# Interaction-Oriented Software Development.

**Article** *in* International Journal of Software Engineering and Knowledge Engineering · June 2001

DOI: 10.1142/S0218194001000530 · Source: DBLP



**1 author:**

Michael N Huhns
University of South Carolina
**324** PUBLICATIONS **7,823** CITATIONS

International Journal of Software Engineering and Knowledge Engineering
Vol. 0, No. 0 (1994) 000–000


# INTERACTION-ORIENTED SOFTWARE DEVELOPMENT*

MICHAEL N. HUHNS
*Department of Computer Science and Engineering*
*University of South Carolina*
*Columbia, South Carolina 29208, USA*

Received December 19, 2000
Accepted March 12, 2001

This paper describes a new approach to the production of robust software. We first motivate the approach by explaining why the two major goals of software engineering—*correct* software and *reusable* software—are not being addressed by the current state of software practice. We then describe a methodology based on active, cooperative, and persistent software components, i.e., *agents*, and show how the methodology produces robust and reusable software. We derive requirements for the structure and behavior of the agents, and report on preliminary experiments on applications based on the methodology. We conclude with a roadmap for development of the methodology and ruminations about uses for the new computational paradigm.

*Keywords*: Agent-oriented programming; multiagent systems.

## 1. Introduction

Computing is in the midst of a paradigm shift. After decades of progress on representations and algorithms geared toward individual computations, the emphasis is shifting toward *interactions* among computations [34, 50]. The motivation is practical, but there are major theoretical implications. Current techniques are inadequate for applications such as ubiquitous information access, electronic commerce, and digital libraries, which involve a number of independently designed and operated subsystems. The metaphor of interaction emphasizes the autonomy of computations and their ability to interface with each other and their environment. Therefore, it can be a powerful conceptual basis for designing solutions for the above applications.

Unfortunately, the field of software engineering has been progressing slowly. This should not be surprising, for three reasons:

1. Software systems are the most complicated artifacts people have ever attempted to construct

*This research was supported by the National Science Foundation, IIS Division, Computation and Social Systems Program, under grant no. IIS-0083362.

2. Software systems are (supposedly) guaranteed to work correctly only when *all* errors have been detected and removed, which is infeasible in light of the above complexity
3. The effect of an error is unrelated to its size, i.e., a single misplaced character out of millions can render a system useless or, worse, harmful.

### 1.1. *Progress in Software Engineering*

Software engineering concerns both the process of producing software and the software that is produced. The major goal for the software is that it be correct, and the major goal for the process is that it be conducted efficiently. One fundamental approach to meeting these goals is to exploit modularity and reuse of code. The expectations are (1) that small modules are easier to debug and verify, and therefore more likely to be correct, (2) that small modules will be more likely to be reused, and (3) that reusing debugged modules is more efficient than coding them afresh. A few examples of software engineering practice based on this approach are the following [4]:

- Parameterized subroutines provide code reuse within an application
- Libraries of subroutines encourage code sharing across applications
- Object-oriented methods allow tailoring of library routines via inheritance and polymorphism
- Client/server paradigms, such as the world-wide web, ODBC, OLAP, and SQL databases, permit sharing of data across platforms
- Remote procedure calls, such as Sun's Java RMI and Microsoft's COM, enable code to be shared across platforms
- Transaction processors, such as Tuxedo and Encina++, enable transactions to be shared
- Distributed object technologies, such as OMG CORBA and Microsoft's .NET, allow sharing of tailorable code across platforms.

Programming paradigms have evolved from machine language in the 1950's, procedural programming in the 1960's, structured programming in the 1970's, and object-based and declarative programming in the 1980's. In the 1990's, methods for structuring collections of objectswere developed, including frameworks, design patterns, scenarios, and protocols.

However, software has not kept pace with the increased rate of performance for processors, communication infrastructure, and the computing industry in general [30]. Whereas processor performance has been increasing at a 48% annual rate and network capacity at a 78% annual rate, software productivity has been growing at a 4.6% annual rate and the power of programming languages and tools has been growing at an 11% annual rate. CASE tools, meant to formalize and promote software reuse, have not been widely adopted [20]. By a different metric, the industry standard for good commercial software is approximately six defects per KLOC (thousand lines of code), and this rate has held constant for decades [14].

The procedural and declarative approaches to programming suffer from being primarily line-at-a-time techniques, with a basis in functional decomposition. Object technology improves these by replacing decomposition with inheritance hierarchies and polymorphisms. It enables design reuse of larger patterns and components [2]. However, inheritance and polymorphism are just as complex and error prone as decomposition, and the great complexity of interactions among objects limits their production and use to a small community of software engineers. By focusing on encapsulating data structures into objects and the relationships among objects, it supports a data-centric view that makes it difficult to think about sequences of activity and dataflow. Scenarios overcome this difficulty by depicting message sequences and threads of control, but they are not well supported by current object languages. Table 1 summarizes the major features of existing software paradigms, and the features promised by the multiagent-based approach described below.

**Table 1.** Features of Programming Languages and Paradigms (from [30])

| Concept | Procedural Language | Object Language | Multiagent Language |
|---|---|---|---|
| **Abstraction** | Type | Class | Service |
| **Building Block** | Instance, Data | Object | Agent |
| **Computation Model** | Procedure/Call | Method/Message | Perceive/Reason/Act |
| **Design Paradigm** | Tree of Procedures | Interaction Patterns | Cooperative Interaction |
| **Architecture** | Functional Decomposition | Inheritance and Polymorphism | Managers, Assistants, and Peers |
| **Modes of Behavior** | Coding | Designing and Using | Enabling and Enacting |
| **Terminology** | Implement | Engineer | Activate |

## 1.2. *A New Software Paradigm*

We believe it is time to consider a completely different approach to software systems. We propose one based on the (intentionally provocative) recognition that

- errors will *always* be a part of complex systems
- error-free code can at times be a *disadvantage*
- where systems interact with the complexities of the physical world, there is a concomitant *power* that can be exploited.

We suggest an open architecture consisting of redundant, agent-based modules [6]. The appropriate analogy is that of a large, robust, natural system. We motivate our approach by means of the following four examples.

### 1.2.1. *Example 1: Avoiding Deadlocks and Livelocks*

Sometimes, when two people approach each other on a narrow sidewalk, they move from side-to-side in unison a few times until they find a way to pass. Now, imagine two robots in a similar situation: if they are each programmed identically and accurately, then they might move in unison and be deadlocked forever. If, however, one had a small flaw in its programming, then it would eventually act differently and break the deadlock.

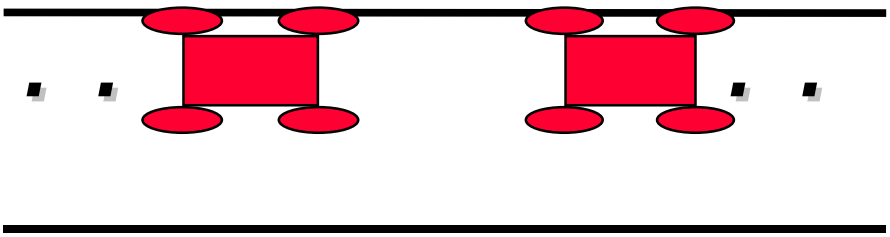

**Fig. 1.** Robots meeting in a hallway might move in unison and "livelock," unless one operates differently than the other

This example illustrates a key concept: *errors can sometimes make a system more robust.* Individual components do not have to be perfect, if there are a sufficient number of them, if their capabilities are basically sound, and if their responsibilities overlap.

Such deadlock behavior is actually quite common-it can occur anytime two processes access a common resource, e.g., when two applications attempt to update a database or communicate over a channel at the same time. When the possibility of the deadlock is known in advance, a solution is to deliberately introduce uncertainty into one or both of the processes; this is the basis for conflict resolution in the CSMA/CD Ethernet protocol.

### 1.2.2. *Example 2: Forming a Circle*

Consider asking a group of children to form a circle. This they will be able to do, relatively independent of the number of children, their sizes, and their ages, without requiring any further directions as to who should stand where. The formation of the circle will be robust with respect to the removal or addition of children. It will even accommodate a few children who do not understand the request. This "circle algorithm" succeeds because each element of the solution is intelligent and autonomous, and possesses basic knowledge of the problem domain. Each element is not, however, required to be perfect.

Contrast this with a conventional approach to developing software for arranging items in a circle. A programmer would first define classes for the items, with attributes describing their size and shape. The programmer would then construct a central control module that, using trigonometry, would compute the precise locations for each of the items. The control module would have to be written to accommodate an arbitrary number of items having a variety of sizes and shapes. Changing any one of the parameters would require the control module to recompute the locations of all items. More significantly, changing the way in which the shape or size of an item is defined would require the control module to be rewritten. (For example, if the control module expected items to be defined in terms of their length and width, then it would have to be modified to handle items defined in terms of their radius.)

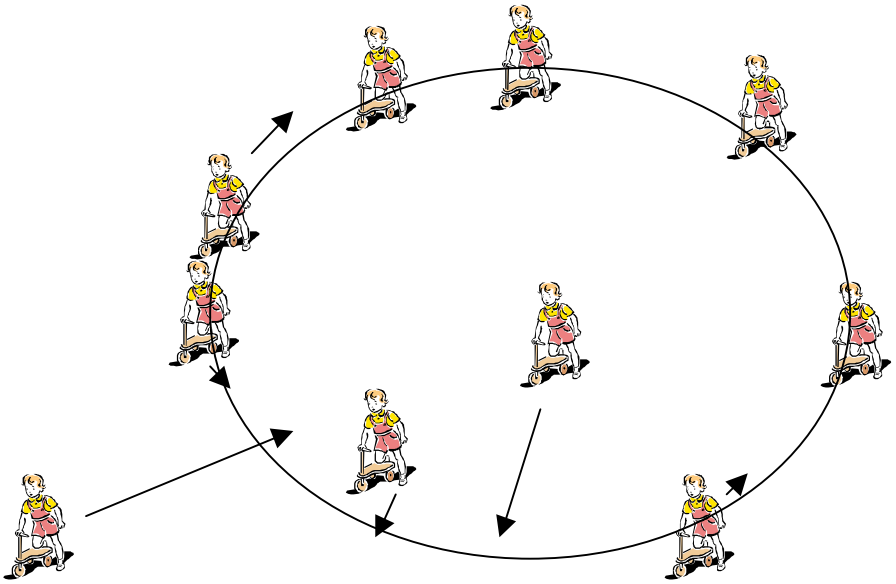

**Fig. 2.** Children (and autonomous agents) can be a robust circle-forming algorithm

1.2.3. *Example 3: Navigating on Mars*

Consider an autonomous vehicle roaming on Mars. There is a very simple algorithm that enables the vehicle to maneuver around obstacles [33]: when an obstacle is encountered, the vehicle

1. Backs up 1 meter
2. Turns clockwise 90 degrees
3. Moves forward 1 meter
4. Turns counterclockwise 90 degrees
5. Goes forward on its original course.

Although in theory it appears that the vehicle can easily become trapped, in practice the vehicle is able to wriggle through any configuration of obstacles that it can physically fit between, because it cannot move *exactly* 1 meter or turn *exactly* 90 degrees. Its errors in these motions give it the variability it needs to move eventually in just the right way to go around an obstacle. Surprisingly, attempts to increase its precision not only increase its complexity, but also make it more likely to become trapped. In essence, *reducing errors can make a system less robust.*

1.2.4. *Example 4: Business Software Objects—Avoiding a Pay Cut*

As a more general and fundamental example, most business software components are intended to be models of some real object within the business, such as an employee. A problem is that, unlike the entities they represent, conventionally implemented components are passive. Why is this a problem? If someone accidentally reduced

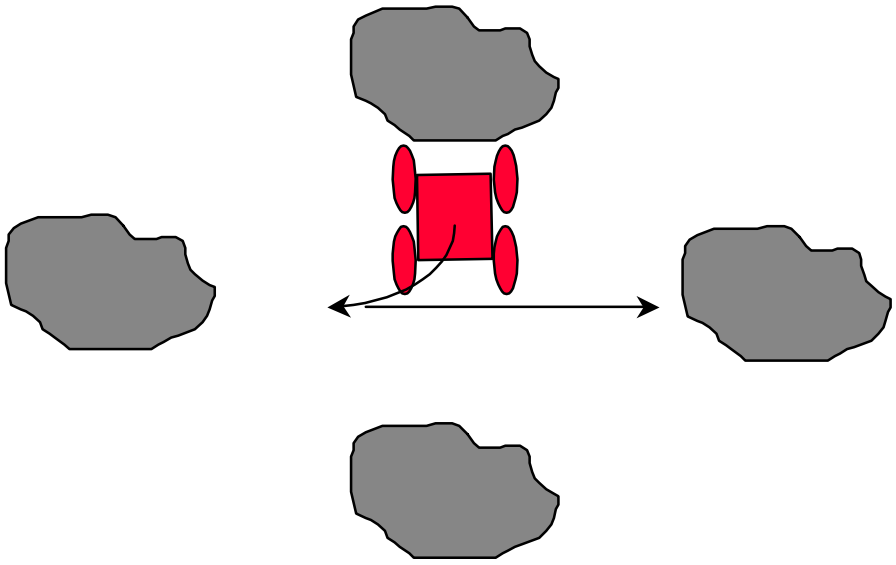

**Fig. 3.** A robot navigating on Mars can wriggle between obstacles via a very simple algorithm that takes advantage of errors in its movement through the environment

the salary of an employee by 50%, a conventional software component would not protest. Like real employees, agents implemented as components with the extra ability to take action would not allow such accidents. As we describe next, agents can also do a lot more.

## 2. Interaction-Based Software Development

The behavior of any system depends on its construction and the environment in which it operates. When the system contains a number of components that interact with each other and a complex environment, the behavior can be difficult to predict and control. Traditional software interfaces are rigid. Often the slightest error in the implementation of a component can have far-reaching repercussions on the behavior of the entire system. However, the output of a component may be erroneous because of its malfunctioning, its environment being out of its design range, or an erroneous input from another component. Traditional approaches for software or hardware fault tolerance are rigid in that they use fixed means, e.g., averaging or voting, to correct errors.

By contrast, we are developing an approach in which the interactions among components are defined in a more robust manner using higher-level abstractions such as social commitments and team intentions. These abstractions enable us to design the components to be more flexible toward their inputs and outputs. Moreover, in real-life situations, a component may be forced to release results that are almost certainly erroneous—it may lack the time and resources to await definite inputs and to process them properly. Our approach can handle these situations

naturally, whereas traditional approaches are incapable of even representing such situations.

Our approach presupposes that the components are able to enter into social commitments to collaborate with others, to change their mind about their results, and to negotiate with others. They must be long-lived (to even detect errors that manifest later in the execution) and persistent (to resolve them). In other words, the components are interacting agents functioning in teams. The agents can detect not only errors, but also opportunities in general. They can volunteer to take advantage of those opportunities, to form teams, negotiate solutions, and enact them in a persistent manner. One risk with such systems is that their persistence may get them into livelocks where interactions prevent progress. It is essential that the agents be able to explore their way out of livelocks. Interestingly, "errorful" behavior by some members of the team can facilitate this exploration, especially in complex environments where the concurrently executing mix of agents is determined dynamically.

Our approach is based on a number of important tenets:

| | |
|---|---|
| Interaction | Persistent action |
| Teamwork founded on social commitments | Negotiation |
| Exploration | Error tolerance and exploitation for robustness |

Although some of these tenets are shared with some recent approaches, e.g., aspect-oriented and agent-oriented programming [40, 3], no existing approach captures all of them. It appears desirable to try to exploit their synergistic mix.

### 2.1. *Requirements for a New Class of Applications*

Thanks to ongoing advances in computer systems, new classes of applications are evolving. These applications require a number of important properties beyond traditional approaches.

- **Disintermediation (the direct association between users and their software [39]).** Providing a user with seamless access to and interaction with remote information, application, and human resources requires a distributed active-object architecture [52].
- **Dynamic composability and execution.** A system should execute as a set of distributed parts, but the resources required will be mostly unknown until run-time: this requires an infrastructure to enable their discovery and composition as needed.
- **Interaction.** There might be subtle and critical patterns of interaction among the components, but the specific interactions may be unknown until run-time, and may vary: this requires that the patterns of interaction be explicitly represented and reasoned with. There is recent, significant work on the power of interactions [1, 48].
- **Error tolerance and exploitation.** As the deployed systems become increasingly complex, they should not only tolerate, but where possible exploit, errors in their components.

Two major convergences now give us the means to address the above requirements. First, large information environments dealing not only with information, but also with the physical world are available to provide crucial computing and communication resources, as well as ready contact with reality. Second, technical advances in computer science provide a foundation for agent architecture and languages. These advances go hand-in-hand, because the existence of the expanding infrastructure changes the trade-offs in carrying out the dictates of the science.

A recent computing paradigm is based on Java, and the ability it provides for users and applications to download the specific functionality they want at the moment they request it. In particular, Java Beans possess two interfaces: one that governs the interaction of a bean with its environment at run-time, and a second that describes the behavior of the bean to developers at program-creation or compile time. DCOM provides a similar capability for COM objects. Such capability is leading to the rise of a software-component industry, which will produce and then distribute on demand the components that have a users' desired functionality [57]. Each user can be presented with a unique customized environment. However, because of this uniqueness, how can component providers be confident that their components will behave properly [5]? This is a problem that can be addressed by agent-based components that *actively* cooperate with other components to realize the user's goals and that express their behavior in terms of their intentions and commitments.

### 2.2. *Agent-Based Software*

Programming based on teams of agents will build on results generated by a large number of researchers. In particular, efforts under the DARPA CoABS program for developing middle agents, wrappers, and agent communications form one of the foundations for our work. We extend the efforts into a complete programming paradigm with a formal semantics. Our extensions and formal semantics are based on the work on agent-oriented programming by Shoham, Wooldridge, and Jennings [40, 54, 53, 55, 21].

A wide variety of software programs have been developed that are characterized as software agents [18]. One category of such agents focuses on the interaction between a user and a computing environment. A second category of agent-based software is focused on the interaction among computing agents. The basic issues addressed concern interoperability among geographically distributed agents executing on heterogeneous platforms. There are two different approaches for communication among the agents. The procedural scripting approach causes execution of a remote task by sending a procedural script for interpreted execution at the remote site. Examples of this approach, termed agent mobility, are Voyager [http://www.objectspace.com/products/voyager/] and Aglets [29]. The declarative approach takes the view that only a declarative description of the task should be sent to the remote site. An example of this approach is ACL [12].

What we are developing differs from current work in software agents in that

- We are not researching new agent capabilities per se
- We are not developing an agent-based system for some new application domain
- We are investigating how agents can be the fundamental building blocks for the construction of general-purpose software systems, with the anticipated benefits of robustness and reuse
- We are characterizing agents in terms of *mental abstractions*, and multiple agents in terms of their *interactions:*

Mental abstractions for agents are beliefs, knowledge, desires, goals, and intentions, whereas multiagent abstractions are

- Social: about collections of agents
- Organizational: about teams and groups
- Ethical: about right and wrong actions
- Legal: about contracts and compliance

These abstractions matter because modern applications go beyond traditional metaphors and models in terms of their dynamism, openness, and trustworthiness. They involve virtual enterprises and electronic commerce, such as in manufacturing supply chains and autonomous logistics, community-ware and social interfaces, and problem solving by collaborative groups. The architecture of future information systems will be agent-oriented, as shown in Figure 4.

Techniques for creating and maintaining societies of autonomous active objects (agents) will be useful not only for large open information environments, but also for large open physical environments. For example, new efficiencies in logistics could result from considering each supply item being deployed to be intelligent (implemented via a "smart card") with a local goal to reach a destination and an ability to take advantage of a global distribution system.

Such information environments are too complex to be centrally developed or controlled. The only alternative is for intelligence to be embedded at many places to provide distributed management. Each locus of intelligence is an autonomous *agent* that must be *long-lived* (to execute unattended for long periods), *adaptive* (to explore and learn about its environment), and *social* (to interact with others to leverage knowledge and capabilities, so as to achieve individual as well as collective goals). Composed as they are of active social entities, *multiagent systems* are ideally suited to the challenges of software development described above. *Teams*, with different members playing specific roles and cooperating to achieve some higher end, emphasize the social and organizational aspects of multiagent systems.

### 2.3. *System Redundancy and Adaptation*

In some circumstances, robustness in the presence of errors is governed by redundancy. That is, if each software module is deemed to be behaving either correctly or incorrectly, then two modules with the same intended functionality are sufficient to detect an error in one of them, and three modules are sufficient to correct

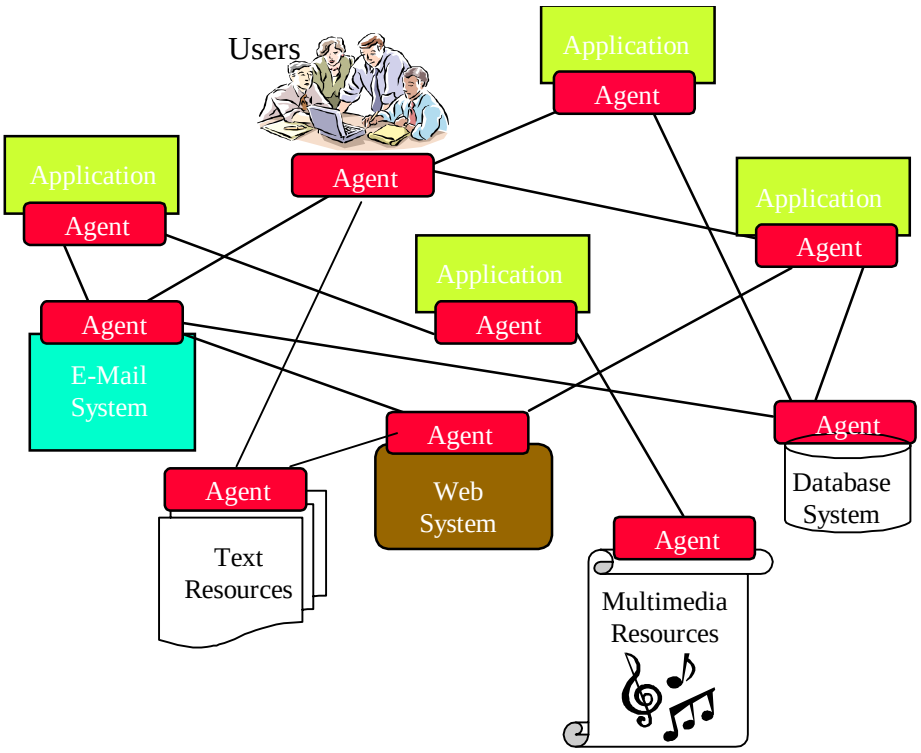


**Fig. 4.** Architecture for an agent-oriented information system, indicating collaborations among users, applications, and resources

the incorrect behavior. Fundamentally, the amount of redundancy required is well specified by information and coding theory. Exemplifying this, HP Labs has built a massively parallel computer with 220,000 known defects, but it still yields correct results [7]. As long as there is sufficient communication bandwidth to find and use healthy resources, it can tolerate the defects. Allowing so many defects enables the computer to be built cheaply.

Similarly, a National Research Council committee last year, in addressing the problem of software security, published a report called Trust in Cyberspace, which advocated the "Theory of Insecurity." The theory suggests that acceptably secure systems can be built out of components that have known vulnerabilities and security holes [25].

When software modules exhibit more complex behavior, then deeper reasoning is needed to determine whether or not the behavior is correct. This requires agents to communicate their intentions and commitments. They can then be monitored to determine if they have acted according to their intentions and have kept their commitments. Activating a group of agents then becomes a type of nondeterministic programming.

Self-adaptive software [28] evaluates its own behavior and changes the behavior

when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible. This implies that the software has alternative ways of accomplishing its purpose, along with enough knowledge of its construction and awareness of its current operation to enable effective changes to be made at runtime. Self-adaptive software requires components to maintain models of themselves and the other components with which they might interact [23]. In a control-system metaphor, runtime software is treated like a factory, with a monitoring and control facility that manages the factory to improve its performance [27].

Intentional programming attempts to coordinate the cooperation of independently developed abstraction objects, termed intentions. Intentions are not executed at runtime, but are called at programming time [41, 49].

### 2.4. *Agent Capabilities*

Figure 4 illustrates how agents might represent, i.e., act on behalf of, various kinds of passive or non-agent like components and entities in an environment, and how they might interact to provide next-generation services to users and applications. Success in this requires that

- Agents stay aware of their own roles, capabilities, and weaknesses by maintaining a model of themselves
- Agents stay aware of their team by maintaining models of its members and their roles
- Agents maintain models of other teams in which they might play a role
- Agents learn from interactions about the goals, capabilities, and intentions of other agents
- Agents rely on commitments from other agents, and maintain commitments to other agents

### 2.5. *Ontologies: Modeling Objects, Resources, and Agents*

A key to enabling agents to interact productively is for them to construct and maintain models of each other, as well as the passive components in their environment. Unfortunately, the agents' models will be mutually incompatible in syntax and semantics, not only due to the different things being modeled, but also due to mismatches in underlying hardware and operating systems, in data structures, and in usage. In attempting to model some portion of the real world, information models necessarily introduce simplifications that result in semantic incompatibilities.

Ontologies appear to be well suited for reconciling heterogeneous semantics. We have been developing mediating mechanisms based on domain-specific ontologies to yield the appearance and effect of semantic homogeneity among agents at the knowledge level [35]. However, if there are n entities in the environment, then each would need a model of each of the other entities, resulting in $n(n-1)/2$ models that must be maintained. This is infeasible for large domains. We solve this via

two means. First, agents maintain and advertise models of themselves, resulting in a total of n models. Second, we consider the source of the models. How should one agent represent another, and how should it acquire the information it needs to construct a model in that representation?

This has, we believe, a simple and elegant answer: *the agent should presume that unknown agents are like itself, and it should choose to represent them as it does itself.* Thus, as an agent learns more about other agents, it only has to encode any differences that it discovers. The resultant representation can be concise and efficient, and has the following advantages:

- An agent has a head start in constructing a model for a just-encountered agent.
- An agent has to manage only one kind of model and one kind of representation.
- The same inference mechanisms it uses to reason about its own behavior can reason about the behaviors of other agents; an agent trying to predict what another will do has only to imagine what it itself would do in a similar situation.
- As information about other agents is acquired through observations and interactions, models of them can be updated, and will diverge from the default.

We portray an agent as a rational decision-maker that perceives and interacts with its environment. Agents are rational in the context of all other agents, because they are aware of the other agents' constraints, preferences, intentions, and commitments and act accordingly.

## 3. Semantics

If agents are constructed modularly, the challenge is in specifying and generating the right interactions. We term our approach *interaction-oriented programming (IOP)*, and include in it high-level abstractions and techniques that capture the structure of the desired interactions. We identify three layers of IOP, from lower to upper:

- *Coordination*, which enables the agents to operate in a shared environment
- *Commitment*, which reflects the agents' obligations to one another, capturing their social structure and the norms governing their behavior
- *Collaboration*, which supports reaching agreement, forming and maintaining teams, and performing complex joint activities.

Informal concepts, such as competition, often have variants that may be classified into different layers. For example, bidding in an auction requires no more than coordination, whereas commerce involves commitments, and negotiation involves protocols for collaboration.

Pieces of the above layers have been studied in distributed computing, databases, and distributed artificial intelligence (DAI), but usually not from a programming perspective. The distributed computing and database work focuses on narrower

problems of synchronization, and eschews high-level concepts such as social commitments [11]. Thus it is less flexible, but more robust, than the DAI work. Our contribution will be in enhancing and synthesizing ideas into a framework [31] that is *rigorous* yet flexible.

The next section provides insights into how we are achieving such a framework. In particular, we are providing abstractions for agents that enable them to be specified at the level of intentions [8], commitments, and organizations.

## 4. Preliminary Results and Discussion

In preliminary experiments, we have constructed a large group of agents, each implemented as a concurrently executing Java thread and interacting through a base class environment. The agents each have an understanding of what a circle is, what it means to be part of a circle, where the nearest agents are located, and an estimate of how close the group is to being in a circle. The agents have the ability to reason about where they should be on a circle and the direction they should move to get there. They also have the ability to help move nearby agents that do not seem to be moving properly or in the right place. Into this environment, we have introduced a few agents that do not have the ability to become part of a circle without help or are stationary. The group overcomes this and produces an acceptable circle. We have anecdotal evidence, via one comparison, that such an implementation can be constructed more rapidly and robustly than a conventional object-oriented implementation in C++.

### 4.1. *Generic Agent Architectures*

To implement systems such as the circle algorithm, the agent components must have an application programming interface (API) known to each other and must support agent-abstraction features that enable and foster the coordination, commitment, and collaboration layers described above. Further, an understanding of the architecture of an agent is a prerequisite for successful implementation [26, 36]. To better communicate the requirements for the implementation of agents who will participate in one of our systems, we provide UML diagrams for agent architectures [45]. However, before we describe these diagrams, we need to review implementations for the basic features of agents. Consider the architecture in Figure 5 for a simple agent interacting with an environment [38].

This architecture describes a simple agent that senses its environment, decides upon an action based on what it has sensed, and then carries out the action through its effectors. Note that the sensory input can include received messages and the action can be the sending of messages.

In order to construct an agent, we have to know what one is in more detail [17]. In particular, if we are going to construct one using conventional object-oriented analysis and design techniques, we should know in what ways an agent is more than just a simple object. The features of an agent that are relevant from an

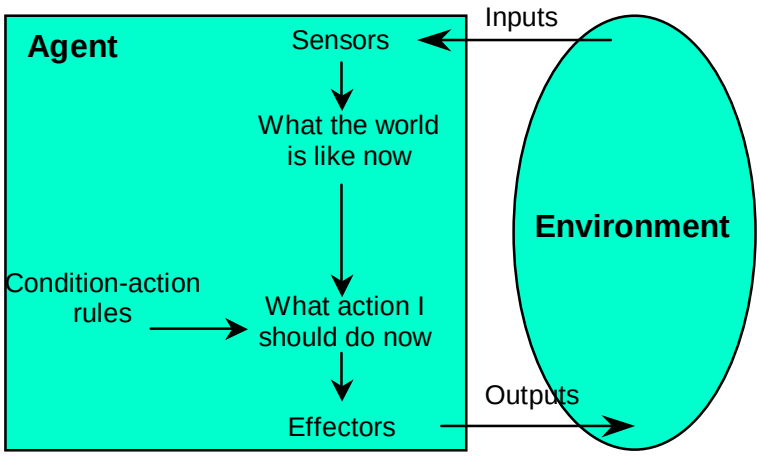


**Fig. 5.** The architecture of a simple reactive agent

implementation standpoint are *unique identity, proactivity, persistence, autonomy,* and *sociability* [51]. We consider each of these features in turn.

An agent inherits its unique identity simply by being an object. To be proactive, an agent must be an object with an internal event loop, such as any object in a derivation of the Java thread class would have. Simple pseudocode for an event loop, where events arrive from a sensing of the environment, is

```
Environment e;
RuleSet r;
while (true) {
    state = senseEnvironment(e);
    a = chooseAction(state, r);
    e.applyAction(a);
}
```

Notice that this is an infinite loop, which provides the agent with persistence as well. If agents were ephemeral, then it would be difficult for them to converse with one another and they would be, by necessity, asocial. Additionally, persistence makes it worthwhile for agents to learn about and model each other. For agents to gain the benefits of such modeling, they need to be able to distinguish one agent from another, and thus must possess unique identities.

Autonomy for an agent is akin to free will for a person. It enables an agent to decide its actions for itself. For an agent constructed as an object with methods, autonomy can be implemented by declaring all of the methods *private*. With this restriction, only the agent can invoke its own methods, under its own control, and no external object can force the agent to do anything it does not intend to do. Other objects can communicate with the agent by creating events or artifacts, especially messages, in the environment that the agent can perceive and react to.

Sociability is achieved by enabling an agent to converse with other agents. The

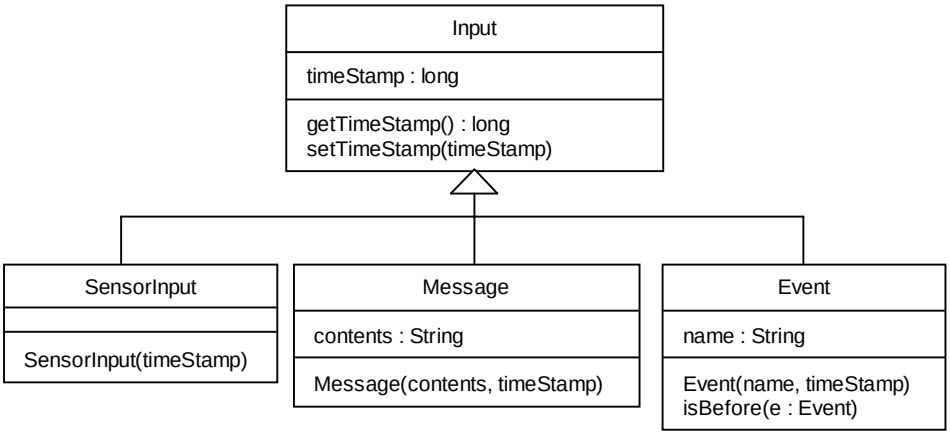


**Fig. 6.** Agents must be able to react to various types of inputs from their environment

conversations, normally conducted by sending and receiving messages, provide opportunities for agents to coordinate their activities and cooperate, if so inclined. This can be achieved by generalizing the input class of objects an agent might perceive to those shown in Figure 6.

The input the agent receives can be either a piece of sensory information, a message from another agent, or an event defined by the agent. Events are simply "reminders" that the agent sets for itself. For example, an agent that wants to wait five minutes for a reply would set an event to fire after five minutes. If the agent receives the reply before the event, then it can disable the event. If it receives the event, then it knows it did not receive the reply in time and can proceed accordingly.

The UML diagram in Figure 7 provides a general framework for implementing a belief-desire-intention (BDI) architecture [37] using an object-oriented language [10, 24]. Some illustrative pseudocode follows:

```
Agent::run() {
Environment e;
e.run(); start environment in its own thread
while (true) {
   I = chooseIntention();
   if (I.execute()) // true if goal was achieved
      D.remove(I.goal);    }}

Environment::run(){
for (a ∈ Agent) {
   while (true) {
       a.B.incorporateNewObservations(getInput(w));
       if ( !a.currentIntentionIsApplicable() )
          a.stopCurrentIntention();
       sleep(timeInterval);    }}}
```

The agent's run method consists of finding the best applicable intention (plan) and executing it to completion. If the intention returns true, it means the goal was achieved, so the goal is removed from the desire (goal) set. If the environment thread finds that an executing intention is no longer applicable and calls for a stop, the intention will promptly return from the `execute()` call with a value of `false`. Notice that the environment thread modifies the agent's set of beliefs. The belief set needs to synchronize these changes with any changes the intentions make to the beliefs.

### 4.2. *Behaviors and Activity Management*

Most popular agent architectures include a set of behaviors and a method for scheduling them. A behavior is distinguished from an action in that an action is an atomic event, while a behavior can span a longer period of time. In multi-agent systems, we can also distinguish between physical behaviors that generate actions, and conversations between agents that generate communicative acts. We can consider behaviors and conversations to be classes inheriting from an abstract activity class. We can then define an activity manager responsible for scheduling activities.

A complete agent-based system also requires an infrastructure to provide for message transport, directory services, and event notification and delivery. These are usually provided as operating system services or, increasingly, in an agent-friendly form by higher level distributed protocols such as Jini, Bluetooth, and FIPA's (the Foundation of Intelligent Physical Agents) emerging standards.

### 4.3. *The Team-Oriented Paradigm*

Using an architecture such as that described above, a developer will program and activate a team by resolving who (role) will do what (subtask), when (coordination), how (capabilities), where (resources or location), and why (team plan and external requirements) [15, 44]. In addition, there are the aggregate matters of how many agents are to be assigned to each role and how much resources are needed. The main steps are agent *creation* (compilation), team *configuration* (linkage), and team *activation* (execution).

The above matters presuppose an agent factory with rich protocols for discovery and software configuration that inherently accommodate flexibility through negotiation. In a general setting, the agents could join and activate teams with minimal programmer intervention. Their negotiated commitments to one another would lead to coordinated and coherent action by the entire team even as the membership of the team evolves and some members behave imperfectly.

We believe that implementing software as a large number of intelligent, but not perfect agents will be successful. Our approach imposes requirements on the structure and behavior of the agents, and facilitates a formal semantics. We will supply the metamodel, architecture, and formal semantics to realize this approach.

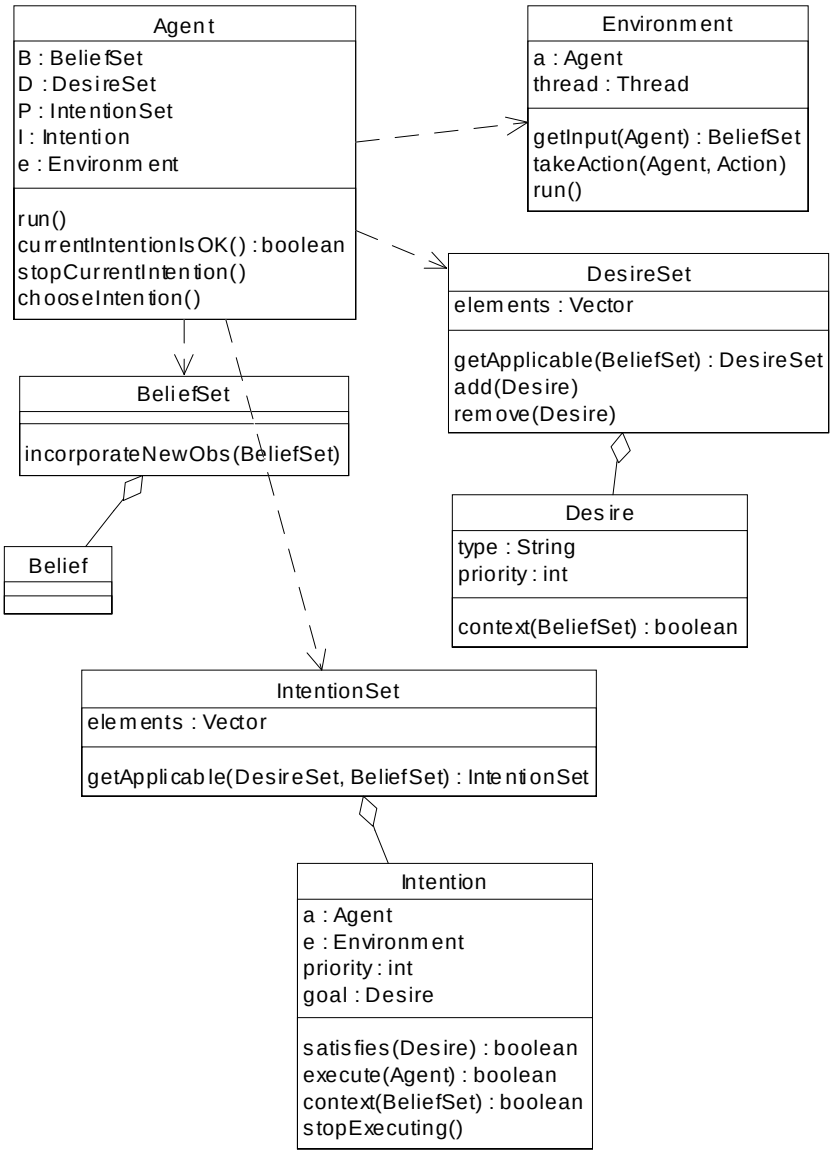


**Fig. 7.** Diagram of a belief-desire-intention architecture for an agent

Research prototypes are being developed using an iterative process called User-Centered Software Engineering.

## 5. Conclusion

We have proposed and begun developing a new software development paradigm—a cooperative paradigm—based on interacting agents, active objects, and active wrappers of legacy components. The resultant methodology and language, termed interaction-oriented programming, represent significant extensions of earlier methodologies, with greater expressive power, different conceptual foundations, such as the beliefs held by the components, and new modeling techniques.

Techniques for creating and maintaining societies of autonomous active objects (agents) will be useful not only for large open information environments, but also for large open physical environments. For example, such techniques would yield new efficiencies in logistics: by considering each item of material to be an intelligent entity (possibly via a "smart card") whose goal is to reach a destination, a distribution system could manage more complicated schedules and surmount unforeseen difficulties. Languages are required for creating and maintaining such environments—an interaction-oriented programming language satisfies this requirement.

Just as today almost anyone can create a web page and contribute information to the Web, so the proposed paradigm will enable anyone to create and contribute customized components to software applications [43]. We are in the midst of a trend toward *disintermediation*—the direct association between users and their software—that enables people to be responsible for their own computing, often without formal training or the support of professional intermediaries. This is healthy, but an infrastructure such as we propose is needed that can

- Analyze component interoperability and then cope with incompatibility
- Support the dynamic reconfiguration of loosely confederated processes and agents
- Monitor and manage persistent autonomous processes (extending the notion of daemons).

It is claimed that the major impediment to the realization of component-based development is *quality* of the components [32]. The proposed paradigm mitigates this through massive redundancy, leading to increased robustness. (A system that is stuck and making no progress can try one of its less popular alternatives.)

## Acknowledgements

This research was supported by the National Science Foundation, IIS Division, Computation and Social Systems Program, under grant no. IIS-0083362.

## References

1. Jean-Marc Andreoli, Paolo Ciancarini, and Remo Pareschi, "Interaction Abstract Machines," in Gul Agha, Peter Wegner, and Akinori Yonezawa, eds., *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, Cambridge, MA, 1993, pp. 257–280.
2. Antoine Beugnard, Jean-Marc Jezequel, Noel Plouzeau, and Damien Watkins, "Making Components Contract Aware," *IEEE Computer*, Vol. 32, No. 7, July 1999, pp. 38–45.
3. F-C. Cheong, "OASIS: An Agent-Oriented Programming Language for Heterogeneous Distributed Environments," Ph.D. Thesis, University of Michigan, 1992.
4. Adam Cheyer, "Agent-Based Interoperation," CSLI Seminar Series on Intelligent Agents, Stanford University, April 27, 1995.
5. Cynthia Della Torre Cicalese and Shmuel Rotenstreich, "Behavioral Specification of Distributed Software Component Interfaces," *IEEE Computer*, Vol. 32, No. 7, July 1999, pp. 46–53.
6. Helder Coelho, Luis Antunes, and Luis Moniz, "On Agent Design Rationale," in *Proceedings of the XI Simposio Brasileiro de Inteligencia Artificial (SBIA)*, Fortaleza (Brasil), October 17–21, 1994, pp. 43–58.
7. Peter Coffee, "Perfect Computers Cost Too Much," *PC Week*, July 6, 1998, p. 54.
8. Philip R. Cohen and Hector J. Levesque, "Persistence, Intention, and Commitment," in *Intentions in Communication*, Philip R. Cohen, Jerry Morgan, and Martha E. Pollack, eds., MIT Press, 1990.
9. Martin Fowler, *UML Distilled, 2nd Edition: A Brief Guide to the Standard Object Modeling Language*, Addison-Wesley, Reading, Mass, 2000.
10. E. Gamma et al., *'Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.
11. Les Gasser, "Social conceptions of knowledge and action: DAI foundations and open systems semantics," *Artificial Intelligence*, Vol. 47, 1991, pp. 107–138.
12. Michael Genesereth and Stephen Ketchpel, "Software Agents," *Communications of the ACM*, Vol. 37, No. 7, 1994, pp. 48–53.
13. R. Goodwin, "Formalizing Properties of Agents," Technical Report CMU-CS-93-159, Department of Computer Science, Carnegie-Mellon University, 1993.
14. Les Hatton, "Does OO Sync with How We Think?" *IEEE Software*, May 1998, pp. 46–54.
15. Michael N. Huhns, "Agent Teams: Building and Implementing Software," *IEEE Internet Computing*, Vol. 4, No. 1, January/February 2000, pp. 90–92.
16. Michael N. Huhns, "Multiagent-Oriented Programming," *Intelligent Agents and Their Potential for Future Design and Synthesis Environments*, Ahmed K. Noor and John B. Malone, editors, NASA Langley Research Center, Hampton, VA, February 1999, pp. 215–238.
17. Michael N. Huhns and Munindar P. Singh, "A Multiagent Treatment of Agenthood," *Applied Artificial Intelligence: An International Journal*, Vol. 13, No. 1–2, January–March 1999, pp. 3–10.
18. Michael N. Huhns and Munindar P. Singh, eds., *Readings in Agents*, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1997.
19. Michael N. Huhns, editor, *Distributed Artificial Intelligence*, Pitman/Morgan Kaufmann, 1987.
20. Juhani Iivari, "Why Are CASE Tools Not Used?" *Communications of the ACM*, Vol. 39, No. 10, October 1996, pp. 94–103.
21. Nicholas R. Jennings, "On Agent-Based Software Engineering," *Artificial Intelligence*, Vol. 117, No. 2, 1999, pp. 277–296.
22. Nicholas R. Jennings, "Commitments and conventions: The foundation of coordination in multi-agent systems," *The Knowledge Engineering Review*, Vol. 2, No. 3, 1993, pp.

223–250.
23. Gabor Karsai and Janos Sztipanovits, "A Model-Based Approach to Self-Adaptive Software," *IEEE Intelligent Systems*, Vol. 14, No. 3, May/June 1999, pp. 46–53.
24. Elizabeth A. Kendall, Margaret T. Malkoun, and Chong Jiang, "Multiagent System Design Based on Object-Oriented Patterns," *Journal of Object-Oriented Programming*, June 1997, pp. 41–47.
25. Stephen Kent, "An Interview with Stephen Kent," *IEEE Spectrum*, Vol. 37, No. 1, January 2000, p. 37.
26. David Kinny and Michael Georgeff, "Modelling and Design of Multi-Agent Systems," in J.P. Muller, M.J. Wooldridge, and N.R. Jennings, eds., *Intelligent Agents III — Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*, Springer-Verlag, Berlin, 1997, pp. 1–20.
27. Mieczyslaw M. Kokar, Kenneth Baclawski, and Yonet A. Eracar, "Control Theory-Based Foundations of Self-Controlling Software," *IEEE Intelligent Systems*, Vol. 14, No. 3, May/June 1999, pp. 37–45.
28. Robert Laddaga, "Creating Robust Software through Self-Adaptation," *IEEE Intelligent Systems*, Vol. 14, No. 3, May/June 1999, pp. 26–29.
29. Danny Lange and Mitsuru Oshima, *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley, Boston, MA, 1998.
30. Ted Lewis, "The Next 10,0002 Years: Part II," *IEEE Computer*, May 1996, pp. 78–86.
31. David L. Martin, Adam J. Cheyer, and Douglas B. Moran, "The Open Agent Architecture: A framework for building distributed software systems," *Applied Artificial Intelligence*, Vol. 13, No. 1–2, 1999, pp. 92–128.
32. Bertrand Meyer and Christine Mingins, "Component-Based Development: From Buzz to Spark," *IEEE Computer*, Vol. 32, No. 7, July 1999, pp. 35–37.
33. David P. Miller, Rajiv S. Desai, Erann Gat, Robert Ivlev, and John Loch, "Reactive Navigation through Rough Terrain: Experimental Results," *Proc. 10th National Conference on Artificial Intelligence*, AAAI Press, Menlo Park, CA, July 1992, pp. 823–828.
34. Robin Milner, "Elements of Interaction," *Communications of the ACM*, Vol. 36, No. 1, January 1993, pp. 78–89.
35. Allen Newell, "The knowledge level," *Artificial Intelligence*, Vol. 18, No. 1, 1982, pp. 87–127.
36. M. J. Pont and E. Moreale, "Towards a Practical Methodology for Agent-Oriented Software Engineering with C++ and Java," Leicester University Technical Report 96-33, December 1996.
37. Anand S. Rao and Michael P. Georgeff, "Modeling rational agents within a BDI-architecture," in *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, 1991, pp. 473–484.
38. Stuart J. Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, Englewood Cliffs, N.J., 1995.
39. Mary Shaw, "Outlook on Software System Design," *IEEE Computer*, Vol. 31, No. 1, January 1998, p. 32.
40. Yoav Shoham, "Agent-Oriented Programming," *Artificial Intelligence*, Vol. 60, No. 2, June 1993, pp. 51–92.
41. Charles Simonyi, "The Future is Intentional," *IEEE Computer*, Vol. 32, No. 5, May 1999, pp. 56–57.
42. Munindar P. Singh and Michael N. Huhns, "Social Abstractions for Information Agents," in *Intelligent Information Agents*, Matthias Klusch, ed., Kluwer Academic Publishers, Boston, MA, 1999.
43. Clement Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison Wesley Longman, 1998.

44. Milind Tambe, David V. Pynadath, and Nicolas Chauvat, "Building Dynamic Agent Organizations in Cyberspace," *IEEE Internet Computing*, Vol. 4, No. 2, March/April 2000.
45. Jose M. Vidal, Paul A. Buhler, and Michael N. Huhns, "Inside an Agent," *IEEE Internet Computing*, Vol. 5, No. 1, January/February 2001, pp. 86–90.
46. Jose M. Vidal and Edmund H. Durfee, "Learning Nested Agent Models in an Information Economy," *Journal of Experimental and Theoretical Artificial Intelligence (special issue on learning in distributed artificial intelligence systems)*, 1998.
47. Guijun Wang, Liz Ungar, and Dan Klawitter, "Component Assembly for OO Distributed Systems," *IEEE Computer*, Vol. 32, No. 7, July 1999, pp. 71–78.
48. Peter Wegner, "Why Interaction is More Powerful Than Algorithms," *Communications of the ACM*, Vol. 40, No. 5, May 1997, pp. 80–91.
49. Peter Wegner, "Interactive Software Technology," *CRC Handbook of Computer Science and Engineering*, May 1996, pp. 1–24.
50. Peter Wegner, "Interactive Foundations of Object-Based Programming," *IEEE Computer*, October 1995, pp. 70–72.
51. Gerhard Weiss, ed., *Multiagent Systems*, MIT Press, Cambridge, MA, 1999.
52. Darrell Woelk, Michael Huhns, and Christine Tomlinson, "Uncovering the Next Generation of Active Objects," *Object Magazine*, July–August 1995, pp. 33–40.
53. Michael J. Wooldridge, "Agents and Software Engineering," *Proceedings AIIA*, 1998.
54. Michael J. Wooldridge, "Agent-Based Software Engineering," *IEE Proceedings on Software Engineering*, Vol. 144, No. 1, February 1997, pp. 26–37.
55. Michael J. Wooldridge and Nicholas R. Jennings, "Software Engineering with Agents: Pitfalls and Pratfalls," *IEEE Internet Computing*, Vol. 3, No. 3, May/June 1999, pp. 20–27.
56. Michael J. Wooldridge, Nicholas R. Jennings, and David Kinny, "A Methodology for Agent-Oriented Analysis and Design," in Oren Etzioni, Jean-Pierre Muller, and Jeffrey Bradshaw, eds., *Agents'99: Proceedings of the third International Conference on Autonomous Agents*, Seattle, WA, May 1999.
57. Edward Yourdon, "Java, the Web, and Software Development," *IEEE Computer*, August 1996, pp. 25–30.