Objects and Agents: How do they differ?

by James Odell

Just how different are objects and agents? Some developers consider agents to be objects, but with more bells and whistles. This approach tends to define agents beginning with the phrase, "An agent is an object that …"—where 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 together in the development of software systems. This article discusses both the differences and similarities between agents and objects and lets you decide which viewpoint you want to choose. Whichever viewpoint you choose, I hope you will find that the agent-based way of thinking brings a useful and important perspective for system development.

INTRODUCTION

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.

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 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.” [1] 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 UML includes many of these notions. (For example, the <<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 like 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 upon 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 article will explain 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 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.

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 activity from being simply passive to entirely proactive. For example, while ants are basically reactive, they still 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.
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—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.

**Nondeterministic Autonomy**

Agents may also employ some degree of nondeterministic (or unpredictable) 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, the behavior of a shopping agent might be highly unpredictable. Sent out to choose, negotiate, and buy a birthday present for your mother-in-law, the agent might return with something odd indeed or with nothing at all. In other words, the agent can also say "no."\(^1\)

---

\(^1\) 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.
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, 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, “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 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. One method per message

**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 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 which is independent of the type of service interface (e.g., cost, reachability).
second involves an anonymizer that hides the identity of a requester from a service provider. Models that matchmake based on strong typing, such as CORBA, RMI, 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. 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 view 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. While object systems are 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 in which 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 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 that any other agent. If an agent can change itself, it can add (as well as subtract) features dynamically. For example, with genetic programming (GP) 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 programed 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—being organized, anarchic, or democratic. In short, nature can provide a rich trove of ideas for multiagent system design.

AGENTS AS A COMPUTATIONAL MODEL

The distinction between agent-based and non-agent-based could be viewed in terms of the computational model used to solve the problem. Just like the difference between object-oriented and structured programming, the same problem can be solved both ways. The resulting programs may look just the same externally. But object-oriented can supposedly deal better with complexity and adaptability. Similarly, agent-based programming should be able to deal with complexity and adaptability even better.

Agents, then, can be defined as an extension of objects. An object is something that encapsulates its identity (who), its state (what), and its behaviour (how). An active object encapsulates its own thread of control (when). A mobile object encapsulates (and controls) the location of its execution (where). An agent may have all of these, but it must have something else. What else can it encapsulate? (Note that many people already equate mobile object = agent, and some even equate active object = agent)

A common approach is that an agent encapsulates why it does something. Objects have a specific contract that defines their behaviour, but in an internet environment agents (like humans) can only attempt to get other agents to believe or do something. Agents, like humans, have only a generic interface to interact with others and the environment. Contracts between agents are negotiated at a meta-level, they are not implicit in the interface. This is where speech acts, trust, emergence, ethics and game theory come into play.

The agent community should provide models and techniques that will allow us to build systems on a very large or even global scale, where many agents interact, each trying to achieve their own goals, and still work as a whole. This is difficult, it may be the ultimate achievement in software engineering. Such a vision is not expected to become a reality for several years from now.

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. For example, 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 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. We can even build aggregates where 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

a) they provide a way to reason about the flow of control in a highly distributed system,
b) they offer a mechanism that yields emergent behavior across an otherwise static architecture, and

c) they codify best practices in how to organize concurrent collaborating objects.

REFERENCES


ACKNOWLEDGMENTS

The author would like to thank the reviewers of this article for their valuable feedback including Geoff Arnold, Conrad Bock, Grady Booch, Francis McCabe, Van Parunak, and David Taylor.

###

James Odell is a consultant, educator, and author. He can be reached by telephone at (734) 994-0833 or by email at jodell@compuserve.com. His website is www.jamesodell.com.