Distributed Computing Architecture/e-Business Advisory Service

Agents (Part 2): Complex Systems

Executive Report, Vol. 3, No. 6

by James Odell

Complexity is all around us. It is part of our life; it is the nature of life. Complexity is caused by the collective behavior of many basic interacting agents. Such agents can produce everyday phenomena such as ant colonies, traffic jams, stock markets, forest ecosystems, and supply chain systems. Complex systems, however, do not have to complicated. For example, the ant colony simulation shown in Figure 1 (see www.media.mit.edu/~starlogo) is not complicated. Each ant has three simple rules:

1. Wander randomly.
2. If food is found, take a piece back to the colony and leave a trail of pheromones that evaporates over time, then go back to rule 1.
3. If a pheromone trail is found, follow it to the food, then go to rule 2.

In Figure 1 (a), the ants are just emerging from the ant hill to begin their random walk. Eventually, an ant discovers a food source and returns some to the colony, leaving a trail of evaporative pheromones as shown in Figure 1 (b). Figure 1 (c) shows the ant colony well under way in retrieving food. Lastly, Figure 1 (d) depicts two very depleted food sources and one that is exhausted altogether.

Ant colonies, then, are not complicated. A system is complicated if it can be completely described in terms of its huge number of components. A system is complex if the system cannot be fully understood by analyzing its components; here, the interaction among the components also must be considered.

Often, complex systems cannot be fully understood, precisely because we do not fully understand the component (or agent) interaction. For example, we might be able to identify the components involved in the New York Stock Exchange, yet we cannot accurately predict when we will have a bull market or a bear market, nor when a market bubble will burst. In such situations, our knowledge of the interaction among the various components is not well understood.

This *Executive Report* focuses on complex systems and follows up on the idea of agents, which was first discussed in the *Executive Report* "Agents: Technologies and Usage (Part 1)," Vol. 3, No. 4.

COMPLEX SYSTEMS

The science of complex systems was first made popular by the Santa Fe Institute (SFI). This group of complex systems

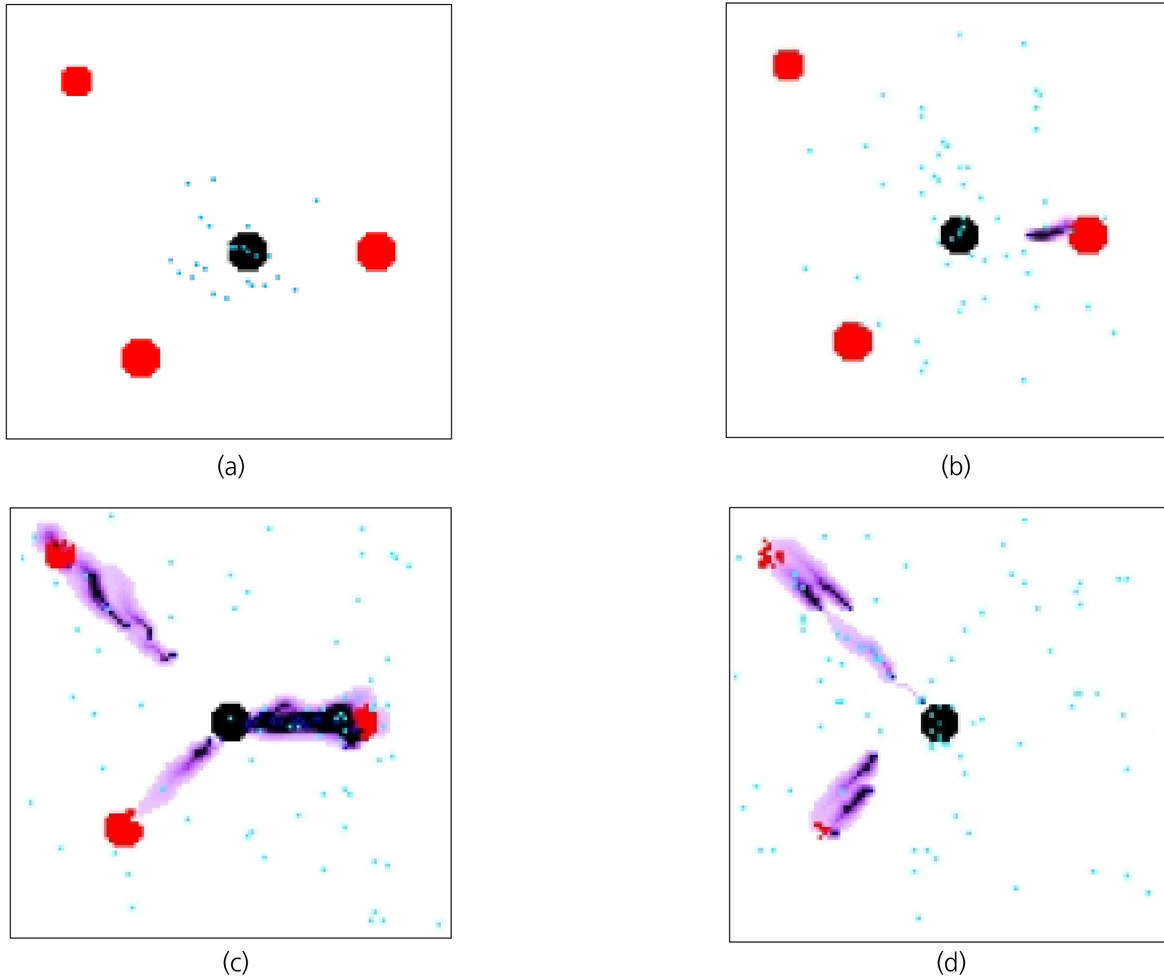

Figure 1 — Snapshots from an ant colony simulation.

theorists and practitioners offers the following definition:

> Complexity refers to the condition of the universe which is integrated and yet too rich and varied for us to understand in simple common … ways. We can understand many parts of the universe in these ways, but the larger more intricately related phenomena can only be understood by principles and patterns — not in detail.

This concept is important to us because the nature of business and IT systems are becoming more complex. For almost a century, our manufacturing model was reasonably stable. Rules for productivity, market dominance, and company success were well understood. However, the information age has turned previous “knowns” on their head. Today, no one can predict what or how rapidly new technologies will be developed. No one can accurately predict how supply chains will be affected or how customers

Distributed Computing Architecture/e-Business Advisory Service Executive Report is published by the Cutter Consortium, 37 Broadway, Suite 1, Arlington, MA 02474-5552. Tel: +1 781 641 5118 or, within North America, +1 800 964 5118, Fax: +1 781 648 1950 or, within North America, +1 800 888 1816, E-mail: consortium@cutter.com, Web site: www.cutter.com/consortium/. Client Services: Megan Nields, Tel: +1 781 641 5118, or Christine Doucette, Tel: +1 781 641 9768, E-mail: info@cutter.com. ©2000 by Cutter Consortium. All rights reserved. Unauthorized reproduction in any form, including photocopying, faxing, and image scanning, is against the law.

will respond. A business plan is only a guess.

Even small to medium-sized business are not insulated from this effect. Satellite communications, the Internet, and air transports commonly provide ways of rapidly moving corporate resources from place to place. All of us are now connected through a global market of online customers and suppliers, supply chain partnerships, and international franchise competition.

As Susan Kelly and Mary Allison suggest in *The Complexity Advantage*, businesses that don't understand and take advantage of the nature of complex systems thinking "will be at the mercy of an increasing number of sudden and unexpected shifts in the marketplace. As uncertainty grows exponentially with today's high rate of technological change and the fallout from it, so does the pressure of global markets" [1]. Many executives try to respond to this with yesterday's mindset and linear cause-and-effect thinking. Often, these responses intensify an already downward spiral. Despite well-conceived plans and well-intended actions, a company that operates without complex systems thinking will find itself unable to respond to the ever-increasing complexity of the business world.

Primary Issues

Common to any discussion of complex systems are several fundamental ideas.

First and most basic among these is *agents*. In complex systems, these are the autonomous entities that interact to carry out their particular tasks. Another fundamental concept is that these agents are *adaptive*. That is, the agents must be able to react to their environment and possibly change their behavior based on what is learned.

Complex systems are also characterized by *emergence*. Emergence is a coherent pattern that arises out of interactions among agents. For example, the process of an entire ant colony being fed was not programmed. It emerged from some very simple rules programmed into each ant. In other words, emergence is a byproduct of individuals — not a choreographed result. Emergent results can be good as well as bad and therefore must be considered when developing agent-based systems.

Successful emergent systems often exist between order and chaos. As with any organism or organization, being orderly or chaotic all the time would result in death. However, the area in between is necessary for continued existence and fitness.

Lastly, *nature* can teach us a lot about designing complex systems. It has been solving large combinatorial problems for billions of years. It makes sense, then, for us to consider notions such as parasitism, symbiosis, reproduction, genetics, mitosis, and survival of the fittest when developing our agent-based systems. For example, British Telecom is using the model of ants and pheromones in its call-routing network. Here, successful calls leave an equivalent of pheromones to guide future calls.

The ideas in the first item were discussed in "Agents: Technology and Usage (Part 1)." The remaining items are presented below. This overview will conclude with a comparison and contrast between agents and objects.

ADAPTATION

An adaptive agent is an agent that responds to its environment. There are four primary ways of adapting:

- **Reacting** — a direct, predetermined response to a particular event or environmental signal.
- **Reasoning** — ability to make inferences.
- **Learning** — change that occurs during the lifetime of an agent.
- **Evolving** — change that occurs over successive generations of agents.

Reactive Agents

In its simplest form, an agent can react with a direct, predetermined response to a particular event or environmental signal. Typically, the behavior of reactive agents is expressed in the form: WHEN event, IF condition(s), THEN action. Examples of such agents are thermostats, robotic sensors, and washing machines that use fuzzy logic. Although these kinds of agents may seem primitive, they are capable of achieving significant results. For example, the ant colony simulation described earlier consists of purely reactive agents — yet a whole colony is fed. The majority of the agents in our IT systems will consist of such agents.

Reasoning Agents

A reasoning agent can follow a chain of rules. Although these kinds of agents are often reactive, they have the added capability of making inferences. Reasoning agents can perform tasks such as network diagnosis or data mining. Reasoning is not a new capability. Expert systems have been around for quite a while. The difference here is that expert system rules can be encapsulated within agents. Furthermore, the agent can be designed to be proactive in its use of reasoning. For example, an agent does not have to wait for a report of a supply chain problem to perform diagnosis; it could perform preventive maintenance as well.

Learning Agents

Some agents can their change behavior based on their experiences. In other words, these agents can "learn." Learning agents don't need to have large brains, they just need the capacity to modify their actions as a way to improve their performance. This can be accomplished by simply programming them to weigh their decisions. Another common technique is to use neural networks, another device from the AI era that can be applied within agent-based systems.

Evolving Agents

In agent systems, changes can occur over successive generations of agents. For example, Darwinian-style evolution is a common technique. Here, agents can be bred using genetic algorithms, and then compete in a survival-of-the-fittest mode. In Lamarckian-style evolution, features acquired and information learned by an agent can be passed on to its offspring. The use of *memes*, or culturally transmitted information, is also a popular technique.

Adaptation Summary

The four primary forms of adaptive agents described above can be used singly or in combination. For example, a reactive agent's neural network could have been bred using genetic algorithms, or a proactive reasoning agent could learn by placing weights on certain decision points in its rules. At a minimum, however, the agent must be able to react to external stimulus.

Reactive and reasoning agents are fairly easy to construct and understand. Learning and evolving agents take more work to design, but the greatest drawback is that it can be very difficult — even impossible — to understand their decisions. Once a neural network has been trained to solve certain kinds of problems, there is almost no way to ask it how it came up with its solution to a particular problem. A similar problem exists with genetic algorithms. Here an agent's program code can be cross-bred with other agents' code until it can successfully accomplish a given task. Eventually, it may perform its tasks perfectly, but understanding how and why it works can be close to impossible, particularly for agents with thousands of lines of code that have been bred over thousands of generations.

In short, we can "teach" and "breed" agents now. The good news it that such a process can generate results faster and usually better than any human. The possibly scary news is that the results can occur without human intervention. In other words, we will be building agents that seem to have a "life" of their own. They can learn to buy, sell, bargain, fabricate, and make decisions for us — making us feel "out of control" [2]. We delegate to human agents all the time, but

delegating to software agents can take time to get used to, even though it is inevitable.

AGENTS AND EMERGENCE

Agents can work as noninteractive individuals or as a collective. When agents work as individuals with little or no interaction, what you get is just that: agents simply doing what they are asked to do. For example, a single "bot" agent sent out to find the cheapest airline fare can be expected to simply return with the requested information. As a collective, however, something new and different can result — something that is *more* than the sum of the individual participants.

The stock market, immune systems, and ant colonies are all examples of agents acting individually, yet from the interactions of these agents a new phenomenon arises. With the stock market, thousands of agents act independently to buy and sell shares of particular stocks and bonds. From this independent behavior, an organism-like product called the stock market emerges. In other words, the rise and fall of the market is not controlled by a central process; it results from agents interacting. The stock market crash of 1929 was a result of individual human agents — not a central controller. The crash of October 1987 partly resulted from individual software agents that buy and sell securities according to programmed rules. The stock market, with its crashes, temporary bubbles, and dead-cat bounces, is more than the sum of the parts; it is an entity in its own right. Such entities are called *emergent* structures.

Ant colonies are emergent structures that arise from individual ants acting interactively. The immune system emerges from the collective behavior of agents such as antigens, T-cells, B-cells, NK-cells, immunoglobulins, lymph nodes, and the spleen. Other examples of emergent structures include families, organizations, societies, markets, flocks of birds, and traffic jams. In IT systems, this can include supply chain, scheduling, trading-floor, and e-commerce systems.

Emergence

Emergence is the existence of a coherent pattern that arises out of interactions among agents. The diagram in Figure 2 illustrates this definition. Emergence embodies several properties:

- In emergent structures, agents organize into a whole that is greater than the sum of its parts. In other words, the parts alone do not result in emergent structures — their interaction is required. Single bot agents will not result in emergence; a multiagent environment, such as shop-floor operations systems, handled by multiple interacting agents can have emergent properties.
- Rules that are almost absurdly simple can generate coherent, emergent phenomena. For example, each ant in the ant colony described earlier has some very simple rules, yet a well-fed colony can emerge from these simple rules.

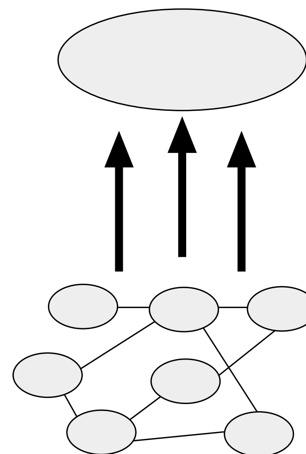

Figure 2 — Local interaction can give rise to global dynamics, creating a coherent structure.

Likewise, a purchaser or supplier agent can have simple rules resulting in an emergent inventory system.

- Instead of being designed from the top down, most emergent systems develop from the bottom up. A human engineer would tend to develop top-down. This approach can be useful under some circumstances. However, most living systems emerge from a population of simpler systems. Developing daily operational plans in a top-down manner work well for fairly predictable organizations. In less predictable environments, top-down plans can easily become obsolete in a manner of minutes. There, a bottom-up, agent-based approach could result in a very effective, emergent operation.
- Persistent emergent structures can serve as components of more complex emergent structures. In other words, hierarchies of emergent structures can be formed. This is how nature obtains scalability; any IT organization could employ the same mechanism.
- Agents and their emergent structure can form a two-way link. Agents can give rise to an emergent structure; the emergent structure can influence its component agents. For example, the stock market is an emergent structure of individual buyers and sellers, yet the rise and fall of such markets can affect the buying and selling habits of individual participants.
- Emergent phenomena are typically persistent patterns with changing components. The birds in a flock or the cars in a traffic jam can change, yet the flock and traffic jam phenomena remain. Likewise, the buyers and suppliers in a company's supply chain change frequently, and the participants in a scheduling system can differ on a daily basis.
- Collections of agents can be homogeneous or heterogeneous. Emergence can occur due to the interaction of similar agents. More often, though, it occurs as a result of different kinds of agents that function in a society or ecosystem. Large organizations employ heterogeneity by specializing corporate resources using different roles and business units.

These properties will be explored in more detail in the sections that follow.

Greater than the Sum of Its Parts

Simple-minded reductionism states that the whole is simply the sum of its parts and that each part can be studied in isolation. However, the parts alone cannot produce emergence. Emergent structures also require the collective behavior and interaction of its components. Emergent structures, then, are a *process* — and the essence of the process is its form, not its parts. Families, organizations, societies, financial markets, schools of fish, and traffic jams are all examples of this phenomenon. A hoard of noninteracting Web spiders will not produce an emergent structure. However, when an ecosystem of supply chain software agents can buy and sell goods and services over the Web, a "supply web" can emerge [3]. The supply web behaves like a financial market — which does give rise to an emergent structure. Agent-based modeling and bidding systems make this possible.

Simple Agent Rules Can Produce Emergent Structures

A common example of simple rules leading to emergence is a flock of birds. Each movement of a flock is so beautiful that it appears choreographed. Furthermore, the movements of the flock seem smoother than those of any one bird in the flock. Yet, the flock has no high-level controller or even a lead bird. Each bird follows a simple set of rules that it uses to react to birds nearby. In StarLogo's flocking simulation, the birds obey only three rules:

1. If you are far away from other birds, head toward the nearest bird.
2. If you are about to crash into another bird, turn around.
3. Otherwise, fly in the same direction as the bird next to you.

Using these three simple rules, no one bird has a sense of an overall flock. The “bird in front” is merely a position of a given bird. It just happens to be there — and will be replaced by others in a matter of minutes. Flocks of birds are not the only things that work like this. Beehives, ant colonies, freeway traffic, the Web, and the phenomenon of Silicon Valley are all examples of patterns that are determined by local component interaction, instead of a centralized authority. Complex behavior need not have complex roots.

Top-Down Versus Bottom-Up Approaches

If you have ever heard classical music or watched a ballet, you have no doubt realized that the performance was orchestrated or choreographed. The centralized, or top-down, development of these kinds of performances is both obvious and necessary. Many of the products we use in everyday life require top-down engineering to be effective. However, most of the emergent phenomena we experience do not occur as a result of top-down efforts; instead, they are the result of decentralized, or bottom-up, processes. For example, the flock of birds mentioned above emerges without an organizer and behaves without a coordinator. So, too, does the applause that follows a classical concert or ballet.

Today, many resource providers and manufacturers are exploring the possibilities of employing a decentralized approach. Many of these organizations are already adopting solutions that will replace their central, globally optimized operations with a distributed, self-organizing, local one. John Holland, a professor at the University of Michigan, is fond of pointing out that New York City maintains a two-week supply of food with only locally made decisions. Companies such as Boeing, John Deere & Company, and Detroit Edison are beginning to do this.

Both centralized and decentralized approaches are useful techniques. Using one technique and not the other limits the possibilities of a system. Often, our human bias toward centralization precludes the consideration of decentralized solutions. For example, three-year-old Rachel developed the theory that clouds rain when the thunder commands them [4]. At four, she developed a new theory: the clouds get together at night and decide whether or not it should rain the next day. People resist decentralization. When people see a pattern, they often assume a centralized control. This does not mean that centralized theories are wrong, it is just that they are not always appropriate:

- A central agent is a single point of failure that makes the system vulnerable to accident.
- Under normal operating conditions, a central agent can easily become a performance bottleneck.
- Even if it is adequately scaled for current operations, a central agent provides a boundary beyond which the system cannot be expanded.
- A central software agent tends to attract functionality and code as the system develops, pulling the design away from the benefits of agents and, in time, becoming a large software artifact that is difficult to understand and maintain.

Emergent Structures Can Themselves Be Components

One of the most difficult challenges for automated systems is scalability. Living systems provide some excellent examples of scaling up. In the physical systems leading up to life, for example, subatomic particles form atoms, and atoms cluster to become molecules in solid, liquid, and gaseous form. Continuing up this hierarchy, molecules can be organized to form organelles, organelles can group to form cells, cells can aggregate to form organisms, and so on. In other words, living systems and their components emerge in a hierarchy of interlocking mechanisms. In the domain of human organization, similar hierarchies occur, as illustrated in Figure 3. Here, economies emerge from markets,

©2000 CUTTER CONSORTIUM

which emerge from enterprises, and so on [5].

Emergence provides the mortar between the bricks to construct viable structures. Furthermore, the new structures can become building blocks for even larger structures — in which each level of the hierarchy is very different from the one before and the one after it. Such a hierarchy of interlocking mechanisms is also an appropriate technique for automated agents.

Agents and Their Emergent Structure Can Form a Two-Way Link

Applause occurs when spectators join in what appears to be spontaneous synchronized clapping. There is no conductor that coordinates this. When everyone starts, the clapping is totally unorganized; each person's tempo is wildly out of phase with the next person. Eventually, groups of people begin clapping at the same tempo. People in the audience sense the emerging rhythms and adjust their clapping to join it. The emerging applause rhythm grows even stronger and more people conform to it. Eventually, the entire audience is clapping in a synchronized pattern. This entire process can take place in a matter of seconds with even thousands of individuals.

We have been exploring how the local interaction and behavior of agents can produce global dynamics of emergent structures. In the example of spectator applause, the interaction of humans in the audience produced the dynamics of applause. However, there was another phenomenon occurring in this example: individually, the spectators adjusted their applause rhythm based on the applause that they heard. In other words, local interaction can give rise to global dynamics, and the global dynamics, in turn, can influence the local interaction.

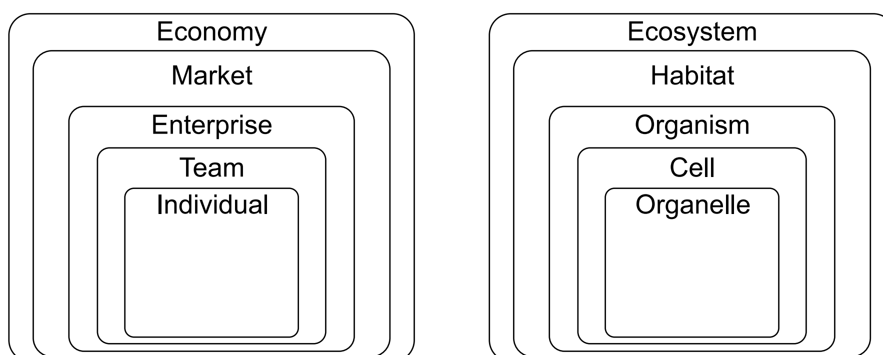

Figure 3 — Parallels between business and biology.

Such an effect is also evident in business systems. The stock market both results from and affects the buyers and sellers of securities. The auto parts market setup as a joint venture of General Motors (GM), Ford, and DaimlerChrysler, will result from and affect the buyers and sellers of automotive parts. As illustrated in Figure 4, emergent structures can be linked to their local agent interaction with the following results:

- This link influences the boundary conditions of the local agents.
- Local agents can then adjust to the presence of the global dynamics.
- Consequently, the conditions under which the agent behaves might change.

Emergent Structures Can Have Components that Change

As stated earlier, the birds in a flock or the cars in a traffic jam can change, yet the flock and traffic jam phenomena remain. The same applies to the stock market and supply chain webs. Just because an emergent structure exists and is stable does not mean that its components cannot change over time. Each of us replaces all the atoms in our body every three years, yet we are still considered to be the same being. Many Silicon Valley companies have more turnover than this, while still being recognized as the same organization. Change in the

underlying agent population is not required for emergence, but it is a common phenomenon.

Homogeneous and Heterogeneous Collections of Agents

Some emergent structures consist of a single kind of agent; for example, a flock of birds consists only of birds. Here, each agent plays the same role. Homogeneous agent collections can still have a single kind of agent, yet its agents can play different roles. For instance, an ant colony can have ants that play different roles. An ant can be a patrolling ant that guards the nest, a nest-maintenance worker, a forager, a brood-care worker, and so on. Furthermore, an ant can change its role depending on the requirements of the colony. For instance, a nest-maintenance worker can become a forager or a patrolling ant when the need for food or security becomes more important.

Heterogeneous collections of agents also play different roles because they contain different kinds of agents. The major difference is that in a heterogeneous collection, agents are different in both structure and behavior. For example, the immune system emerges from the collective behavior of various kinds of agents, such as antigens, T-cells, B-cells, NK-cells, immunoglobulins, lymph nodes, and the spleen. T-cells and B-cells not only play different roles, but their structure and behavior is also different enough to be considered heterogeneous.

When we construct complex business systems, we need to think of agents as functioning as a society or ecosystem. In designing such systems, we need to consider how we can effectively employ homogeneous and heterogeneous agents.

Emergence Conclusion

When constructing agent systems, emergence is an important concept to consider. On the one hand, emergence is something that can happen to you without your consent or predictive ability. This can be good or bad. On the other hand, as a system developer, you can try to “design in” the emergence that you want. In other words, you can try to design the agents in such a way that the desired structure emerges. In summary:

- You control the action of the parts, not the whole.
- You act as a designer, but the resulting pattern is not designed.
- Self-organizing patterns are created without a central designer.
- You must have enough agents acting in parallel to get a “critical mass.” A colony of 10 ants will not suffice.
- The parts must be interacting — parallelism is not enough. Without interactions, interesting colony-level behaviors will never arise.
- Remember: a flock is not one big bird, and a traffic jam is not just a collection of cars.

BETWEEN ORDER AND CHAOS

Nature has moments both of order and chaos. Interestingly enough, those forms of nature

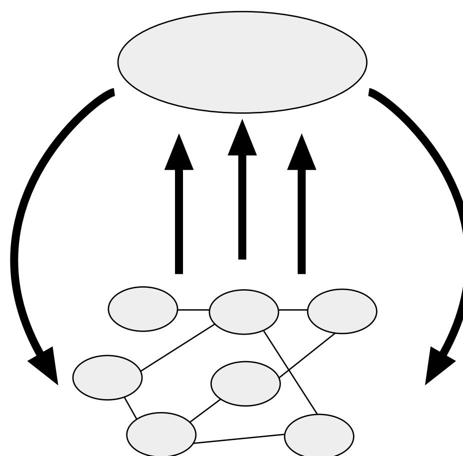

Figure 4 — Local interaction and global dynamics can influence each other.

that are considered most fit actually reside someplace in between. In fact, many consider this in-between state a necessary property of emergence for nature. Such a phenomenon also applies to business and software agents.

Basic Agent Behavior

Before we discuss order versus chaos, this section gives some brief technical background. One of the earliest forms of agents is called a *cellular automata* (CA). The idea was originally conceived by the Polish mathematician Stanislaw Ulam in the early 1950s, and further developed by John von Neumann and Arthur Brooks. Basically, a CA consists of a lattice of cells, or sites. Each cell has a state whose value is commonly expressed as 0 or 1, black or white, on or off, or a color selected from a set of colors. At discrete "ticks" of the CA clock, this value is updated according to a set of rules that specifies how the state of each cell is computed from its present value and the values of its neighbors.

The most familiar example is John Conway's game, Life. As described in the October 1970 issue of *Scientific American*, only a checkerboard and an ample supply of markers are needed. The rules of Life are simple:

- A dead cell (state 0), with exactly three of its eight immediate neighbors alive (state 1), is born. Under the right conditions, the cell comes alive.
- A living cell with two or three living neighbors remains alive; that is, the cell stays alive when nurtured by its neighbors to the right extent.
- All other cells die (or remain dead) due to overcrowding or loneliness.
- Each cell is updated once per time period.

The checkerboard rules represent the laws of physics (or life), and, although the cells themselves are not mobile, an amazing amount of behavior emerges. Figure 5 (a) depicts how a CA society can die out over three generations. Figure 5 (b) shows how a society can form a fixed configuration. Lastly, Figure 5 (c) illustrates how some patterns oscillate indefinitely.

Classifying Agent Behavior

Over the long run, CA societies have similar kinds of emergent behavior. The patterns shown in Figure 6 illustrate the four classes of behavior identified by Stephen Wolfram in 1983 when he was at Princeton's Institute for Advanced Studies. Class I societies are those that exhibit a static, or *limit point*, behavior. Figures 5 (a) and 5 (b) are examples of this class, because the lattice will not change after generation 3. Class II societies exhibit periodic, or *limit cycle*, behavior, which is the indefinite oscillation depicted in Figure 5 (c).

Class I and II can be considered one extreme of CA behavior because everything is predictable and orderly. Class III, on the other

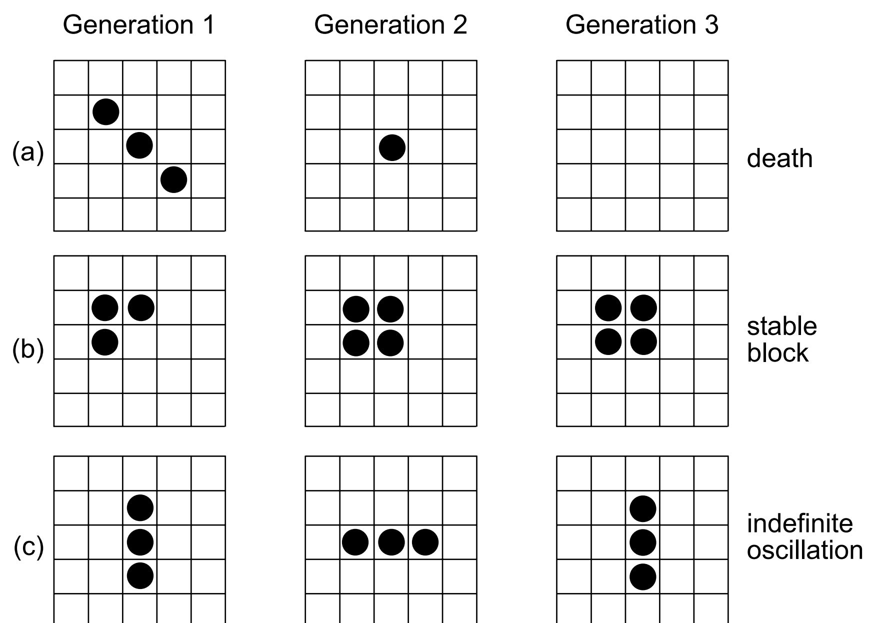

Figure 5 — Some examples of Life patterns.

hand, is aperiodic, or chaotic; that is, its structures display no obvious order or uniformity. In between these extremes is a mysterious and complex class of behavior: Class IV. Such automata exhibit considerable local organization, yet also have areas of irregular behavior. In other words, Class IV automata are someplace in between the two extremes: they exhibit orderly behavior as well as some chaotic behavior [6]. (Images in Figure 6 are courtesy of Andrew Wuensche, generated using his software “Discrete Dynamic Lab” from www.santafe.edu/~wuensch/ddlab.html.)

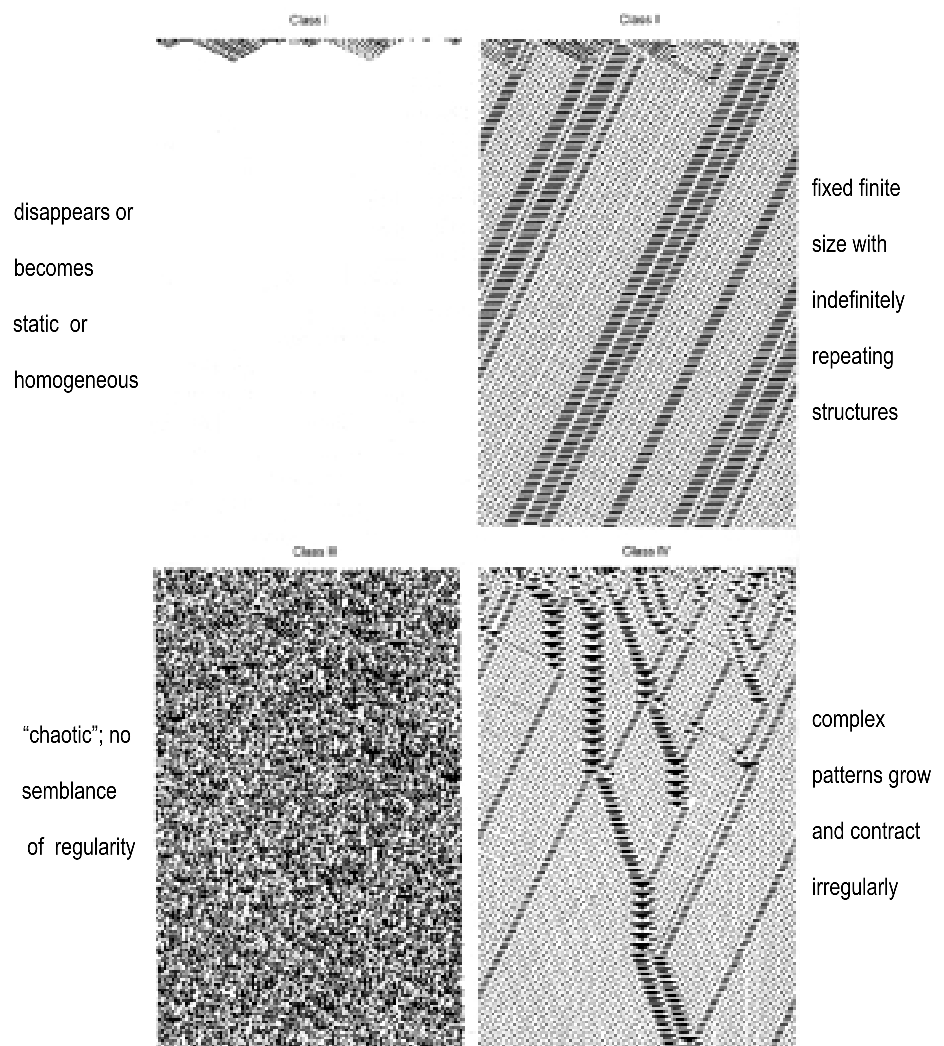

Figure 6 — Wolfram’s four classes of long-run behavior.

Order Versus Chaos

Cellular automata offer a way to model natural and artificial processes, such as modeling crystallization, complex fluid flows, chemical reactions, and hardware architecture, yet CA involves an elementary form of agent. Imagine the kinds of systems that can be built with agents that are mobile and have sophisticated forms of communication and interaction. Such agent systems provide not only a richer way of modeling natural and artificial processes but also a way of *implementing* such systems.

Such mature agents systems are subject to the same Wolfram behavior. You can build agent systems that are orderly (Class I and II), and such orderly behavior is appropriate for some kinds of systems. However, when agents are expected to learn and change their behavior, an orderly system discourages change. In a business example, all the jobs in an organization would be subdivided so that employees have no latitude and only do the job for which they are hired. For an automated supply chain system, the results would always be predictable. On the surface, such predictability would seem to be a good thing. However, when the business landscape changes (as it often does), the supply chain operation would no longer suit the organization’s needs. Instead, it would be predictably wrong. In both of these scenarios, everyone would benefit if the individual agents had the freedom to change. In short, orderly agent systems should become more fluid — and a bit closer to chaos.

Conversely, if agents are deep in a chaotic regime (Class III), they can never get the job done. For

©2000 CUTTER CONSORTIUM

example, employees who do not know what they're supposed to do often end up working at cross purposes. A supply chain system would not be able to deliver the right product to the right person at the right time. In both of these scenarios, if the individual agents could have tighter connections with fewer individuals, a greater degree of stability would be introduced. Chaotic agents, then, should become less fluid by adapting to what other agents are doing, resulting in aggregate behavior. This means pulling back from chaos.

The Edge of Chaos

Neither order nor chaos seems to be the best place for complex systems — whether their agents are implemented using software, hardware, machines, or people. Instead, such agent systems need to be somewhere in between. With too much order, the system stagnates and dies in the face of new competition that needs to be only a little bit better. With too much chaos, the system will not survive because it cannot make useful products. The edge of chaos is, on average, where fitness is best (see Figure 7). Such systems can exploit what they have learned and extend that learning through exploration.

Complex systems (including both living and business systems) are characterized by perpetual novelty. This approach can be scary: things can get out of control, and errors will be made. Yet without this kind of approach, there will be no change — only status quo. To talk about complex adaptive systems being in equilibrium is meaningless because the system never gets there. It is always unfolding, always in transition. If a system ever reaches equilibrium, it is not just stable — it is dead.

Now, I am not suggesting that such complex systems be built immediately. In fact, this would probably result in chaos itself. Complex systems should be built simply at first, initially placing any edge-of-chaos processing with human agents rather than automated agents.

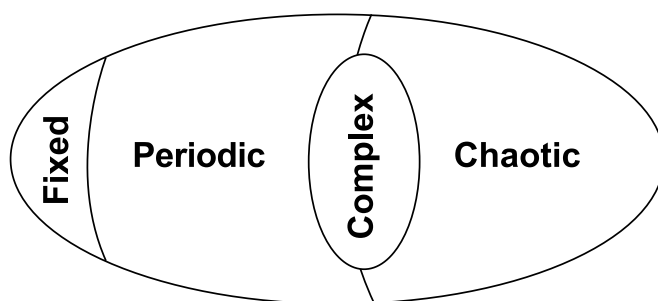

Figure 7 — Complex systems poised between order and chaos are best able to carry out ordered yet flexible behaviors.

The reasons for this are technical and psychological. Technically, we do not yet fully understand how to build complex systems that function properly. We lack both a systematic methodology and industrial-strength agent-system toolkits. Psychologically, living on the edge of chaos can make us uncomfortable. And when we must delegate our tasks to automated agents, we will feel even more out of control. It's bad enough when people are intimidated by their home appliances. What will happen when automated agents choose the articles we read, automatically answer our mail, and schedule appointments? On top of this, imagine how uneasy we will feel when automated agents begin making critical business decisions and acting on them. Confidence and understanding come slowly.

Edge of Chaos Conclusion

Stability is something valued in accounting and payroll systems. Nevertheless, the next generation business systems should be operating on the edge of chaos. Order entry, inventory control, and supply chain systems are particularly appropriate. These are systems whose agents are people, machines, and software. To work effectively, these agents must work together as a living

system — requiring flux, change, and the forming and dissolving of patterns.

Complex systems theory points us away from isolated units and toward interactions between individuals and their environment. Strategy focuses on the management of volatility, not the achievement of specific goals. Growth comes from agent relationships and rules, rather than through a significant increase in size. Opposing thoughts or points of view are held simultaneously. Mild instability is encouraged. Build something workable, rather than "optimal." Developing complex systems is not for the fainthearted, which applies to both executives and IT system developers. We need to unleash our software and let it grow and learn like any living system. Only then can our systems mature beyond our limitations — and exceed our expectations.

As Petruska Clarkson, a psychologist and chartered UK consultant put it, "A greater kind of courage and a different psychology is now required — to be willing to let go and experience the creativity, innovation, and disturbance which comes about when we risk the outer boundaries of trying to maintain a balance and the excitement of living, developing, and coaching at the edge of chaos." Learning will perhaps ultimately prove less valuable in the third millennium than the skill and attitudes of unlearning — in the same way that knowing what to do may become far less important than knowing what to do when you no longer know what to do.

DESIGNING AGENTS: USING LIFE AS AN ANALOGY

Analogies enable us to use one concept in place of another to suggest a likeness. They are particularly useful for introducing new ideas to an existing area, thereby expanding and enriching it. Using life as an analogy is already quite common when discussing agents, because agents can appear to interact in a lifelike manner. As such, we can think of agents as being subjected to stress from environmental pressures, resource shortages, and restriction of growth. We can imagine them with the ability to evolve their behavior by developing ways of coping with such stresses. Some agents will survive and succeed by growing, increasing their ability to command resources, and reproducing. Those that fail will shrink and will either be replaced, be absorbed, or die [7].

In short, we can use life as an analogy to develop agent-based systems, whether the agents are software, hardware, equipment, corporate entities — or even people.

Agents and Scalability

One of the most difficult challenges for automated systems is scalability. Here, life as an analogy brings with it many useful concepts, including some excellent examples on how to scale up. As we said earlier, in the physical systems leading up to life, for example, subatomic particles form atoms, and atoms cluster to become molecules in solid, liquid, and gaseous form. Continuing up this hierarchy, molecules can be organized to form organelles and cells, cells can aggregate to form organisms, and so on (see Table 1) [8]. In other words, living systems and their components emerge in a hierarchy of interlocking mechanisms.

In a general sense, complex systems are large and intricate systems involving active, autonomous agents. Such a hierarchy

Table 1 — A Hierarchy of Interlocking Mechanisms

System (Science)	Typical Mechanisms
Nucleus (physics)	Quarks, gluons
Atom (physics)	Protons, neutrons, electrons
Molecule (chemistry)	Bonds, active sites, mass action
Organelle (microbiology)	Enzymes, membranes, transport
Cell (biology)	Mitosis, meiosis, genetic operators
Multicellular organism (biology)	Morphogenesis, reproduction
Social group (biology)	Individuals, social relationships
Ecosystem (ecology)	Symbiosis, predation, mimicry

is a necessary — and some say, a natural — occurrence for complex systems. Without such a hierarchy, scalability would not be possible. Life would not be possible. Without a more complex formation, 10^{50} subatomic particles floating around our universe would just be 10^{50} subatomic particles floating around. Individually, they are subatomic particles and no more. To create something more complex, there must be a way to produce a new structure that is more than just the sum of its particles.

However, these new aggregate formations cannot become too large. Such structures can easily become unstable and collapse under their own weight; for example, increasing the size of a molecule a trillion times to produce something substantive is as impractical as building sand castles a mile high.

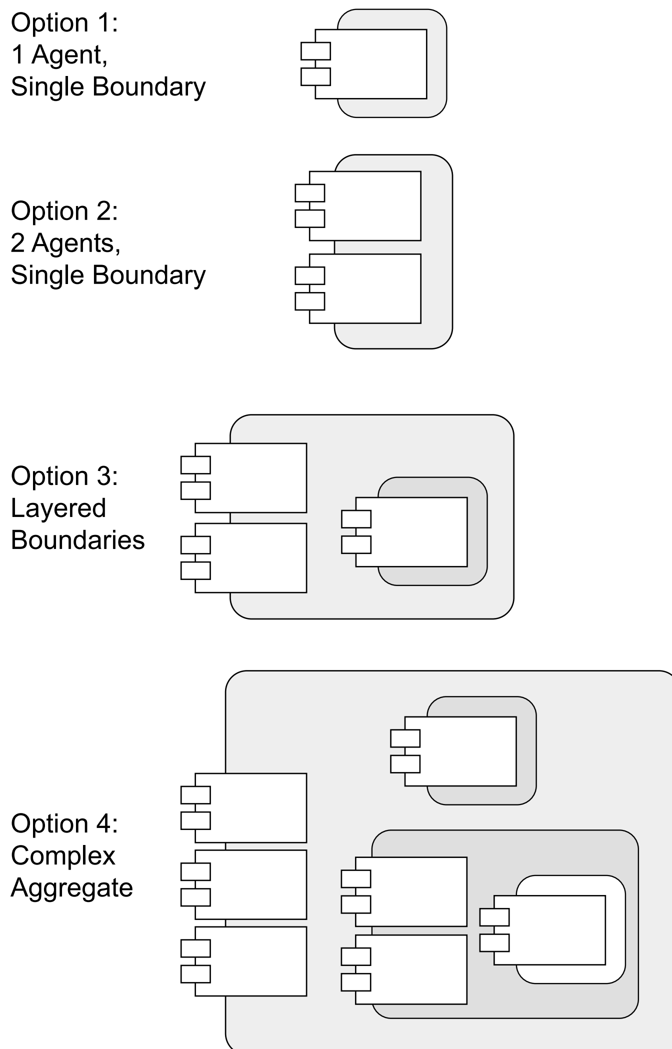

Figure 8 — Some forms of agent aggregation.

Sheer numbers without organization are unmanageable, and bigger is not necessarily better. It is life's hierarchy of interlocking mechanisms that provides the right mortar between the bricks to construct viable structures. Furthermore, the new structures can become building blocks for even larger structures in which each level of the hierarchy is very different than the one before and the one after it. For example, hydrogen and oxygen have very different properties than a water molecule which comprises them. A cell has a different structure and behavior than the molecules and organelles that comprise it, and so on. Why can't the same approach work for automated agents?

Aggregate Agents

Agents can be aggregated to form variable structures. These aggregates can be colonial in nature (such as sponges and coral reefs) or metazoan (that is, multicellular animals). Agents that adhere to one another can behave in a unified manner and still maintain their autonomy. Automated agents might choose to aggregate for various reasons, such as protection, resources, or improvement. In other words, the agents might decide that they are better off together than apart. Furthermore, the agents may adapt to serve the aggregation as a whole.

Figure 8 depicts several aggregation options [9]. The first option, of course, is that there is no

aggregation. The second option is that two or more agents can aggregate as a single unit. The containment boundary indicates that the entire construct can be treated as an agent in its own right. In option 1, the agent and the boundary are the same. In option 2, the boundary can be treated as an agent separate from the two agents it contains. The term *boundary* is chosen because it also provides an interface barrier much like that of a cell membrane. Agents on the membrane (as in option 2) are still accessible from the outside as individuals while benefiting from the proximity to chosen neighbors. In contrast, agents that are contained completely within the boundary, as illustrated in option 3, are encapsulated. Only those agents on or immediately *within* a boundary may communicate with encapsulated agents. However, the boundary itself may have specialized rules that permit the passage of agents through the boundary — either into or out of the agent. In fact, the membrane could be constructed to allow implicitly the passage of "substrate" that it does not see or care about. Granted this "breaks" encapsulation, but that's life. In a complex aggregate configuration (option 4), the nesting continues.

As we saw in Table 1, this is a common phenomenon: molecules emerge as aggregations of atoms, cells emerge as aggregates of molecules and organelles, and so on. For the world in general, it is common to see building blocks at one level combining to form building blocks at a higher level. At each level, new structures form and engage in new emergent behaviors.

In object orientation, this is consistent with how components are handled. Each component has its own interface. Those classes and components contained by the component are encapsulated; that is, they are not directly accessible by the external classes and components. Although there is no equivalent object-oriented (OO) support for option 2, it just means that the component interface includes the interfaces of those classes on the component boundary. In business, autonomous agents can be grouped into aggregates. At the Industrial Technology Institute, for example, Van Parunak (www.erim.org/~vparunak) suggests that sensor and actuator agents can be grouped into a machine, several machine agents can form a workstation, and workstations can aggregate into a manufacturing cell. (The automotive company case study in "Agents: Technology and Usage (Part 1)" treated manufacturing cells in the same manner.) In other words, grow by chunking instead of starting with large, complex agents. Favor smaller, specialized agents over more general ones. Small individual agents are easier to construct and understand than monolithic ones, and, if they fail, their impact will be minimal.

AGENTS AND DISTRIBUTED CONTROL

Distributed control is also a fundamental mechanism of life-forms. It provides an alternative to having a single elaborate control center directing every single task, by having multiple structures that specialize in their own subtasks. Furthermore, each of these structures may consist of many substructures, offering a finer degree of specialized control.

Distributed control of any complex system has many advantages. This is especially true when system components are widely dispersed, as in a communication, transportation, or banking network. Completely centralized systems require two-way communications links with all components. In any situation subject to rapid change, a completely centralized control requires high bandwidth communication links, a powerful central computer, and an elaborate operations control center. However, all of these are subject to disruption at any time by system bugs, natural disasters, espionage, or stress-related events. In situations where fast response and rapid recovery are important, distribution of control is usually preferable. Here, as much control as is practical is delegated to the local level. This way, when a failure occurs, each component can act as an independent agent. If these agents have adaptive capabilities, they can organize themselves and

make efficient use of whatever resources remain.

An even richer model involves autonomous adaptive agents that partly cooperate and partly compete with each other in their local operations. Industries that have multiple ownership and management often operate with a minimum of regulation — while depending on the goodwill and cooperation of all parties. Dr. Martin Wildberger of EPRI in Palo Alto, California, USA, (mwildber@epri.com) has demonstrated the feasibility of these kinds of agents in computer simulations of a deregulated power industry.

Agent Sensors and Effectors

Life-forms can sense their environment via an assortment of stimuli, and they can also effect changes in their environment. Agents, too, can have *sensor* and *effector* mechanisms. There are several ways agents can "sense."

Events

Events are changes in the environment that might be noteworthy to an agent. They can be sent directly to the agent on a broadcast or subscription basis or be directly observable by the agent in its network interaction. A broadcast event can be sent to one or more agents without the receiving agent's request. In contrast, an agent may define the kinds of events it wishes to know about by notifying its external world via a subscription. Instead of being *reactive* to its environment, an agent can be *proactive*. A proactive agent can selectively scan its environment for specific kinds of events whenever it wishes. Some agents employ both approaches.

Direct Communication

Life-forms need to "communicate" with their world to determine if other nearby entities are edible or are potential partners for procreation, protection, or symbiosis. For an automated agent, this might mean locating parts to ship (Order), finding the best possible price on a contract for electricity (Energy Purchase Contract), or Web spiders finding the right kind of requested information. Communication between specific agents may be direct or indirect. Direct communication can be one-way or two-way. An agent can request information from another agent, or it can provide information to that agent with or without any expectation of acknowledgment or response.

Indirect Communication

Life-forms can gain a good deal of information about their world without directly communicating with another agent. For example, ants leave a pheromone trail that guides other ants to find food. British Telecom uses a similar technique to optimize call routing. Previously completed calls leave "trail markers" that indicate paths for successfully reaching certain calling destinations. Sensing, then, can play a major role, as it does for life-forms — and such senses are not limited to the five human ones. A good example is the recent discovery of a kind of auditory "sight" via sonar. Here, dolphins not only do echolocation (ping-reflect) and passive listening, they probably use a form of signal processing akin to triangulation in vision. In doing so, they can process the reflection of ambient noise in the surrounding waters. This is not really broadcast (where the agent emits a signal) and it is not really subscription (where the observer probes the object). It is more like continually updating their knowledge of the immediate environment. The way in which an agent senses the world (like the difference between ambient response and broadcast) can make a substantial difference in the way an agent behaves.

Finite Communication

Organisms cannot know everything about their vast world, but they can scan their neighborhood for friend or foe. Similarly, it is probably not feasible (or useful) for an electric company to know about all possible sources of electricity. Instead, its "order" agents would primarily be interested in energy sources within the company's neighborhood. Each agent, then, must have the ability to "sense" particular kinds of agents within a given range. For

an autonomous agent to sense things in its environment, it is often useful to treat the environment as an agent in its own right. An agent can then communicate directly with the environment to learn more about its state.

Lessons from AARIA

Using living systems as an analogy suggests many mechanisms for designing systems of autonomous agents. We've discussed just a few of those mechanisms already being applied in complex systems. Other notions include getting and giving resources, transforming resources, interaction, rejection, pursuit, enablement, protection, adhesion, reproduction, evolution, and even death.

From the Autonomous Agents at Rock Island Arsenal (AARIA) Web site (www.aaria.uc.edu) come some compelling reasons to use agents to develop software:

- Agents are consistent with the OO paradigm. The efficiencies of programming with agents begins with the efficiencies of the OO paradigm.
- A multiagent hierarchy matches the vision many have for the future of Internet computing. The idea of intelligent entities communicating and coordinating with each other over wide area networks is a common concept in the Internet community.
- Multiagent systems can be designed to be self-configuring. Agents can be added and subtracted from the system while it is running, with no external intervention required.
- Self-configuration and decentralization provide fault tolerance. A system of autonomously functioning components will not collapse when one or more of the components fail or malfunction.
- Multiagent architectures are inherently scalable and modular. From a hardware perspective, it is substantially less expensive to use a large number of inexpensive processors than a single processor having equivalent total processing capabilities.

OBJECTS AND AGENTS: HOW DO THEY DIFFER?

Just how different are objects and agents? Some developers consider agents to be objects, with more bells and whistles. This approach tends to define agents beginning with the phrase, "An agent is an object that …," in which the definers add their favorite discriminating features. Then, there are those who see agents and objects as different, even though they share many things in common. Both approaches envision using both objects and agents in the development of software systems. This section discusses the differences and similarities between agents and objects and lets you decide which viewpoint you want to choose.

Evolution of Programming Approaches

Figure 9 illustrates one way of thinking about the evolution of programming languages. Originally, the basic unit of software was the complete program, over which the programmer had full control. The program's state was the responsibility of the

	Monolithic Programming	Modular Programming	Object-Oriented Programming	Agent Programming
Unit Behavior	Nonmodular	Modular	Modular	Modular
Unit State	External	External	Internal	Internal
Unit Invocation	External	External (called)	External (message)	Internal (rules, goals)

Figure 9 — Evolution of programming approaches [10].

programmer, and its invocation was determined by the system operator. The term modular did not apply because the behavior could not be invoked as a reusable unit in a variety of circumstances.

As programs became more complex and memory space became larger, programmers needed to introduce some degree of organization to their code. The modular programming approach employed smaller units of code that could be reused under a variety of situations. Here, structured loops and subroutines were designed to have a high degree of local integrity. Although each subroutine's code was encapsulated, its state was determined by externally supplied arguments, and it gained control only when invoked externally by a call statement. This was the era of procedures as the primary unit of decomposition.

OO added to the modular approach by maintaining its segments of code (or *methods*) as well as by gaining local control over the variables manipulated by its methods. However in traditional OO, objects are considered passive because their methods are invoked only when some external entity sends them a message.

Software agents have their own thread of control, localizing not only code and state but also their invocation. Such agents can also have individual rules and goals, making them appear like "active objects with initiative." In other words, when and how an agent acts is determined by the agent.

Agents are commonly regarded as *autonomous* entities, because they can watch out for their own set of internal responsibilities. Furthermore, agents are *interactive* entities that are capable of using rich forms of messages. These messages can support method invocation, inform agents of particular events, ask something of the agent, and receive a response to an earlier query. Lastly, because agents are autonomous, they can initiate interaction and respond to a message in any way they choose. In other words, agents can be thought of as objects that can say "no" as well as "go." Due to the interactive and autonomous nature of agents, little or no integration is required to physically launch an application. Van Parunak summarizes it well: "In the ultimate agent vision, the application developer simply identifies the agents desired in the final application, and they organize themselves to perform the required functionality" [10]. No centralized thread or top-down organization is necessary, since agent systems can organize themselves.

Object/Agent Boundaries

Before proceeding, it should be noted that OO technology can be extended in various ways to support many of the properties ascribed to agents. In fact, much of the current work on the Unified Modeling Language (UML) includes many of these notions. For example, the UML `<<thread>>` and `<<process>>` stereotypes can be considered active objects. The point here is that the agent-based approach is an extension to how we think in an OO world — just as OO was an extension to the modular programming world. Yes, objects could be used to support the agent-based approach, just like any modular language (such as C or COBOL) could be used to write OO code. So, why not just write in C and forget about C++ or Java? The answer lies in building on what we know to provide another way of thinking about systems and their implementation. Agents, then, are an evolution rather than a revolution.

The rest of this section explains those aspects of agents that are different from the *conventional* OO approach (i.e., the way OO is commonly practiced and supported by most OO languages, such as C++ and Smalltalk). Different, here, does not mean bad or good — only different. In the end, you might conclude that agents are really just enhanced objects or that agents and objects are different but can peacefully coexist and even *support* one another in the same system. Either way, the agent-based way of thinking brings with it a useful and important perspective for system development. If we can imagine agents as a pattern for

systems, we can avoid any of the "OO versus agents" controversy and just get on with developing systems in a richer way. Both approaches are useful for IT development.

Agents Are Autonomous

Since a key feature of agents is their autonomy, agents are capable of initiating action independent of any other entity. However, such autonomy is best characterized in degrees, rather than simply being present or not. To some degree, agents can operate without direct external invocation or intervention.

Dynamic Autonomy

Autonomy has two independent aspects: dynamic autonomy and nondeterministic autonomy. Agents are dynamic because they can exercise some degree of activity. As illustrated in Figure 10, an agent can range from passive to entirely proactive. For example, although ants are basically reactive, they exhibit a small degree of proactivity when they choose to walk, rest, or eat. A supply chain agent can react to an order being placed and be proactive about keeping its list of suppliers up to date.

GM paint booths are treated as agents. Here, information about an unpainted car or truck coming down the line is posted in an automated form that is accessible to all paint booths. When a paint booth nears completion of its current job, it basically says, "Hmmm, I'm running out of work, I'll look over at the jobs posted." If the booth is currently applying the color of paint required by an upcoming job, it will bid more for the job than a booth having a different color. Other bidding criteria could include how easy or important the job is. In a top-down, planned, "push-through" world, if one booth malfunctions, the plan would require immediate recomputing; with bottom-up, "pull-through" paint-booth agents, there are other booths to pick up the bidding slack at a moment's notice [11].

Agents can react not only to specific method invocations but also to observable events within the environment. Proactive agents will actually poll the environment for events and other messages to determine what action they should take. To compound this, in multiagent systems, agents can be engaged in multiple parallel interactions with other agents — magnifying the dynamic nature of agents. In short, an agent can decide when to say "go."

Objects, in contrast, are conventionally passive, with their methods being invoked under a caller's thread of control. The term autonomy barely applies to an entity whose invocation depends solely on other components in the system. However, UML and Java have recently introduced event-listener frameworks and other

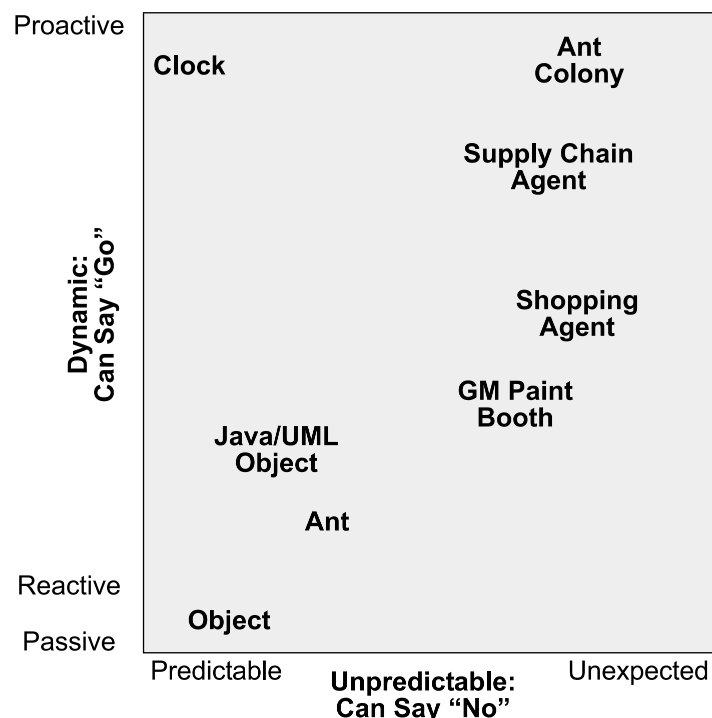

Figure 10 — Two aspects of autonomy (based on collaborative work with Van Parunak).

mechanisms that allow objects to be more active. In other words, objects are now capable of some of the dynamic capability of agents.

Unpredictable Autonomy

Agents can also employ some degree of unpredictable (or nondeterministic) behavior. When observed from the environment, an agent can range from being totally predictable to completely unpredictable (see Figure 10 on previous page). For example, an ant that is wandering around looking for food can appear to be taking a random walk. However, once pheromones or food are detected, its behavior becomes reasonably predictable. In contrast, it is difficult to predict which GM paint station will paint which vehicle. The behavior of a shopping agent might be highly unpredictable. For example, giving it criteria for a gift will not predict exactly which gift it will choose. In fact, the agent might return empty handed because it did not find any gifts that match the criteria. In other words, the agent can also say "no."[1]

Conventional objects do not have to be completely predictable. However, the typical usage and direct support with OO languages tends toward a more predictable approach. For instance, when a message is sent to an object, the method is predictably invoked. Yes, an object may determine whether or not to process the message and how to respond if it does. However, in common practice, if an object says no, it is considered an error situation; with agents, this is not the case.

Usually, object classes are designed to be predictable, to facilitate buying and selling reusable components. Agents are commonly designed to determine their behavior based on individual goals and states, as well as the states of ongoing conversations with other agents. Although OO implementations can be developed to include nondeterministic behavior, this is common in agent-based thinking.

Agent behavior can also be unpredictable because the agent-based approach has a more "opaque" notion of encapsulation. First, an agent's knowledge can be represented in a manner that is not easily translated into a set of attributes. Even if an agent's state were publicly available, it may be difficult to decipher or understand. This is particularly true when the agent involves neural networks or genetic structures. You can look at it, but you can't always understand what you see.

Second, the requested behaviors that an agent performs may not even be known within an active system. This is a clear distinction from object systems, because current OO languages only let you ask an object what interfaces it supports. Since the programmer needs to have some idea what interface to ask for, this makes coding difficult. In OO, there is no provision in current languages for an object to "advertise" its interfaces. In contrast, an agent can employ other mechanisms, such as publish/subscribe, protocol registration, and "yellow page" and "white page" directories. Another common mechanism provides specialized *broker* agents to which other agents can make themselves known for various purposes but are otherwise unlisted to the rest of the agent population.

Lastly, the underlying agent communication model is usually asynchronous. This means that there is no predefined flow of control from one agent to another. An agent may autonomously initiate internal or external behavior at any time, not just when it is sent a message [12]. Asynchronous messaging and event notification are part of agent-based messaging systems, and agent languages need to support parallel processing. These are not part of the run-of-the-mill OO language. Those that require such functionality in an OO system typically layer these features on top of the object model and OO environment. Here, the agent model explicitly ties together the objects

[1] The FIPA standards organization states that all agents must be able to handle all messages that they receive. Here, an agent may choose various actions, such as respond in a manner of its choosing, decide that the request is outside of its competency, ignore the message because it is not well formed, or just refuse to do it on various grounds.

(data and functionality) with the parallelism (execution autonomy, thread per agent, etc.). According to Geoff Arnold of Sun Microsystems, "Just as the object paradigm forced us to rethink our ideas about the proper forms of interaction — access methods versus direct manipulation, introspection, etc. — so agents force us to confront the temporal implications of interaction — messages rather than RMI, for instance."

Agents Are Interactive

Interaction implies the ability to communicate with the environment and other entities. As illustrated in Figure 11, interaction can also be expressed in degrees. On one end of the scale, object messages (method invocation) can be seen as the most basic form of interaction. A more complex degree of interaction would include those agents that can react to observable events within the environment. For example, food-gathering ants don't invoke methods on each other; their interaction is indirect, through direct physical effects on the environment. In multiagent systems, agents can be engaged in multiple, parallel interactions with other agents. Here, agents can act as a society.

One Method Per Message

An object's message may request only one operation, and that operation may only be requested via a message formatted in a very exacting way. The OO message broker has the job of matching each message to exactly one method invocation for exactly one object.

Agent-based communication can also use the method invocation of OO. However, the demands that many agent applications place on message content are richer than those commonly used by object technology. Although *agent communication languages* (ACLs) are formal and unambiguous, their format and content vary greatly. In short, an agent message could consist of a character string whose form can vary yet it obeys a formal syntax, while the conventional OO method must contain parameters whose number and sequence are fixed. Theoretically, this could be handled with objects by splitting the world into two portions: one including messages for which we have conventional methods, another including messages that we send as strings.

To support string-based messages in an OO language, you could either anticipate every possible variation by supplying a specialized method for each or use a general utility `AcceptCommunicativeString` method. The `AcceptCommunicativeString` method could cover the multitude of services that an object might handle. However, with just a single method, the underlying services would not be part of the published interface. In the traditional OO environment, such an environment would be boring and not very forthcoming. In agent-based environments, agent public services and policies can be made explicit through a variety of techniques (described earlier).

Agent Communicative Languages

Since we may wish to send a message to any (and every) agent, we need the expressive power to cover all desired situations — including method invocation. Therefore, an ACL is necessary for expressing communications among agents and even objects. The ACL syntax could be specially crafted for each application. However, the lack of standardization would quickly result in a tower of Babel. Here, two applications could have difficulty interacting with one another; for an entire organization, it would be totally impractical. Standard ACL formats, then,

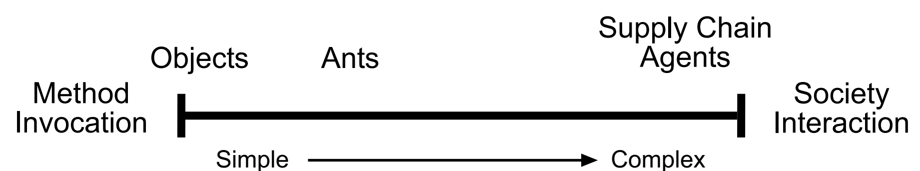

Figure 11 — Degrees of interaction.

would be desirable. Two of the most popular general-purpose ACLs are KQML and the FIPA ACL. These ACLs communicate agent speech acts, specify ontologies, and participate in discussion patterns called protocols. (See Part 1 of the *Executive Report*, Vol. 3, No. 4, for more information.)

Conversations and Long-Term Associations

Another way in which agent interaction can be more than just method invocation is that agents can be involved in long-term conversations and associations. Agents may engage in multiple transactions concurrently, through the use of multiple threads or similar mechanisms. In an agent messaging environment, each conversation can be assigned a separate identity. Additionally, either a unique message destination or a unique identifier can be used to sort out the threads of discourse. Conventional OO languages and environments have difficulty supporting such a requirement, directly or indirectly. It should be mentioned that objects could be used for the elements of agent conversation — including the conversation itself. In other words, agents can employ objects for those situations requiring entities with little autonomous or interactive ability.

Third-Party Interactions

Arnold has considered the question of third-party interactions, which are very hard for strongly typed object systems to handle. Here, two patterns come to mind. The first involves a broker that accepts a request and delegates it to a particular service provider based on an algorithm that is independent of the type of service interface (e.g., cost, reachability). The second involves an anonymizer that hides the identity of a requester from a service provider. Models based on strong typing, such as CORBA, Remote Method Invocation, and Jini, cannot easily support these patterns.

Philosophical Differences

Two key areas that can differentiate the agent-based approach from traditional OO are autonomy and interaction. However, there are other ways in which agents may differ from objects. The list below describes some underlying concepts that agent-based systems can employ. None of them are universally used by agents, and no agent system is required to use any of them.

- **Decentralization.** Objects can be thought of as centrally organized because an object's methods are invoked under the control of other components in the system. Yet some situations require techniques that are decentralized and self-organized. For example, classical ballet requires a high degree of centralization called choreography, while at the other extreme the processes of nature involve a high degree of individual direction. Most businesses require a balance of standardized procedures and individual initiative: one extreme or the other would be detrimental to the business.

  Supply chain systems can be planned and centrally organized when the business is basically stable and predictable. In unstable and unpredictable environments, supply chains should be decentralized and self-organized (an option not supported by commercial supply chain systems today). Agent-based environments can employ both centralized and decentralized processing. Although agents can certainly support centralized systems, they can also provide us with the ultimate in distributed computing.

- **Multiple and dynamic classification.** In OO languages, objects are created by a class and, once created, may never change their class or become instances of multiple classes (except by inheritance). Agents provide a more flexible approach. For example, a particular agent can be a person, employee, spouse, landowner, customer, and seller all at the same time (or at different times). When the agent is an employee, that agent has all the state and procedural elements consistent with being an employee. If the agent is

terminated from his or her job, the employment-related state and procedural elements are no longer available to the agent. Whether employed or not, the agent is still the same entity — it just has a different set of features. The ability to express roles and role changes is not new to OO. However, most OO languages do not directly support this mechanism (despite the fact that UML does).

Furthermore, agents can play different roles in different domains. When you go to work, you play the employee role. When you return home, you change roles — for example, playing the spouse role. OO languages do not directly support such domain-dependent mechanisms. The single-class OO approach is efficient and reliable; the multiple and dynamic approach provides flexibility and more closely models our perception of the world. Agents can use either approach; the choice belongs to the system designer.

- **Business concepts.** Agent-based systems can support concepts such as rules, constraints, goals, beliefs, desires, and responsibilities. Although object systems are being built to include these (particularly the IF-THEN rules of expert systems), they are not directly supported by traditional OO. In other words, some agent-based approaches expressly consider these notions as useful components of its entities; the traditional OO approach does not. We could either extend objects with these agent-based concepts and call them objects++ or simply refer to them as agents. The result is essentially the same: we have extended the way we can build systems using an agent-based view.
- **Instance-level features.** The features possessed by each object are defined by its class — a benefit also enjoyed by agents. However, each agent can acquire or modify its own features; i.e., features that are not defined at the class level, but at the *individual* agent (or instance) level. In other words, if an individual agent has the ability to learn, it can change its own behavior, permitting it to act differently than any other agent. If an agent can change itself, it can add (as well as subtract) features dynamically. For example, with genetic programming software, agents are created genetically. Each "parent" contributes some portion of an offspring agent's genetic string, in much the same way this occurs in nature. This approach is particularly popular in one area of agent-based systems known as artificial life. (Artificial life is the study of man-made systems that exhibit the behavioral characteristic of natural living systems. It models "life as we know it" within the larger picture of "life as it should be.")
- **Small in impact** [10]. Both objects and agents can be described as slim or fat, small grained or large grained. Additionally, in systems with large numbers of agents or objects, each can be small in comparison with the whole system. However, an individual agent can have less impact on a system than an object. For example, each ant is an almost negligible part of the entire ant colony. As a result, the behavior of the whole tends to be stable, despite performance variations or the death of a single agent. In an agent-based supply chain, if a supplier or a buyer is lost, the collective dynamics can still dominate. If an object is lost in a system, an exception is raised.
- **Small in time.** Naturally occurring agent systems can forget. Ant pheromones evaporate; our own memories can fade. Even the death of unsuccessful organisms in an ecosystem is an important mechanism for freeing up resources for better adapted organisms. Such analogies work for both agent-based and OO software systems. With agents, such comparisons are a natural part of the approach.
- **Small in scope.** Animals can usually sense only their immediate vicinity. Despite this

restriction, they can generate effects that extend far beyond their own limits. For example, an ant can sense a trail of pheromones only when its path intersects with the pheromone trail, but the ant's ignorance of the vast pheromone network laid out by all the other ants does not prevent the overall system from working. In other words, it is not necessary — in fact, it is not feasible — for every agent to know everything. Instead of being omniscient and omnipotent, large agent-based systems are local sensing and acting. Objects, too, employ this analogy to some extent because objects generally only interact with other objects linked to them. Also, objects using integrated databases can be programmed to access databases having only local knowledge. Although being restricted to local knowledge is not a new concept, the notion is commonly used with agents.

- **Emergence.** The interaction of many individual agents can give rise to secondary effects where groups of agents behave as a single entity. For example, ant colonies, flocks of birds, and stock markets have emergent qualities. Each consists of individual agents acting according to their own rules and even cooperating to some extent. Yet, ants colonies thrive, birds flock, and markets achieve global allocations of resources — all without a central cause or an overall plan. Agents can possess just a few very simple rules to produce emergence. In fact, when constructing agent-based systems, starting out with simple agents is important because emergence is then easier to understand and harness. More complexity can be added over time to avoid being overwhelmed.

  Since traditional objects do not interact without a higher-level thread of control, emergence does not usually occur. As more agents become decentralized, their interaction is subject to emergence — either positive or negative. This phenomenon is both good news and bad news for large, multiagent systems.

- **Analogies from nature.** The autonomous and interactive character of agents more closely resembles natural systems than do objects. Since nature has long been successful, identifying analogous sit-uations to use in agent-based systems is sensible. For example, agents can die when they lack supportive resources. In supply chain manufacturing, when a manufacturing-cell agent cannot operate profitably, it dies of "malnutrition." Another manufacturing cell could come by and scavenge useful bits from the newly dead cell.

  Agents can exhibit properties of parasitism, symbiosis, and mimicry. They can participate in "arms races" where agents can learn and outdo other agents. Agents can participate in sexual (and asexual) reproduction that can incorporate principles from Darwinian and Lamarckian evolution. Agent societies can exhibit political and organizational properties — whether they're organized, anarchic, or democratic. In short, nature can provide a rich trove of ideas for multiagent system design.

Agents Versus Objects Conclusion

Agents employ some of the mechanisms and philosophies used by objects. In fact, many software developers strongly advocate composing agents from objects — building the infrastructure for agent-based systems on top of the kind of support systems used for OO software systems. For example, many structures and parts of agents can be reasonably expressed as objects. These might include agent names, agent communication handles, agent communication language components (including encodings, ontologies, and vocabulary elements), and conversation policies.

In multiagent systems, an additional layer of software components may be naturally expressed as objects and collections of objects. This is the underlying

infrastructure that embodies the support for agents composed of object parts. This might include communication factories, transport references, transport policies, directory elements, and agent factories.

Agents are autonomous entities that can interact with their environments. But are they just objects with extra attributes or are they an entirely different approach? And how important is it to answer this question? What is important is that objects and agents are distinct enough to treat them differently. When we design systems, we can choose a well thought-out mixture from both approaches. We can even build aggregates in which agents consist of both objects and other agents, and vice-versa. In short, there is no right answer here — only a useful one. For Grady Booch, the answer is clear:

Agents are important/useful because:

- They provide a way to reason about the flow of control in a highly distributed system.
- They offer a mechanism that yields emergent behavior across an otherwise static architecture.
- They codify best practices in how to organize concurrent collaborating objects.

REFERENCES

1. Kelly, Susanne, and Mary Ann Allison. *The Complexity Advantage: How the Science of Complexity Can Help Your Business Achieve Peak Performance*. McGraw-Hill, 1999.

2. Kelly, Kevin. *Out of Control: The Rise of Neo-Biological Civilization*. Addison-Wesley, 1994.

3. Roy, Beau. "Using Agents to Make and Manage Markets Across a Supply Web." *Complexity*, 3:4, 1998, pp. 31-35.

4. Resnick, Mitchel. "Unblocking the Traffic Jams in Corporate Thinking." *Complexity*, 3:4, 1998, pp. 27-30.

5. Rothchild, Michael. *Bionomics: Economy as Ecosystem*. College Board, 1995.

6. Coveney, Peter, and Roger Highfield. *Frontiers of Complexity: The Search for Order in a Chaotic World*. Fawcett Columbine, 1996.

7. Epstein, Joshua M., and Robert Axtell. *Growing Artificial Societies: Social Science from the Bottom Up*. MIT Press, 1996.

8. Holland, John H. *Emergence: From Chaos to Order*. Perseus Books, 1999.

9. Holland, John H. *Hidden Order: How Adaptation Builds Complexity*. Perseus Books, 1996.

10. Parunak, H. Van Dyke. "'Go to the Ant': Engineering Principles from Natural Agent Systems." Annals of Operations Research, 75, 1997, pp. 69-101.

11. Morley, Dick. "Cases in Chaos: Complexity-Based Approaches to Manufacturing." Embracing Complexity proceedings from Ernst & Young Center for Business Innovation, August 1998, pp. 97–102.

12. Wooldridge, Michael, Nicholas R. Jennings, and David Kinny. "The Gaia Methodology for Agent-Oriented Analysis and Design." *Autonomous Agents and Multi-Agent Systems* (forthcoming issue, 2000).

Cutter Consortium Distributed Computing Architecture/e-Business Advisory Service

Senior Consultants

Douglas Barry

on storing objects using DBMSs

Douglas Barry is principal of Barry & Associates, Inc., which he founded in 1992, and executive director the Object Data Management Group, an industry standards organization. Mr. Barry has worked with DBMS technology for more than 20 years and with object technology and databases since 1987. He has been actively involved in creating and promoting standards for storing objects in databases. Mr. Barry specializes in the strategies, technologies, and products associated with objects and object-relational database applications.

Peng Boey

on component-based architecture for e-business applications

Peng Boey is vice president of Consulting Services with NetNumina Solutions. With more than 10 years of IT experience, Mr. Boey is an expert in distributed systems architecture. He has provided professional advice to Global 1000 companies on how to design, build, and deploy component-based architectures for e-business applications. He has been a leader in creating the VIEW methodology, a revolutionary architecture process that utilizes the latest enabling technologies for building mission-critical e-business systems for the Internet, as well as for intranets and extranets. Currently, Mr. Boey is conducting research on developing component frameworks for rapidly building e-business solutions.

Theodore R. Burghart

on developing heterogeneous client-server and distributed systems

Theodore Burghart is principal engineer at Quoin, Inc. He has extensive experience with heterogeneous client-server and distributed systems design and development. His projects have included communications, database, technical, and process control services implementations. Mr. Burghart is experienced in cross-platform enabling technologies, such as CORBA as well as with LDAP, and with relational, object/relational, object-oriented, and full-text databases. Currently, Mr. Burghart collaborates with development teams to define and construct CORBA-based infrastructures. In addition, Mr. Burghart provides technology training and consulting services to clients in the healthcare, insurance, and financial services industries.

Richard Dué

on component development methods and project management

Richard T. Dué is president of Thomsen Dué and Associates Limited. He specializes in object and component development methods and in object technology project management. Mr. Dué has developed and presented information technology training courses in 28 countries to participants from hundreds of organizations. He is a member of the OPEN methodology consortium and has been actively involved in developing business object standards. Mr. Dué is a frequent contributor to the *Cutter IT Journal*, and has held various management positions in the public and private sector in the US and Canada.

David Frankel

on Java- and Internet-based component architectures

David Frankel is chief scientist at Genesis Development. He assists clients in developing and customizing advanced component architectures based on CORBA, DCOM, Java, the Internet, and related technologies. Mr. Frankel has been instrumental in formalizing advanced component architecture to support large-scale software development and systems integration. He is a member of the OMG Architecture Board, was a major contributor to the CORBA/COM Internetworking standard, and is cochair of the OMG Business Object Initiative Working Group.

Max Grasso

on distributed secure transaction systems

Max P. Grasso is chief technology officer of NetNumina Solutions. He is a recognized expert on distributed secure transaction systems with a focus on high reliability, mission-critical applications. He also has significant expertise in the management issues involved with deploying such systems. Mr. Grasso has been at the forefront of distributed computing technology since its beginning, both as a member of the Open Software Foundation's team and as a cofounder of the Open Environment Corporation. As CTO of Internet Business Solutions, his mission was building the technology for the execution of secure distributed transactions on the Internet. In that role he designed a framework for business-to-business transactions and interenterprise transactional workflows. Mr. Grasso has overseen the architecture and the design of large systems in the telecommunication, financial, banking and gaming industries.

Michael Guttman

on transitioning to enterprise component technology

Michael Guttman is chief technical officer and cofounder of Genesis Development, where he assists clients in planning their transitions to enterprise component technology. Mr. Guttman, who has been a pioneer in the use of component and object technology for large-scale distributed systems, is a specialist in advanced component architectures. Mr. Guttman has more than 20 years of experience in software development and has been a major contributor to several OMG standards, including CORBA 1.0, CORBA IIOP, and CORBA/COM Internetworking.

Senior Consultants

Curt Hall

on data warehousing and data management strategies

Curt Hall, editor of *Business Intelligence Advisor*, is an expert on data warehousing technologies and products. His recent study on the corporate use of data warehouses and the issues associated with data warehousing projects has resulted in the in-depth report *Data Warehousing for Business Intelligence*. Mr. Hall is the coauthor of *Intelligent Software Systems Development* and a contributing editor to James Martin and James Odell's *Object-Oriented Methods: Pragmatic Considerations*. He is the former associate editor of *Object-Oriented Strategies* and *Application Development Strategies*. Mr. Hall's work has appeared in technical journals such as *IEEE Expert*. He has also been an organizer of and speaker at industry events such as *ObjectWorld*.

Paul Harmon

on distributed computing and component development for business applications

Paul Harmon is a well-known consultant and analyst of software trends. Mr. Harmon has been very influential in the movement to commercialize object and component technologies for business applications. Mr. Harmon recently completed a study of the acceptance of object technology and components in corporate development groups. As editor since 1991 of *Component Development Strategies*, published by Cutter Information Corp., Mr. Harmon has studied the commercial and business applications of object technology. He has also been the editor of three other Cutter Information Corp. newsletters over the years: *Intelligent Software Strategies*, *Application Development Strategies*, and *Business Process Strategies*. Mr. Harmon is a frequent speaker on the strategic impact of new software technologies on business. Mr. Harmon is the coauthor of several books.

Ian Hayes

on e-business strategy

Ian Hayes is founder president of Clarity Consulting, Inc., where he provides strategic consulting on issues affecting the management and support of corporate business systems. Mr. Hayes has advised dozens of *Fortune* 1000 companies on a variety of IT issues, including major Y2000 initiatives, e-business, insourcing, outsourcing, and process improvement. Mr. Hayes is a regular contributor to the *Cutter IT Journal* and is on the editorial advisory board of the *Enterprise Application Integration Journal*. Mr. Hayes was a cofounder of Language Technology, Inc., an early software redevelopment product vendor, and a practice manager at Keane, Inc. before founding Clarity Consulting in 1993.

J. Bradford Kain

on distributed business components

Brad Kain is CEO and cofounder of Quoin, Inc., providing consulting, mentoring, and software development services in object and distributed technology. Mr. Kain has used object-oriented analysis and design since 1987. He has helped define the use of object and distributed technology to realize distributed business components. This work has involved the specification of sophisticated intranet, Java, and distributed applications. Mr. Kain has managed the technical direction and development teams of distributed application infrastructure development projects for managed care, client management, general ledger, securities trading, marketing, engineering and manufacturing design, and other applications. Mr. Kain has participated in the work of the Object Management Group's Technical Committee on CORBA and the specification of domain services.

André Leclerc

on formal specification approaches to the development of information and management systems

André Leclerc is the director of development for Technology Development Associates, Inc., where he is active in developing, training, consulting and mentoring object-oriented information systems. Mr. Leclerc's interest is in formal specification approaches to the development of information and management systems. In 1984, Mr. Leclerc was appointed vice president of Yourdon, Inc. Following his tenure at Yourdon, Inc., he served as vice president of Kenneth G. Moore and Associates. Mr. Leclerc has authored a book on structured PL/1, and a variety of articles, seminars, and tutorials on information systems, including the OO seminars for Ptech, Inc.

Jean Pierre LeJacq

on architecture and implementation of distributed systems

Jean Pierre LeJacq, an experienced architect, designer, and implementer of distributed systems, is CTO and cofounder of Quoin, Inc., providing consulting in object and distributed technologies to clients worldwide. Mr. LeJacq is the architect and technical lead for the development of an infrastructure for distributed application development, and is responsible for the design and implementation of a CORBA-based system. He has extensive experience in Java, C++, and UNIX-based systems, and in a variety of design methods. Mr. LeJacq has been using object-oriented languages and modeling systems since 1984 for clinical, managed care, client management, engineering and manufacturing design, and aircraft control simulation applications.

Jason Matthews

on transitioning to enterprise component technology

Jason Matthews is cofounder of Genesis Development. He has nearly 20 years of technical and management experience in software development and related professional services. Mr. Matthews is a pioneer in the use of component/object technology for large-scale distributed systems and the Internet, and a specialist in the process of transitioning large organizations to component technology. Mr. Matthews has managed end-user information systems organizations and the development of commercial software products. He has been a consultant to a wide range of industries, including financial services, insurance, healthcare, manufacturing, telecommunications, and energy.

Senior Consultants

James Odell

on object-oriented methodologies and agent technology

James Odell was an early innovator of information engineering methodologies, and has spent most of his 30-year career developing better methods to understand, communicate, and manage system requirements. Working with the Object Management Group (OMG) and other major methodologists, Mr. Odell continues to innovate and improve object-oriented methods and techniques. He participated in the development of the UML, and is the cochair of both the OMG's Object Analysis and Design Task Force as well as the Agents Work Group. Formerly, Mr. Odell was the principal consultant for KnowledgeWare, Inc., where he pioneered the concepts of data modeling, information strategy planning, and CASE technology application. Mr. Odell has coauthored several books with James Martin, including the most recent title *Object-Oriented Methods: A Foundation, UML Edition*.

Chris Pickering

on e-business trends and strategies

Chris Pickering, president of the research and consulting firm Systems Development, Inc., analyzes industry practices. Mr. Pickering's areas of focus include information architecture, business-IT alignment, technology acquisition and deployment, organizational change, system modeling, and software practices. He is the author of the survey-based study *E-Business Trends, Strategies, and Technologies* and the periodic *Survey of Advanced Technology*, which tracks the use of advanced information technologies, assesses the effectiveness of that use, and identifies the benefits and hazards of using the leading technologies. He then applies the lessons learned from the research to helping clients maximize their information technology investments. Mr. Pickering's articles and his research findings have appeared in leading industry magazines and books, and he is a speaker at a variety of software conferences.

John R. Rymer

on tools, middleware, and application development for distributed systems

John Rymer is president of Upstream Consulting, which he founded in 1997. Mr. Rymer is a well-known strategy advisor and a veteran industry analyst. Since 1989, Mr. Rymer has developed a strong track record of helping software companies solve difficult market and technical problems. He specializes in application development technology for distributed systems, including tools and middleware. Mr. Rymer is a former vice president and founding analyst at Giga Information Group, Inc., where he was responsible for tracking application development technology and products. Mr. Rymer has been a keynote speaker at *OOPSLA, Networld + Interop, ObjectWorld,* and other industry conferences.

Greg Sabatino

on architecting and implementing highly scalable distributed e-business solutions

Greg Sabatino, cofounder of NetNumina Solutions, specializes in the architecture and implementation of highly scalable distributed e-business solutions. Mr. Sabatino's career has centered on the training, support, and delivery of distributed computing architectures and applications for IT organizations worldwide. His efforts focus on enabling organizations to successfully integrate and employ emerging technologies in order to realize a strategic advantage. Mr. Sabatino's experience spans the retail, petrochemical, telecommunications, pharmaceutical and, especially, finance industries. Mr. Sabatino contributes to several industry publications and speaks at conferences on a variety of distributed computing issues.

Kent Seinfeld

on enterprise information architecture development

Kent Seinfeld is the founder of Enright Consulting, a small group of senior IT consultants. Mr. Seinfeld specializes in enterprise information architecture development. Mr. Seinfeld is a former senior vice president of IT and served in three different positions with CoreStates Bank. He was the founder and manager of the Technology Planning and Research group at CIGNA, a global insurance and financial service company, where he was responsible for computing standards, security policy, development methodologies, and the research and development program. Mr. Seinfeld was the CIO for Girard Bank. Earlier in his tenure he was the principal architect in the design and implementation of a large-scale highly integrated banking system. This system evolved into the foundation of one of the first large ATM networks.

Roger Sessions

on distributed middle-tier technologies

Roger Sessions is the world's leading expert on Microsoft's distributed middle-tier technologies, including COM, DCOM, and MTS. Prior to starting his company, ObjectWatch, Inc., Mr. Sessions worked at IBM, where he was an architect of one of the CORBA services. He was also a lead architect for IBM's implementation of the CORBA Persistence Service, gaining an unparalleled perspective on middle-tier technologies. Mr. Sessions has written four books; his most recent is *COM and DCOM: Microsoft's Vision for Distributed Objects*. He writes the highly respected and often controversial online *ObjectWatch Newsletter*. In addition to frequent speaking engagements worldwide, Mr. Sessions writes articles for many industry publications.

Ed Yourdon

on object-oriented design and analysis

Ed Yourdon, chairman of Cutter Consortium, is widely known as the lead developer of the structured analysis/design methods of the 1970s. He was a codeveloper of the Yourdon-Whitehead method of object-oriented analysis/design and the popular Coad-Yourdon OO methodology. Mr. Yourdon began his career at Digital Equipment Company more than 30 years ago. He has been involved in a number of pioneering computer technologies, such as time-sharing operating systems and virtual memory systems. He is currently a member of the Airlie Council, a group of high-end advisors formulating software "best practices" for the US Department of Defense. Mr. Yourdon is the editor of the *Cutter IT Journal*. He has authored more than 200 technical articles and written 25 computer books since 1967, including *The Rise and Resurrection of the American Programmer* and *Death March: The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects.*