AGENT TEAMS: Building and Implementing Software

Michael N. Huhns • *University of South Carolina* • *huhns@sc.edu*

Since the very first issue of *IEEE Internet Computing*, this column has focused on the many ways in which agents can help users exploit the Web. Soon, agents will begin to play an even more important role, becoming the fundamental building blocks for general-purpose Internet-based software. The software may not display any explicitly agent-like characteristics, but it will exhibit the benefits of tolerance to errors, ease of maintenance, adaptability to change, and speed of construction that agents provide. Moreover, an agent-based approach to software development can lead to new types of software solutions that might not otherwise be obvious.

Conventional Software Development

Nearly everyone agrees that something is fundamentally wrong with current software development practice. Even as the infrastructure for computing and communications is increasing dramatically and the number of users working with a wide range of applications is growing, techniques for developing software systems remain woefully inadequate. Although we use phrases such as "computer age" to mean advanced or leading edge, the systems that give the modern age its name are constructed in a manner more resembling sorcery than science.

Software development has simply not kept pace with the rest of the computing industry. Whereas processor performance has been increasing 48 percent and network capacity 78 percent annually, software productivity has been growing only 4.6 percent annually. Moreover, we are stuck with a legacy of approximately 50 billion lines of Cobol, representing roughly 80 percent of all software written since 1960. It is unlikely that we can replace it anytime soon, even though maintaining it costs US$3 billion annually.

Programmers still produce approximately the same number of lines of tested and debugged code per day as they did in 1975, despite such "silver bullets" as structured programming, declarative specifications, object-oriented programming, formal methods, and visual languages. Actually, this shouldn't be surprising, considering that software systems are the most complicated artifacts people have ever attempted to construct. Paradoxically, they are (supposedly) guaranteed to work correctly only when all errors have been detected and removed, which their very complexity makes infeasible. Moreover, the effect of an error is unrelated to its size, so that a single misplaced character out of millions can render a system useless—or worse, harmful.

Agent-Based Software Development

What we need is not simply another variant of conventional software engineering but a radically new way of thinking about software and the purposes it serves. From this new way of thinking will arise new technical approaches for constructing software. An approach based on teams of active, cooperative, and persistent software components—that is, agents—shows special promise in enabling the rapid construction of robust and reusable software.

Most business software components are designed to be models of some real entity, such as an employee. But unlike the entities they represent, these components are passive. Why is this a problem? Let's say someone accidentally reduced the salary of an employee by 50 percent. A conventional software component would accept the change without protest, but a well-designed agent that is active and aware would object, just like a real employee.

Agents can do much more. Imagine what software development would be like if programmers were freed from debugging lines of code and instead spent their time selecting "volunteers" (agents from an active repository) and assembling them into a problem-solving team. These agents would cooperate and provide mutual assistance, compensating for each other's mistakes or limitations. The team would be open in that its membership would be dynamic, making it more suitable than conventional software for dealing with open information environments, such as the Internet.

Any system's behavior depends on its construction and its operating environment. When the system contains numerous components that interact with each other and with a complex

environment, behavior can be difficult to predict and control. Traditional software interfaces are rigid. Often the slightest error in a component's implementation can have far-reaching repercussions on the behavior of the entire system. A component's output may be erroneous because it malfunctions, its environment is out of its design range, or it receives erroneous input from another component. Traditional, rigid approaches for software or hardware fault tolerance use fixed means such as averaging or voting to correct errors.

By contrast, with an agent-based approach 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 programmers to design components to be more flexible toward their inputs and outputs. Moreover, in real-life situations, a component may be forced to release erroneous results because it lacks the time to await definite inputs and/or the resources to process them properly. The new approach can handle these situations naturally, whereas traditional approaches are incapable of even representing them.

The team approach presupposes that the components can enter into social commitments to collaborate and negotiate with others and, when necessary, reverse previous decisions on the basis of new information. They must be long-lived to detect errors that manifest later in the execution, and they must be 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. They can volunteer to take advantage of those opportunities, to form teams, negotiate solutions, and enact solutions in a persistent manner.

One risk with such systems is that their persistence may get them into "livelocks," where continuous interactions prevent progress. For example, two agents that are designed to acknowledge every message they receive might get stuck acknowledging each other's acknowledgments. Agents must be able to explore their way out of livelocks, possibly by continually reassessing whether or not they are getting nearer their goals or objectives.

Forming a Circle

When asked to form a circle, children can comply regardless of their number, sizes, or ages; they do not need explicit directions as to who should stand where. 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, possessing basic knowledge of the problem domain. Each element is not, however, required to be perfect.

Contrast this with a conventional object-oriented 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 would use trigonometry to 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, redefining the shape or size of an item would require the control module to be rewritten.

A Team of Agents

The approach envisioned here would consist of multiple, redundant, agent-based components interacting over a network. For the above example, each would behave like a child in understanding what a circle is and what its role in a circle would be. The appropriate analogy is that of a large, robust, natural system. Programming and activating a team of agents requires that the following matters be resolved: who (role) will do what (subtask), when (coordination), how (capabilities), where (resources or location), and why (team plan and external requirements). In addition, there are the aggregate matters of how many (agents per role), how much (extent of resources), and how long (performance requirements). The main steps are agent creation (compilation), team configuration (linkage), and team activation (execution).

The approach presupposes either a repository of agents or the availability of "agent factories," together with protocols for discovery, negotiation, and software configuration. 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. Table 1 compares the major features of two existing software paradigms with those promised by the new paradigm.

A Leaderless Orchestra

But wouldn't a self-assembled team of agents behave just like an orchestra without a leader? Yes, but maybe this isn't so bad. One such orchestra, Orpheus, performed recently at Carnegie Hall in New York City, and with remarkable success.[1] Orpheus is the ultimate flat, nonhierarchical organization, and its experiences with being leaderless are of great interest to hundreds of American companies that are trying to become "flatter" by removing layers of middle management. Flatter organizations, according to the management theory promoting them, have lower overhead costs and can react more quickly to technological change. Wal-Mart, Cisco Systems, and General Electric have been operating according to this theory.

Orpheus recognizes that the key techniques for getting people to do things when you lack authority over them are negotiation and persuasion. These are exactly the abilities that agents in teams need.

Customized Components

Although the pragmatics and methodology of how to go about developing software based on agents are still being worked out, there has already been significant progress.[2] We are beginning to see the rise of a software-component industry that will distribute on-demand components with func-

Table 1. Major features of the procedural, object-oriented, and team-oriented software paradigms.

	Procedural	Object-oriented	Team-oriented
Abstraction	Data type	Class	Society
Building block	Instance, data	Object	Agent
Computation model	Procedure / call	Method / message	Perceive / reason / act
Design paradigm	Tree of procedures	Method invocations	Cooperative interaction
Architecture	Functional decomposition	Inheritance and polymorphism	Managers, assistants, and peers
Modes of behavior	Program	Design and use	Enable and enact
Terminology	Implement	Engineer	Activate

tionality customized to users' needs. Rather than specifying how computations should be done, programmers will specify requirements, possibly in terms of social commitments and intentions. The components might negotiate, make commitments to collaborate, and reverse previous decisions about their results.

These capabilities will let us achieve the two major goals of software engineering—robustness and increased productivity—that are not being addressed by the current state of software practice. Early versions of tools for building agent teams are now operating in research laboratories and will soon be available for implementing the software systems of this millennium.

REFERENCES

1. D. Leonhardt, "Soothing Savage Structures: A Leaderless Orchestra Offers Lessons for Business," *The New York Times*, Nov. 10, 1999, pp. 13-14.
2. M.J. Wooldridge and N.R. Jennings, "Software Engineering with Agents: Pitfalls and Pratfalls," *IEEE Internet Computing*, May-June 1999, pp. 20-27.

Michael N. Huhns is a professor of electrical and computer engineering at the University of South Carolina, where he also directs the Center for Information Technology.