

Objects and Agents Compared

James Odell, Independent Consultant

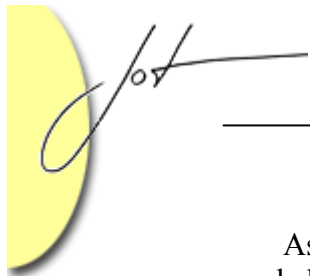
Just how different—or the same—are objects and agents? Some developers consider agents to be objects, except with more bells and whistles. Then, there are those who see agents and objects as different even though they share many things in common. Both approaches, however, envision using objects and agents together in the development of software systems. In other words, objects and agents are two distinct notions—each having its own particular place in software development. The important point here is that the agent-based way of thinking brings a useful and important perspective for system development, which is different from—while similar to—the object-oriented way. This paper discusses some of the differences and similarities between agents and objects and lets you decide which viewpoint you want to choose.

1 EVOLUTION OF PROGRAMMING APPROACHES

Figure 1 illustrates one way of thinking about the evolution of programming languages. Originally, the basic unit of software was the complete program where the programmer had full control. The program's state was the responsibility of the programmer and its invocation 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.

	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 1 — Evolution of programming approaches [1].



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. While 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.

In contrast, object orientation 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 their invocation as well. 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.

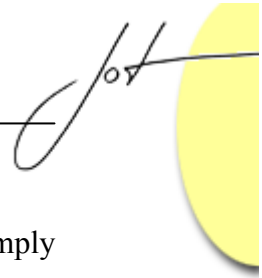
Agents are commonly regarded as *autonomous* entities, because they can watch out for their own set of internal responsibilities. Furthermore, agents are *interactive* entities that are capable of using rich forms of messages. These messages can support method invocation—as well as informing the agents of particular events, asking something of the agent, or receiving 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 interaction 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 the *agents* organize themselves to perform the required functionality." [1] No centralized thread or top-down organization is necessary since agent systems can organize themselves.

2 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 Fig. 2, an agent can have some degree of behavior from being simply passive to entirely proactive.

For example, paint booths were experimentally treated as agents at GM. 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 on 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. [2]

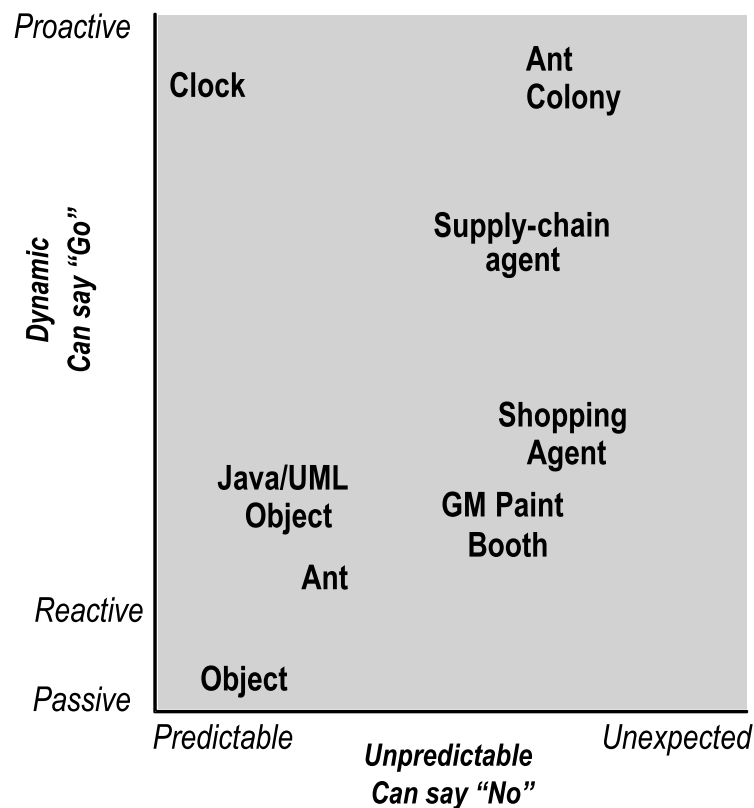
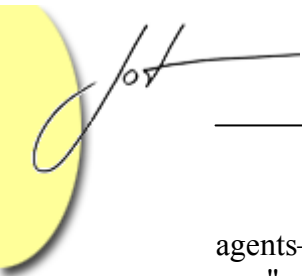


Figure 2 — Two aspects of autonomy (based on collaborative work with Van Parunak).

Agents can react not only to specific method invocations but to observable events within the environment, as well. 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, on the other hand, 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 mechanisms for allowing objects to be more active. In other words, objects are now capable of some of the dynamic capability of agents.

Unpredictable Autonomy

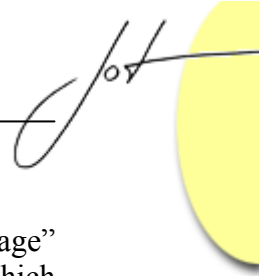
Agents may also employ some degree of unpredictable (or nondeterministic) behavior. When observed from the environment, an agent can range from being totally predictable to completely unpredictable (Fig. 2). For example, an ant that is wandering around looking for food can appear to be taking a random 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. 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 in order 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. While 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, 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

¹ The FIPA agent 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.



publish/subscribe, protocol registration, “yellow page,” “green 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.

Second, 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 anytime, not just when it is sent a message [3]. 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 (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 vs. direct manipulation, introspection, etc.), so agents force us to confront the temporal implications of interaction (messages rather than RMI, for instance).”

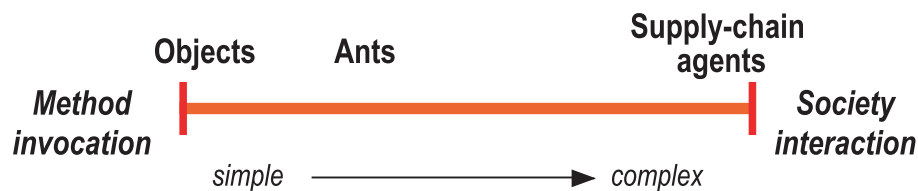


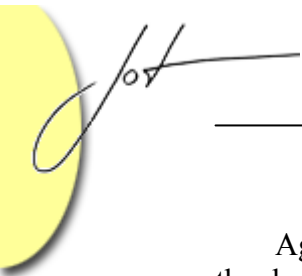
Figure 3 — Degrees of interaction.

3 AGENTS ARE INTERACTIVE

Interaction implies the ability to communicate with the environment and other entities. As illustrated in Fig. 3, 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. And finally 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. While *agent communication languages* (ACL) 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 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, then, 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 both 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 agent communication language 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, 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.

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.



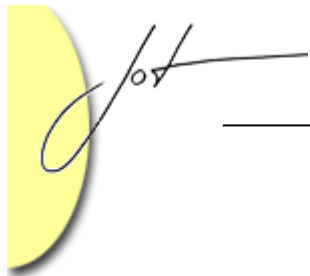
Third-party interactions

Geoff 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 some 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, RMI, and Jini, cannot easily support these patterns.

4 PHILOSOPHICAL DIFFERENCES

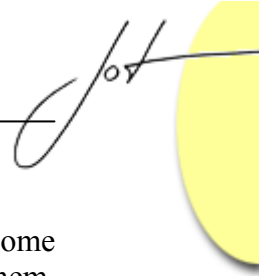
Two key areas that can differentiate the agent-based approach from traditional OO are autonomy and interaction. However, there are other ways in which agents may seem to differ from objects. The list below describes some underlying concepts that agent-based systems can employ. None are universally used by agents: active object systems may use them as well. Furthermore, 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. However, 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. While 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 can 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 now 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



(even though UML does). Furthermore, agents might 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 that are necessary for agent-based environments. 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.

- **Instance-level features** The features possessed by each object are defined by its class—a benefit enjoyed by agents as well. However, each agent may also 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. Here, each parent contributes some portion of an offspring agent's genetic string—much in the same way that 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** [1] 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 any 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 object-oriented software systems. With agents, such comparisons are a natural part of the approach.
- **Small in scope** Animals can usually sense only their immediate vicinity. In spite of 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. Despite the ant's ignorance of the vast pheromone network laid out by all the other ants, ant colonies work. In other words, it is not necessary—in fact, 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. So, while being restricted to local knowledge is not a new concept, with agents the notion is commonly used.
- **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 the 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 very successful, identifying analogous situations 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." Furthermore, 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 are organized, anarchic, or democratic. In short, nature can provide a rich trove of ideas for multiagent system design.

5 TOWARDS THE COEXISTENCE OF AGENTS AND OBJECTS

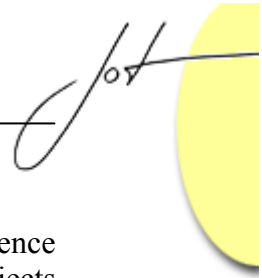
It is tempting to assert: "An agent is an object that...."—completing the phrase in a variety of ways. Jeff Bradshaw [4] refers to agents as "objects with an attitude" in the sense of being objects with some additional behavior added, for instance, mobility, inference, etc. Here, we are using objects as a generic modeling or abstraction mechanism, independently of whether agents are implemented as objects (using object-oriented programming techniques).

The viewpoint might help resolve the problem of multiple kinds of agents. In a sense, we could view the maximal agent as potentially having all behaviors found in the agent attributes list, and that degenerate forms of agents are those containing fewer than all properties. In this view, *objects are agents without these extra agent attributes*. This helps explain how agents might literally be "objects with an attitude." Taking this view, considering agents that use agent communication languages as having the ability to "just say no" to message requests, we can view objects as degenerate agents that always do as they are told.

One can now simultaneously argue that agents are objects and that agents are different from objects. These extra agent capabilities could be added to objects but then they would become agents. The nice thing, then, is we have a path for treating agents and objects the same in models and both as first class model elements. For example, we could distinguish agents that communicate natively as *agent-oriented* and objects that encapsulate and simulate native agent capabilities as *agent-based*, in comparison to the traditional, similar distinction between *object-oriented* and *object-based*.

However are some of the challenges that confront us in the area of trying to understand the relationship of objects to agents [5]:

- We know how to wrap legacy code and data with an object wrapper so the legacy code and data can participate in object message passing. Can we do the same thing with code, data (and objects), e.g., wrap them in an agent wrapper so they can communicate as agents? Or do we need to add qualitatively more to turn legacy code into agents? One example of a place where this is important is in agent services. There seem to be useful services like registration, logging, persistence, transactions, matchmaking (trading), administration, replication, security, and many more that agents and agent systems would benefit from. Can we just straightforwardly use object services for these? How important is it to wrap these in agent wrappers to permit them to communicate via an agent communication language (ACL)? Is there value in these middleware services having autonomy?
- Can we really add additional features to objects so they become agents? This is consistent with the general view that we progress computer science by adding and packaging useful capability first to climb to abstract data types, then to objects,



eventually to agents. It is pretty clear that we added distribution and persistence to objects already. We can similarly add mobility to objects to get mobile objects and most mobile agents are just that. Information agents are generally active objects in a network that implement database-like functions to aggregate data. Traders exist in both the object and agent worlds, playing the same roles. The puzzling cases come when we add agent belief and goal systems, inference, learning, evolution, and ontologies. There does not seem to be an object correlate for these, not yet, at any rate.

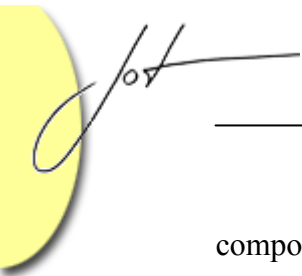
- If agents are going to be widely useful and common, then how do we go from the current state of practice where agents are in the minority and agent systems are often closed so that agents only communicate with other agents in their own society, to a next generation world where agent technology is comparably dominant to object technology and other possibly related technologies. If agents are to coexist with objects, what implementation changes can be made to add them seamlessly into programming environments like Java or C++? Can we find ways to piggyback agent technology onto other pervasive technologies like email, XML, distributed objects, databases, etc.? Some of the benefits might be scalability and immediate pervasiveness.

In some sense, the burden of getting along with other technologies is a problem for the agent camp - the sooner we provide migration strategies, the sooner agent technology can become more widely available and useful. A way to put the problem in perspective is, pretend you give some application developer an agent toolkit and also conventional tools like scripting languages. At what point will solutions commonly be developed using many of the tools from the agent toolkit? Will they just be used when they are useful, like other tools. For instance, inference might just be used occasionally (perhaps rarely); trading more often; mobility in certain families of situation; and so on.

6 AGENTS VERSUS OBJECTS CONCLUSION

Agents are autonomous entities that can interact with their environments. But, are they just objects with extra attributes or are they really an entirely different approach? And, just 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. In some sense, the burden of getting along with other technologies is a problem for both the agent and object camps.

The sooner we provide migration strategies, the sooner agent technology can become more widely available and useful. Some 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. For example, this might include communication factories, transport references, transport policies, directory elements, and agent factories.

In short, when we design systems, we can choose a well thought-out mixture from both approaches. We can even build aggregates where agents consist of both objects and other agents, and vice-versa. For Grady Booch, employing agents with object systems is useful because the agent-based approach[6]:

- a) provides a way to reason about the flow of control in a highly distributed system,
- b) offers a mechanism that yields emergent behavior across an otherwise static architecture, and
- c) codifies best practices in how to organize concurrent collaborating objects.

BIBLIOGRAPHY

- 1) [Parunak, Van Dyke 1997] Parunak, H. Van Dyke, *'Go to the Ant': Engineering Principles from Natural Agent Systems*, Annals of Operations Research, 75, 1997, pp. 69-101.
- 2) [Morley 1998] Morley, Dick, *Cases in Chaos: Complexity-Based Approaches to Manufacturing*, Embracing Complexity, Ernst & Young Center for Business Innovation, Boston, MA, August, 1998, pp. 97-102.
- 3) [Wooldridge et al 2000] Wooldridge, Michael, Nicholas R. Jennings, and David Kinny, *The Gaia Methodology for Agent-Oriented Analysis and Design*, Autonomous Agents and Multi-Agent Systems, forthcoming, 2000.
- 4) [Bradshaw 1997] Bradshaw, J. (ed.), *Software Agents*, MIT Press, Cambridge, MA, 1997.
- 5) [Odell 2000] Odell, James (ed.), *Agent Technology*, green paper, OMG Agent Special Interest Group, OMG document agent/00-09-01, 2000.
- 6) Booch, Grady, private communication, 2000.



About the author



James J. Odell is a consultant, writer, and educator in the areas of object-oriented and agent-based systems, business reengineering, and complex adaptive systems. He has written four books on object orientation and has two books in progress on agent-based system design. His website is www.jamesodell.com.