have little more chance of producing success or failure that what any other manager argues for. Who convinces whom of what is essentially random.

Insidiously, managers *agree* on what is the best policy, even though that policy is actually chosen randomly. This "random consensus" results from loss of information or opinion. Technically, recombination doubles one part (the part that is learned) and discards another part (the part that is "unlearned"). For example, if we had a population of three individuals described by 11, 00, and 10, and if the second individual learns the initial digit of the first, we end up with three individuals 11 (unchanged), 10 (changed), and 10 (unchanged). The second individual discarded his "idea" of a zero in the first position. At this point, no one in the population believes in a 0 in the first position. The idea has been lost and recombination will never rediscover it.[8]

Random consensus often characterizes learning drift in organizations, just as "fixation" characterizes genetic drift in finite natural populations. Not only can one manager convince another; when the other manager is convinced the group loses an opinion. Eventually, all opinions are lost except one. This process is easily observed in organizations. Put a group of (friendly) managers together in a room and they will in fact ultimately agree on some policy. Indeed, the process is so reliable that consultants can make their living by *promising* to will lead managers to consensus. Of course consultants don't promise that the consensus solution will be good solution.

Giving direction: Pointing and pushing mechanisms. In order to for an organization to good policies naturally, it must point managers in the right direction. Unfortunately, we just argued that no one in the organization can tell the difference between a good policy and poor one, that is no one can be relied upon to point managers toward the right policy[9]. This conundrum is surprisingly easy to solve: An organization need not point managers toward *what* should be imitated as long as it points them toward *who* should be imitated.

This is the way that natural selection works. Natural selection does not tell bacteria *what* to inherit, only that they should inherit from other bacteria that have been successful

[8] As mentioned, we had turned innovation off in the simulation. A simulation with innovation will continually introduce variance into the population. However, innovation will not result in better average performance – There is still no direction being given.

[9] We feel compelled to offer one exception. In our experience managers guided by a system dynamics process can sometimes detect defective policies that are causing problems.

enough to survive. Nature essentially points to successful bacteria by eliminating unsuccessful ones.

Similarly, an organization, through less draconian means, can *point* to managers who others should imitate and *push* others to learn from them. That is organizations can (and in fact do) establish pointing and pushing mechanisms.[10] Who should the mechanism point to? Like natural selection, an organization's pointing and pushing mechanism should encourage people to learn from those who have been successful[11].

An organization can point to successful managers by putting them in corner offices, having them serve a stint in training, paying them more (and publishing pay scales), and even, like nature, by eliminating unsuccessful managers. Each of these mechanisms is actually in use. But, probably the most powerful and certainly the most widespread pointing and pushing mechanism is the management hierarchy.

The hierarchy points to senior people. A person is motivated (pushed) to imitate senior people by the hope that he himself might rise as well. Those long-ago Mellon trainees did not seek advice at random; they zeroed in on the most senior people they felt they could approach.

Note that the people do not need to *know* which policies cause the kind of success that moves senior managers up the hierarchy. Indeed, people will no doubt mistakenly learn some things that are irrelevant or even inimical to promotion, but still there will be a systematic bias toward learning whatever it is that causes promotion. If a company promotes managers who manage successful business units, then people will be biased toward learning policies that produce successful business units.

The Full Algorithm. In order to represent a pointing and pushing device, we give each manager an additional characteristic: a "hierarchical" position, represented as a variable initially set to one. We also expand the algorithm to include two additional steps: performance evaluation and promotion. These two steps are inserted between steps one and two of the original algorithm (so step two of the initial algorithm is now step 4). In addition, step 5 is modified. The new algorithm is diagrammed below.

[10] Recent interviews have suggested that there may also be a role for pushing successful managers to *teach*.

[11] Our theory does not specify how "success" in the real world should be defined. In fact, organizations have enormous latitude in deciding what to label success – and hence in deciding the directions along which they want to evolve. It would be fine to measure success by profitability. It would be just as satisfactory to measure success by employee satisfaction. Competition imposes some constraints, but whether and how tightly they bind an organization is not known.

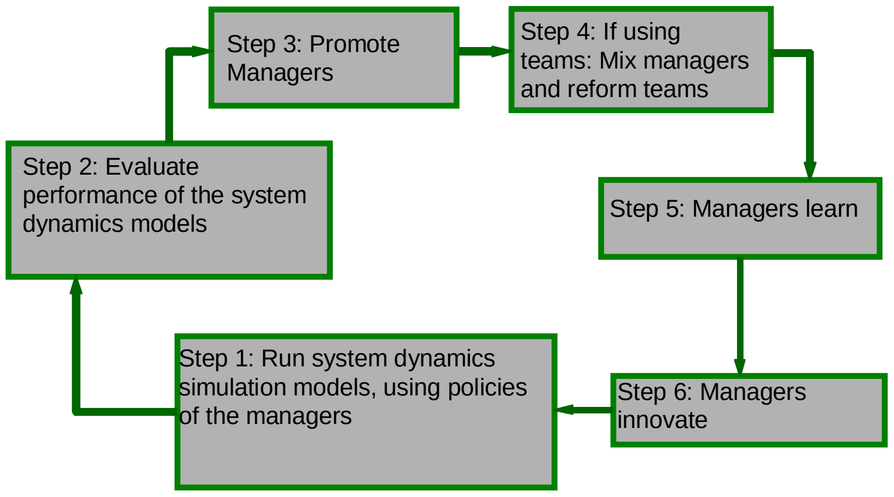

Step 3: Promote Managers
Step 4: If using teams: Mix managers and reform teams
Step 2: Evaluate performance of the system dynamics models
Step 5: Managers learn
Step 1: Run system dynamics simulation models, using policies of the managers
Step 6: Managers innovate

Figure 7: The full algorithm

Step 1: Simulate the model. Same as prior algorithm

Step 2. Evaluate performance. The "result" of managers' policies is evaluated and ranked on explicit criteria. For example, in our very simple model, described above, we rank the releases for all the different products on the basis of how quickly the release was completed.

Step 3. Promote managers. The manager who managed (or the managers who were on the team that managed) the release with the shortest completion time is given a new position equal to twice his prior position. The manager (or team) that managed the release with the longest completion time is given a new position equal to *half* his prior position—that is he is demoted.[12] Promotions (demotions) of all other managers are spread out between these two extremes depending on how their project ranked, so the manager who managed the median project is neither promoted nor demoted (his new position is one times his old position).

Step 5. As before, managers learn by recombination. And, in the base case managers always learn from someone. However, now the probability that a particular person will be chosen to be a "teacher" is proportional to the person's position in the company.

Step 6. Innovate. Same as prior algorithm.

[12] In our simulations it is important that people be discouraged from learning from unsuccessful managers. Whether this actually requires demotion or not in the real world is unknown – what is clear is that people need to be strongly directed to learn from managers who have been successful. If small differences in pointing are *perceived* as being large differences, demotion (or more generally pointing *away* from unsuccessful people) may not be necessary. On the other hand, at least two of our partner companies actually do demote people.

The performance of learning with a pointing and pushing mechanism is remarkably good. In most runs, like the one shown below, managers converge rapidly and to a very good policy – often (though not always) the best possible.

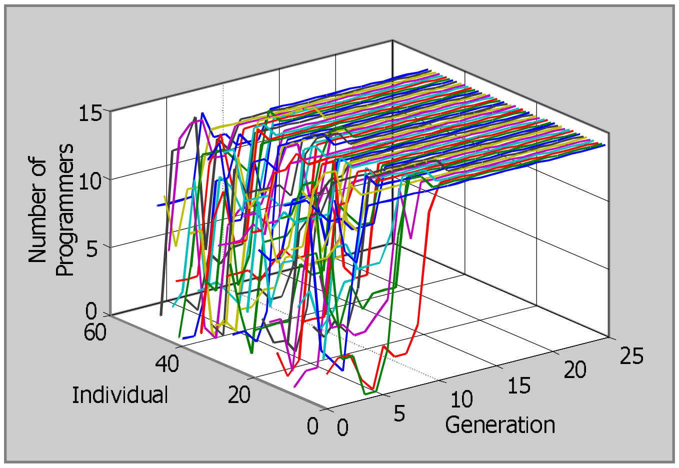


Figure 8: Pointing and pushing mechanisms provide direction to policy evolution

At this point our learning algorithm is similar to the "standard" genetic algorithm [Goldberg, 1989 #9]. The genetic algorithm is usually used to find an optimal solution of some problem. It and its variants have been applied with great success to areas as diverse as microwave antenna design and airline scheduling. We have compared the performance of our learning algorithm to the "standard" genetic algorithm [House, 2000 #13]. Keep in mind that our learning algorithm was designed to represent a real process of policy development and not to be efficient at optimizing. Nonetheless the learning algorithm is about 70% to 80% as efficient as the standard genetic algorithm.

5. Handles on policy formation.

5.1 Innovation. Sometimes as above, the algorithm gets stuck at a sub-optimal place. In the plot above, the simulation gets stuck at 14, rather than 15 programmers. It's possible, (though in these simulations very rare) that a simulation will get stuck at an even lower number of programmers.

A good pointing and pushing mechanism does not *replace* random consensus. Rather pointing and pushing is layered on top of it. (See Gillespie 1998 for a cogent description of the biological analogue). Random consensus causes a random loss of information, while pointing and pushing is a process that favors the preservation of *good* policy components and the *loss* of poor policy components. The two processes are in a stochastic race, the outcome of which depends on initial policies, the particular sequence of randomness, *and* on the relative strength of the two processes.

In the graph above, managers start moving in the right direction, but reach a consensus a bit too early. The force of consensus is operating a bit faster than the movement to optimum. The result is premature consensus, completely similar to premature convergence in genetic algorithms. Premature consensus can be ameliorated by injecting new ideas; that is, by innovation.

Managers in the simulations we've discussed to this point don't innovate, that is they never change what they learn. They do not make mistakes when imitating and they do not try anything new on purpose. Injecting a little innovation into the run above produces the run below:

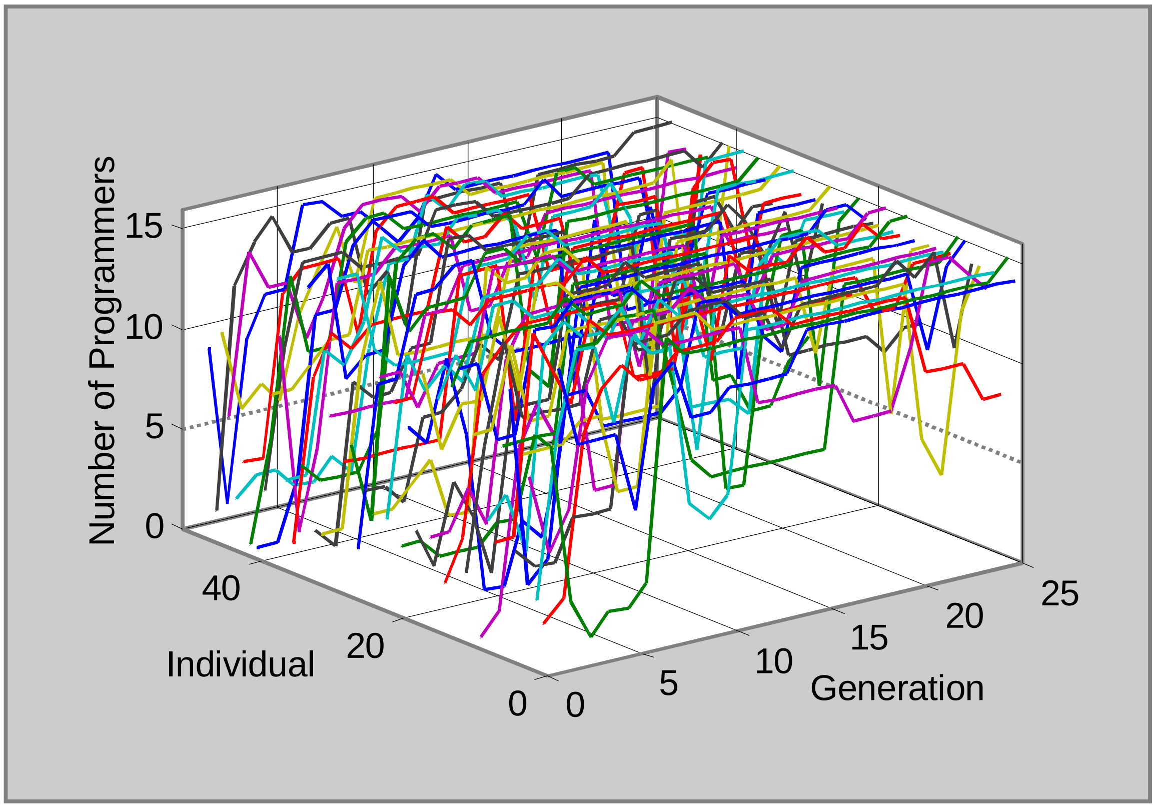


Figure 9: Innovation prevents premature consensus

The simulation no longer gets stuck at a number below 15 programmers. Of course the continuing innovation also prevents the simulation from getting "stuck" at the optimum.

That innovation can be good won't surprise many people in 21st century, industrial economies. Most people attribute progress to innovation. Long ago at Mellon Bank Hines held this belief, too. However, in our framework imitation, rather than innovation, is responsible for progress. The other Mellon trainees, not Hines, were participating in progress. Innovation in our framework serves only to prevent premature consensus. We view improvement as a social process of people learning from or imitating one another. Individual innovation by itself is unlikely to create the right policy; however innovation does create (or recreate) ideas that then can be recombined with other ideas to produce better solutions.

However, there is a dark side to innovation: Innovation is disruptive and in particular can disrupt policies or relationships between policies that were working well. In making your favorite dish, it is inadvisable to randomly vary proportions or to randomly substitute a new ingredient for another. People anticipating the birth of a child are rightly concerned about genetic mutation ("innovation") – the human system is so highly interconnected that a mutation is far more likely to *disrupt* something that works reasonably well than it is to *improve* it. Sterman, Kaufman, and Repenning report on an innovation at Analog Devices in which yields were greatly improved and manufacturing costs reduced [Sterman, 1995 #74]. Unfortunately, the new innovation changed the old relationship between direct and indirect costs, a relationship that had been implicitly built into the pricing policies. As a result, Analog reduced its prices and touched off a price war with competitors. The result was disastrous financial performance over several years.

The problem with too much innovation is illustrated in the figure below, showing average number of programmers for simulations run with different innovation (mutation) rates (the results from several runs at each mutation rate are averaged).

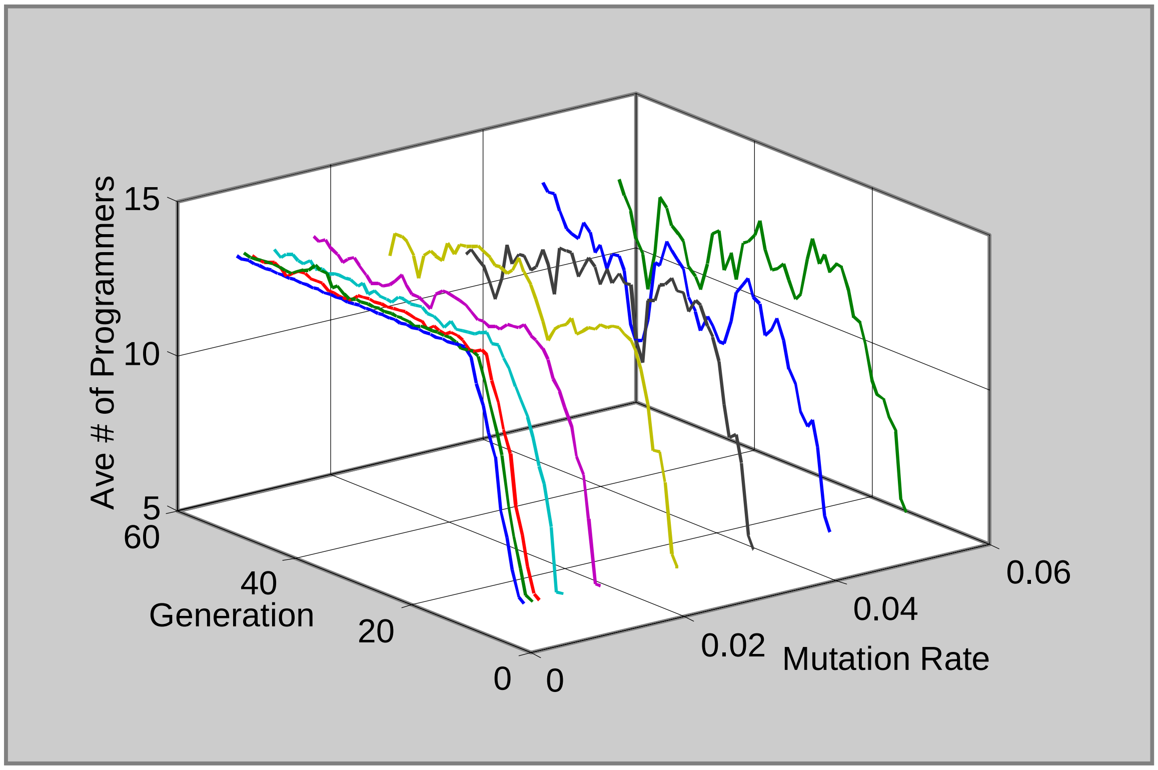


Figure 10: Excessive innovation degrades performance

The "equilibrium" value that each run achieves represents the maximum complexity (length of consecutive digits) of a solution that can be maintained in the presence of innovation. High probability of innovation (6% per manager per project-time) results in policies that "max out" at relatively low performance; the lower the innovation rate in the figure, the better the policy. (Note that 0.02 is the innovation rate that we used in Figure 9 above; the optimal mutation rate is even lower).

Because mutation is most often disruptive, genetic systems go to considerable trouble to reduce its frequency. Even simple bacterial cells only rarely make a mistake. How rarely? Mutations in bacteria and higher cells are around one in one hundred million duplications (cell divisions) (Smith ***). Much of this accuracy is attributable to error detection and correction machinery in the cell. Similarly, a manager's ability to sniff out intent can probably correct some errors in the policy being imitated (cf. Meltzoff 1995).

Organizations themselves can also discourage innovation. Hines felt that Mellon managers discouraged creativity in credit evaluations. It's possible that organizational suppression of new ideas is beneficial. Regardless of the merits of Hines' innovation in isolation, it's rather highly that a radically new way of assessing credit worthiness would have reduced Mellon's overall efficiency in making and collecting loans.

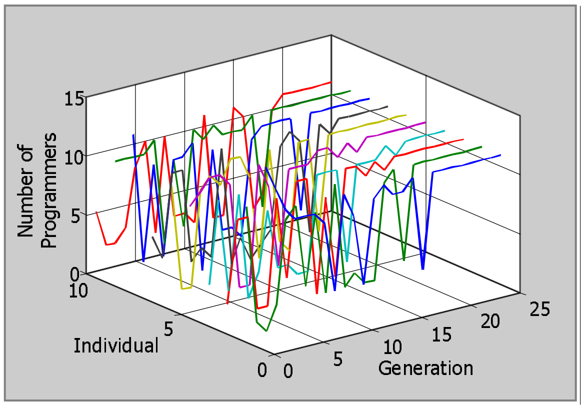
Regardless of managers' ability to sniff out intent and regardless of organized suppression of new ideas, policy copying among humans is obviously grossly less accurate than genetic copying in organisms. Is it "too" inaccurate? Our own work suggests that rates of innovation must be very small in order for the benefits of policy innovation to out-weigh the costs. It seems quite possible that the pace of policy innovation – reorganizations, reassignments, new initiatives, new policy ideas etc – actually reduces the pace of organizational improvement. Our guess is that human organizations "suffer" from too much innovation.

5.2. Number of Business Units. In deciding how much innovation is appropriate companies need to trade off premature consensus against policy disruption. The number of business units within an organization also influences the possibility of premature consensus.

A particular policy component is more likely to be lost if it is held by only a few people. For example if only one person uses cash flow in his credit assessment policy, as soon as that one person changes his mind and begins using profit, no one will be left to demonstrate (or argue in favor of) using cash flow. When only a few managers in total are involved, then *every* policy components is held by only a few people, and they will reach consensus faster. This notion is supported by everyday observation: Whether the subject at hand is deciding which corporation should get loans or where to go to lunch, agreement can be reached faster when fewer parties are involved. In other words, the force of random consensus is stronger, and premature consensus is more likely, when fewer managers are involved.

The principle is illustrated in the runs shown below, in which pointing and pushing is turned *off*, so that *only* random consensus (and learning drift) remains. There are ten managers running ten releases in the first simulation, and fifty managers running fifty releases in the second simulation. Note that consensus is reached much sooner in the simulation with ten managers.[13]

[13] The run with fewer managers converges on a higher number of programmers only by chance in these runs. *Where* the runs converge is completely random. *When* the runs converge is not.



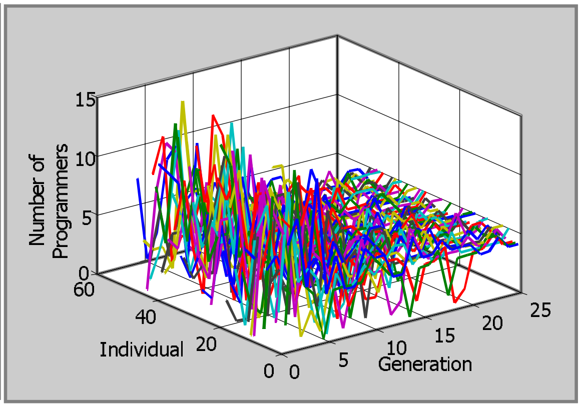


Figure 11: Random consensus comes earlier with fewer business units

If we now reinstate pointing and pushing, we set up the race between consensus and movement toward better solutions. Consensus is strengthened as the population of managers and projects becomes smaller and vice verse. When the force toward consensus is too strong, premature consensus becomes a very real threat. When the force toward consensus is too weak, consensus takes longer than necessary so that parts of the company are needlessly managed sub-optimally.

In the simulations shown up until now, the number of business units (i.e. the number of project models) has been the same, because each manager runs a release. In Figure 11, when we increase or decrease the number of managers we also increase or decrease the number of business units. Which is more important in setting the pace of consensus, the number of managers or the number of business units?

The figure below plots the time to convergence of a number of runs that differ in the number of business units, but not the number of managers. The number of managers stays the same, but they are grouped into differing numbers of teams, with each team managing a project. (Pointing and pushing is active in these runs). The trend is unmistakable; it takes longer to converge as the number of business units (that is, the number of management teams) increases.

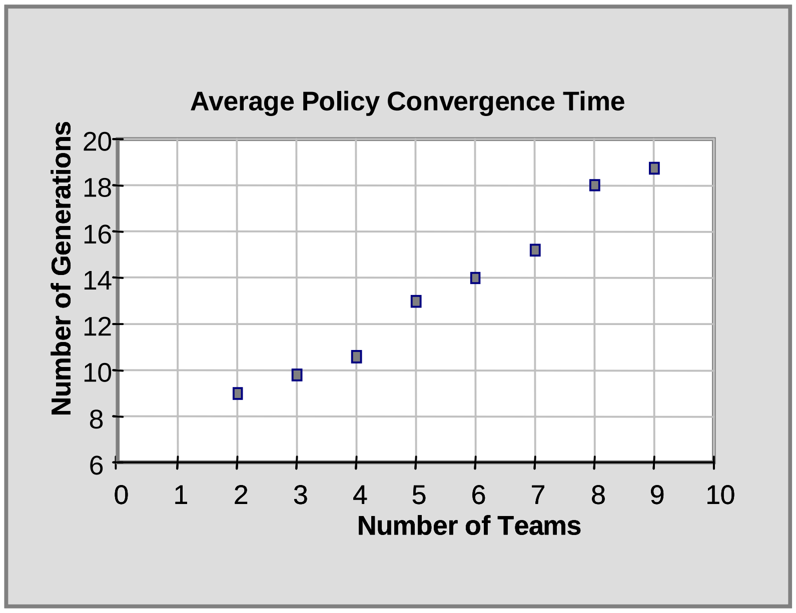


Figure 12: The time to consensus lengthens with the number of business units

While the agreement time lengthens, the *quality* of what is agreed upon increases with increasing number of products. The plot below shows the average policy value after convergence for runs with different numbers of teams.

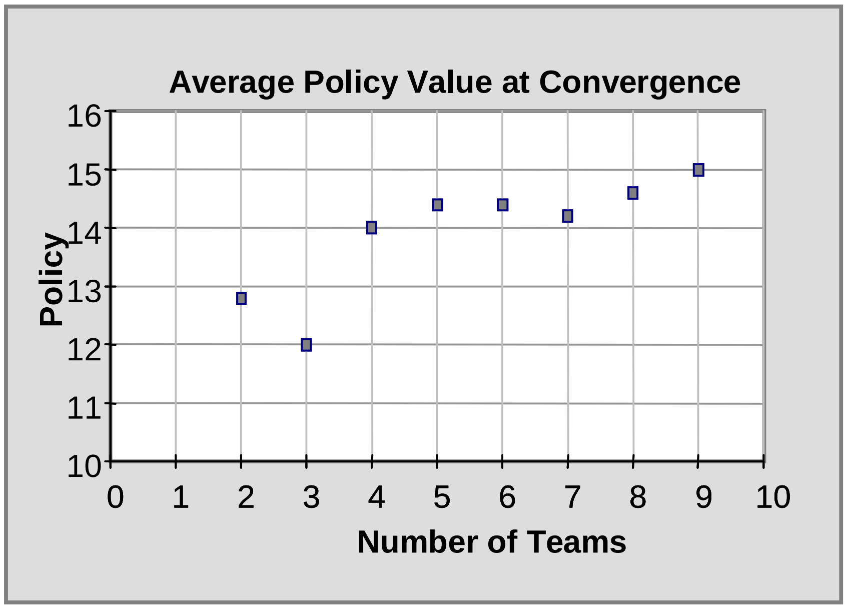


Figure 13: The ability to find the best policy increases with the number of business units

We have also run the opposite experiment, maintaining the number of business units, but varying the number of managers. We find a much weaker association between the number of people and the convergence times, than between the number of business units and the convergence times.

The reason that the number of business units has more evolutionary oomph than the number of people is that the most important information is in the managerial outcomes. That is our fitness function looks at how each business unit performed. The number of managerial experiments in an evaluation period is equal to the number of business units, not the number of people managing the business units. What causes slow or fast and good or poor convergence is the number of experiments (the amount of outcome information) – the more business units, the more managerial experiments (outcome information). As the number of business units increases, it takes longer for a good policy to spread; but the organizations ability to find a good policy increases.

5.3. Appraisal Cycle Time. One frequent criticism of publicly held companies is the shortsightedness of the stock market and, hence, of top management. Critics believe that the short time horizon creates the wrong incentives – managers are unwilling to take actions whose benefits accrue in the distant future and, particularly, actions whose short-term impact is negative.

Although a short time horizon may induce managers to sacrifice the future in favor of the present, our exploration of policy evolution suggests a further problem; one that cannot be combated by new and better incentive programs. A too-short time horizon will weaken a company's pointing and pushing mechanism, perhaps to the point where learning drift and random consensus dominate.

Business decisions like granting a loan, building a plant, or deciding to introduce a new product should be evaluated for success or failure after a period of years. But short-term pressures militate in favor of more frequent – at least annual – performance appraisals. In most instances a yearly review will be appropriate only by coincidence. Take our model where the criterion of success is time to ship a product. The obvious evaluation point is when a product has been shipped. If a release takes a couple of years (as it does for operating systems and computer languages), but managers are evaluated yearly, there will be a mid-project performance review containing very little information for every end-of-project review. The pointing and pushing mechanism will incorporate noise (the mid-project review) of the about the same magnitude as the real signal (the end-of-project review).

Illustrating this with our usual model is difficult, because it's unclear what mid-project evaluations would look like when the criteria is time to ship. Instead we investigate with the criteria of quality: The best projects are now those that ship, not soonest, but with the fewest bugs.[14] To illustrate, we compare a run of our simple model where three

[14] Other changes have been introduced to create more of a "spread" in the bugginess of runs as the number of programmers varies. These additional changes are: Quality is reduced from 0.7 to 0.5; and time to find bugs is lengthened from 0.25 to 2. In addition, the shipment criterion is changed from shipping when the product is actually 95% to shipping when the product is *apparently* 95% complete – that is, when undiscovered bugs plus correct code is greater than or equal to 95% of the original code to write. Finally, each project model is given until time 100 to ship the product.

evaluations are made during every project with the run in which a single evaluation is made at the end of the project.

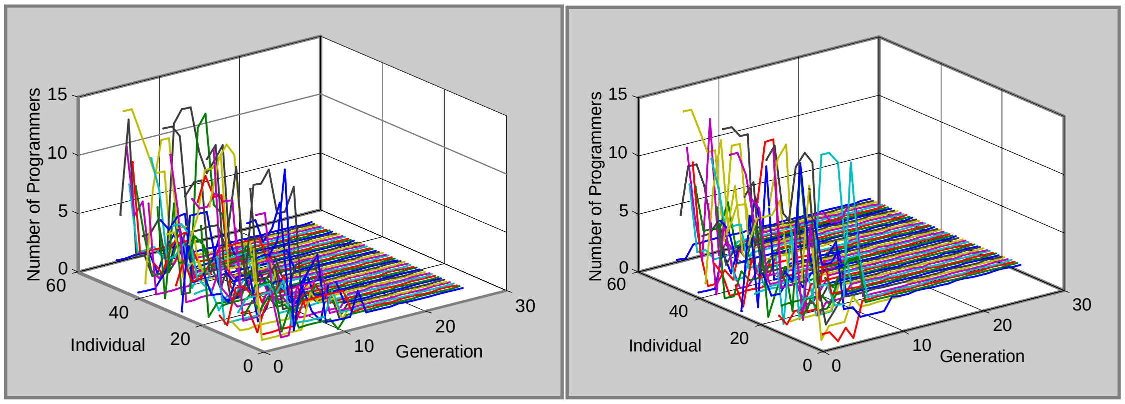


Figure 14: <u>Measuring Quality.</u> Single appraisal at project end (left) compared to three appraisals during every project (right). (Better performance is fewer number of programmers)

The run on the left, with a single evaluation at the proper time shows that the optimal number of programmers is now one programmer, rather than fifteen as it was when our criterion was speed. Why? Finding bugs is modeled as a simple exponential drain of he stock of undiscovered bugs. Consequently, the faster the code is written, the greater the level of undiscovered bugs. Hence, the criterion of shipping when the project is *apparently* done causes us to ship with more bugs, the faster we write code.

The run on the right shows that improvement stalls, and a premature consensus emerges when the appraisal period becomes so short that some appraisals are based on highly inadequate information. The plot on the right actually represents a particularly strong test: The mid-project appraisals contain good information if the project finishes early as it will if there are many programmers working on it. Only when comparing runs that are already pretty good, will there be information-free appraisals. Even so, some people are promoted who shouldn't be and some people are demoted who shouldn't be. As a consequence hierarchical position no longer points dependably to people who have been successful.

The failure shown in the plot is not caused by undesirable incentives – incentives and motivation are not represented in our model. Rather the failure is due to a pointing and pushing device that is less likely to spotlight the right teacher.

4.5. Flavors of the month. If the time between appraisals against standard is important to evolution, so is the time between *changing* the standard. For pointing and pushing to work, the mechanism needs to point and push in some direction. Many employees believe that their own companies suffer from inconsistent direction. Like Baskin-Robbins' monthly ice-cream selections, they believe that managerial focus shifts from one criterion to another in a flavor of the month rotation. Surprisingly, good policies can still evolve in a situation of rotating objectives, as long as the shifts occur with the proper timing.

We use an admittedly extreme version of our model to investigate policy rotation. There are several changes in our new runs. The most important is that we now have two potential standards by which to evaluate performance. As before, we can still gauge performance by the time it takes to ship a product (although we use the modifications discussed in footnote 14), but in addition we can also evaluate managers by the number of bugs remaining in the release at the shipment time.

For comparison purposes, we show a run in which managers get promoted for shipping products fastest side by side with a run in which managers get promoted for shipping fewer bugs.

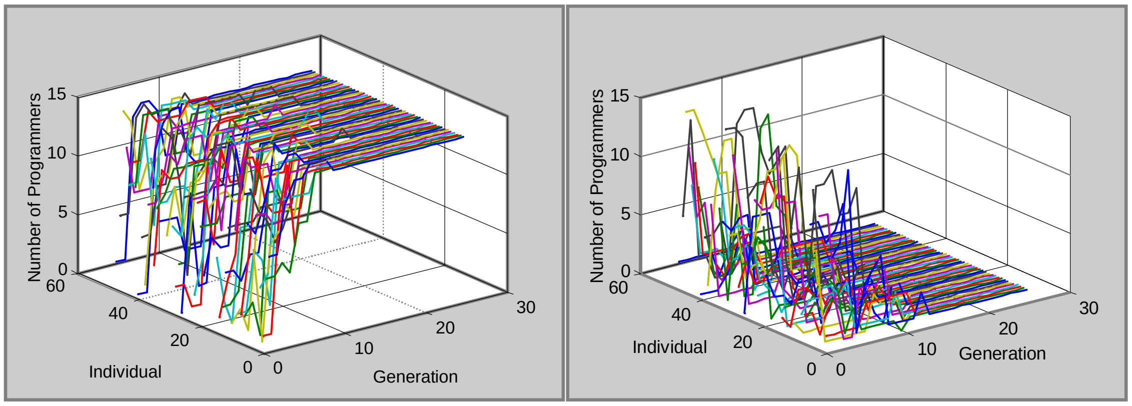


Figure 15: Two promotion criteria: Timeliness (left) and Quality (right)

As before, the best number of programmers is 15, if you are going for timeliness (the run above converges at 14 due to premature consensus. In contrast, the best number of programmers is 1, if you are going for quality (i.e. fewer bugs) for reasons discussed above.

In the plot below, managers are first evaluated in terms of the time it takes them to ship the product. Then (perhaps after customer complaints) on the basis of bugs in the shipped product. Then (perhaps after being pummeled by a speedy competitor), the focus company's focus shifts back to time to ship product. The run continues in this way with the criterion of success sifting from timeliness to bugginess after each release.

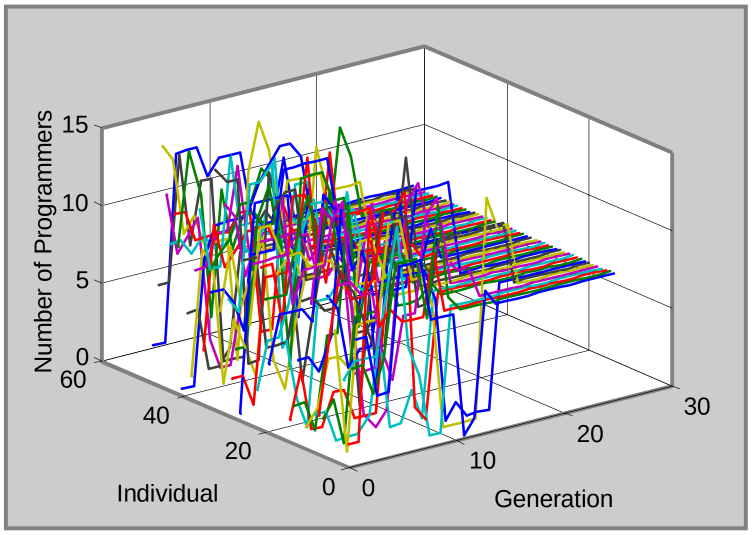


Figure 16: Criterion of success rotates between timeliness and quality

This plot may look like learning drift and random consensus (cf Figure 5), and one might be tempted to conclude that switching criterion is damaging to evolution. But, consider the similar looking plot below in which the measure of success *simultaneously* considers timeliness and quality. In this is run, the criterion of success does not change; the company has decided to value *both* timeliness and quality. The result is an evolution toward a middle ground. And, that is the same thing that happens when the criterion of success switches rapidly back and forth between the two policies.[15]

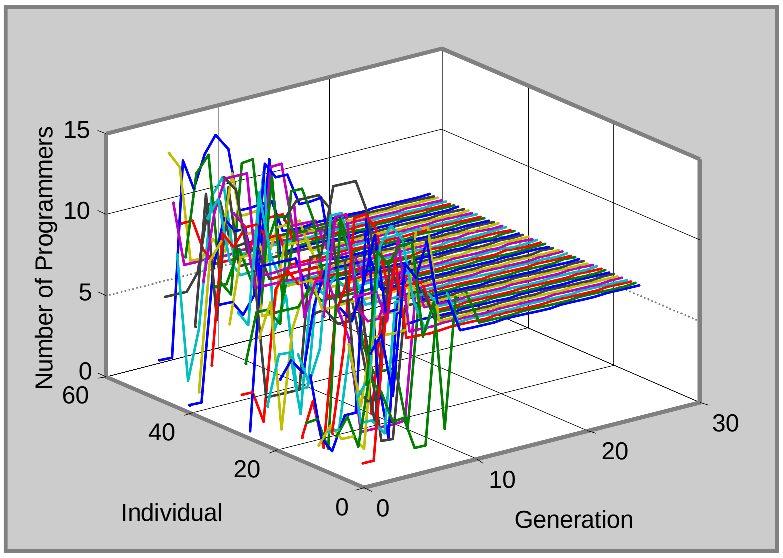


Figure 17: Criterion of success combines quality and focus.

[15] An intriguing difference between the two plots is that the simulation where the criterion switches converges more slowly. We do not know yet whether this difference reflects a true tendency for slower convergence when corporate focus rotates.

Random learning does not necessarily result whenever shifting focus plagues a company, as long as the focus rotates and does not continually move on to *new* criteria and as long as the shifts occur with about the same frequency as that with which information about outcomes becomes available. (In the run for Figure 16, the criterion of success is applied each "generation" after the outcomes are reported). The situation changes, however, if the corporate focus shifts, but very infrequently.

The plot below looks quite different from Figure 16, and yet the underlying run is similar. As before, focus changes from timeliness to quality. The difference is that the shift occurs later, and once the shift is made it is held. In this case, the shift in focus comes after consensus has started to form – that is after information has been lost. Consequently, after the shift, the evolutionary process works on a much more homogeneous population. Evolutionary movement is very slow after the shift and consensus comes much too soon.

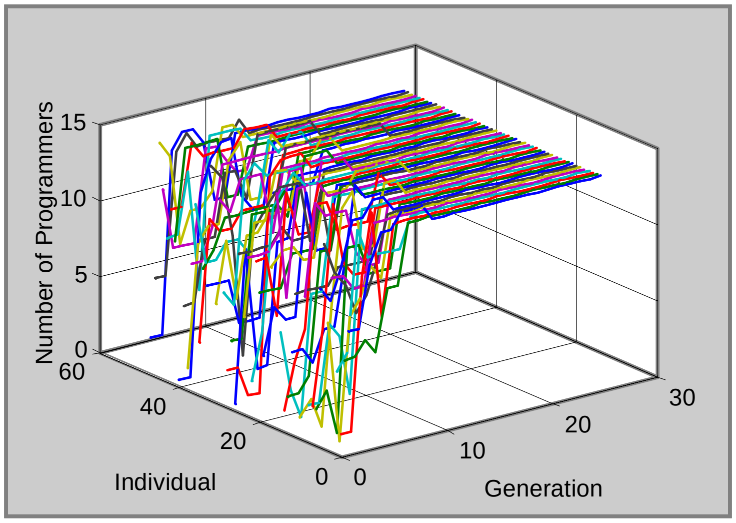


Figure 18: Criterion of success changes once, after consensus has started to form

A shift in criteria after consensus has begun to form will start a new evolutionary direction, but this time starting from a more homogenous population. The result is slower evolution in the new direction than in the original direction. Premature consensus is more likely.

Discussion. We have presented an evolutionary theory of policy formation in organizations. The theory explains how policies arise by a process similar to biological evolution.

Organizational evolution depends on people learning policies through imitating others. No other animal imitates as well as humans, or teaches as well as humans. And consequently, no other species is as well endowed to create organizations that can evolve socially. But even among humans, organizational evolution – that is the steady creation of better and better policy – is not assured.

Evolution requires direction. In biology, the direction is provided by natural selection. In organizations, pointing to people involved in successful efforts and pushing other managers to imitate them provide direction. Lack of direction in the natural world gives rise to genetic drift. In the organizational world, lack of direction results in learning drift.

Evolution proceeds by a process of combining and recombining parts of relatively successful policies into even more successful policies. As the best policy components spread, the “worst” policy components are lost. These twin activities really define a race between a movement toward fitter combinations and the loss of policy components. If the movement toward “fitter” policies occurs too slowly, or the loss of policy components occurs too rapidly, managers will reach policy consensus prematurely – before the “best” one has actually emerged.

We have highlighted five particular characteristics that influence whether policies will tend to get better and better: The quality of the pointing and pushing mechanisms, the amount of innovation, the number of organizational units, the frequency of performance appraisals, and the frequency with which appraisal criteria change. Each one of these characteristics is a “handle” that changes the balance between improvement and consensus. There are undoubtedly other characteristics that influence the emergence of good policy. We anticipate that each of these will also be a handle on improvement or consensus.

Poor policies are not responsible for everything that goes wrong in human organizations. Even well managed companies are subject to warehouse fires and fickle customers. But, the cost of poor policy is still enormous, and poor policy represents what we can *solve* about organizations. Most efforts aim at fixing poor policies after they have already arisen. We’d like to suggest that an additional and potentially powerful approach to improvement is to help companies evolve healthy policies from the start.

References

Abdel-Hamid, T. and S. Madnick (1991). Software Project Dynamics: An Integrated Approach. Englewood Cliffs, NJ, Prentice Hall.

Bandura, A. (1986). *Social Foundations of Thought and Action*, Englewood Cliffs: Prentice Hall.

Cooper, K. (1980). "Naval Ship Production: a Claim Settled and a Framework Built." Interface **10**(6).

Cooper, K. G. (1993). "The Rework Cycle: Vital Insights Into Managing Projects." IEE Engineering Management Review **21**(3): 4-12.

Diehl, E. W. and J. D. Sterman (1995). "Effects of Feedback Complexity on Dynamic Decision Making." Organizational Behavior and Human Decision Processes **62**(2): 198-215.

Forrester, J. (1968). "Industrial Dynamics - After the First Decade." Management Science **14**(7): 398-415.

Forrester, J. W. (1961). Industrial Dynamics. Portland, OR, Productivity Press.

Gillespie, J. (1998). Population Genetics. Baltimore, Johns Hopkins University Press.

Goldberg, D. E. (1989). Genetic Algorithms in Search, Optimization and Machine Learning. Reading, MA, Addison Wesley Publishing Company, Inc.

Heyes, C. and B. Galef (1999). Social Learning in Animals: The Roots of Culture. San Diego, Academic Press.

Hines, J. (1998). "Five Rules for Evolutionary Management." System Thinker.

Hines, J. and J. House (1998). Harnessing Evolution for Organizational Management. International Conference on Complex Systems, Nashua, New Hampshire.

Hines, J. H. and J. L. House (1999). Policy Evolution within an Organization. 2000 Design and Manufacturing Conference, Vancouver, B.C.

Holland, J. H. (1992). Adaptation in Natural and Artificial Systems. Cambridge, MA, MIT Press.

House, J., A. Kain, et al. (2000). Evolutionary Algorithm: Metaphor for Learning. GECCO, Las Vegas.

Kampmann, C. E. and J. D. Sterman (1992). Do Markets Mitigate Misperceptions of Feedback in Dynamic Tasks? Proceedings of the 1992 International System Dynamics Conference of the System Dynamics Society, Utrecht, the Netherlands, The System Dynamics Society.

Koza, J. R. (1992). Genetic Programming. Cambridge, MA, MIT Press.

Leach, D. R. F. (1996). Genetic Recombination. Cambridge, MA, Blackwell Science, Ltd.

Li, W.-H. and D. Graur (1991). Fundamentals of Molecular Evolution. Sunderland, MA, Sinauer Associates, Inc.

Meltzoff, A. (1995). "Understanding the intentions of others: Re-enactment of intended acts by 18 month old children." Developmental Psychology(31): 838-850.

Meltzoff, A. (1996). The Human Infant as Imitative Generalist: A 20-Year Progress Report on Infant Imitation with Implications for Comparative Psychology. *Social Learning in Animals: The Roots of Culture*. San Diego, Academic Press.

Morecroft, J. and J. Hines (1985). Strategy and the Design of Structure. Strategic Management Society Conference, Barcelona, Spain.

Nei, M. (1987). Molecular Evolutionary Genetics. New York, Columbia University Press.

Paich, M. and J. D. Sterman (1993). “Boom, Bust, and Failures to Learn in Experimental Markets.” Management Science **39**(12): 1439-1458.

Senge, P. M. (1990). The Fifth Discipline: the Art and Practice of the Learning Organization. New York, Doubleday/Currency.

Smith, J. M. (1989). Evolutionary Genetics. New York, Oxford University Press.

Sterman, J. D. (1989). “Modeling Managerial Behavior: Misperceptions of Feedback in a Dynamic Decision Making Experiment.” Management Science **35**(3): 321-339.

Sterman, J. D., N. Repenning, et al. (1996). The Improvement Paradox: Designing Sustainable Quality Improvement Programs. 1996 International System Dynamics Conference, Cambridge, Massachusetts, System Dynamics Society.

Sterman, J. D., N. P. Repenning, et al. (1995). Unanticipated Side Effects of Successful Quality Programs: Exploring a Paradox of Organizational Improvement. System Dynamics '95, Tokyo, International System Dynamics Society.

Summers, D. (1996). The Biology of Plasmids. Cambridge, MA, Blackwell Science.

Whiten, A. and D. Custance (1996). Studies of Imitation in Chimpanzees and Children. *Social Learning in Animals: The Roots of Culture*. San Diego, Academic Press.