Inside an Agent

José M. Vidal • *University of South Carolina* • *vidal@sc.edu*
Paul A. Buhler • *College of Charleston* • *pbuhler@cs.cofc.edu*
Michael N. Huhns • *University of South Carolina* • *huhns@sc.edu*

Over the past four years, this column has focused on the uses and behaviors of Internet agents but ignored their implementation and internal appearance. Multiagent-system platforms[1] aid in creating agent-based systems, but to use them effectively we must understand an agent's architecture.

When we discuss agent-based-system construction with software developers or ask students to implement common agent architectures using object-oriented techniques, we find that it is not trivial for them to create an elegant system design from the standard presentation of these architectures in textbooks or research papers.

To better communicate our interpretation of popular agent architectures, we draw UML (Unified Modeling Language) diagrams[2] to guide an implementer's design. However, before we describe these diagrams, we need to review some basic features of agents. Consider the architecture in Figure 1, showing a simple agent interacting with an environment.[3] The agent senses its environment, uses what it senses to choose an action, and then performs the action through its effectors. Sensory input can include received messages, and action can be the sending of messages.

"This is a Space for a small quote"

To construct an agent, we need a more detailed understanding of how it functions. In particular, if we are to build one using conventional object-oriented analysis and design techniques, we should know in what ways an agent is more than just a simple object. Agent features relevant to implementation are unique identity, proactivity, persistence, autonomy, and sociability.[4]

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 similar to that possessed by an object in a derivation of the Java thread class. Here is simple pseudocode for a typical event loop, where events result from sensing the environment:

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

This is an infinite loop, which also provides agents with *persistence*. Ephemeral agents would find it difficult to converse, making them, by necessity, asocial. Additionally, persistence makes it worthwhile for agents to learn about and model each other. To benefit from such modeling, they must be able to distinguish one agent from another, hence the need for unique identities.

Agent *autonomy* is akin to human free will and enables an agent to choose its own actions. 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 doesn't intend. 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.

Enabling an agent to converse with other agents achieves *sociability*. The conversations, normally conducted by sending and receiving messages, provide opportunities for agents to coordinate their activities and cooperate, if so inclined. We can achieve sociability by generalizing the input class of objects an agent might perceive, as shown in Figure 2. Events serving as input are simply reminders the agent sets for itself. For example, an

 1089-7801/00/$10.00

agent wanting to wait five minutes for a reply would set an event to fire after five minutes. If the reply arrives before the event, the agent can disable the event. If it receives the event, then it knows it did not receive the reply in time and can proceed accordingly.

UML Agent Descriptions

The UML diagrams in Figure 3 and Figure 4 should help anyone interested in understanding or participating in software agent development (also called agent-based software engineering). These diagrams don't address every functional aspect of the architecture. Instead they provide a general framework for implementing traditional agent architectures[4] using an object-oriented language. We've had good experiences using them, and we encourage readers to contact us if they have a better way to implement these architectures.

Reactive Agents

A reactive agent is the simplest kind to build, since it doesn't maintain information about the state of its environment but simply reacts to current perceptions. Our design for such an agent, shown in Figure 3, is fairly intuitive, encapsulating a collection of behaviors, sometimes known as plans, and the means for selecting an appropriate one. A collection of objects, in the object-oriented sense, lets a developer add and remove behaviors without having to modify the action selection code, since an iterator[5] can be used to traverse the list of behaviors. Each behavior fires when it `matches` the environment, and each can `inhibit` other behaviors. Our action-selection loop is not as efficient as it could be, since `getAction` operates in $O(n)$ time (where n is the number of behaviors). A better implementation could lower the computation time to $O(\log n)$ using decision trees, or $O(1)$ using hardware or parallel processing. The user is responsible for ensuring that at least one behavior will match for every environment. This can be achieved by defining a default behavior that matches all inputs but is inhibited by all other behaviors that match.

BDI Agents

A belief-desire-intention (BDI) architecture includes and uses an explicit representation for an agent's beliefs, desires, and intentions. The BDI implementations that we know of—Procedural Reasoning System (PRS), University of Michigan PRS, and JAM—all define a new programming language and implement an interpreter for it. The advantage of this approach is that the interpreter can stop the program at any time, save state, and execute some other plan, or intention, if it needs to. The BDI architecture shown in Figure 4 doesn't do this. Instead it uses a voluntary multitasking method whereby the environment thread constantly checks to make sure the current intention is applicable. If it finds that it isn't, it will tell the intention to stop itself, which the intention does by calling `stopCurrentPlan()`. This method in turn will call `stopExecuting()`. Thus the plan is responsible for stopping itself and cleaning up. By giving each plan this capability, we eliminate the possibility of a

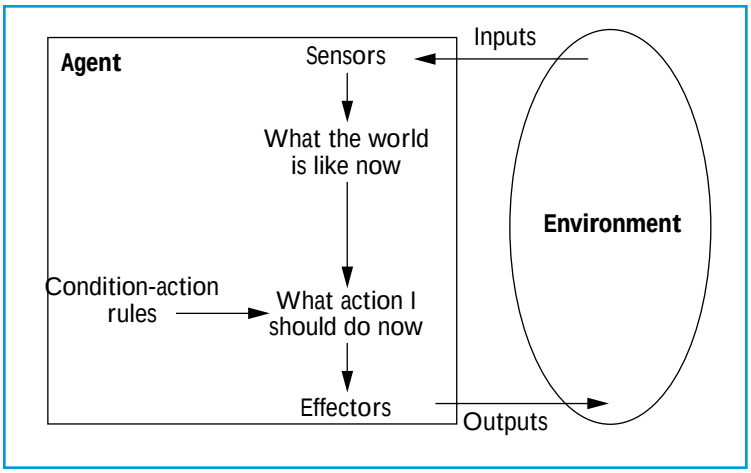

Figure 1. Simple agent-environment interaction.

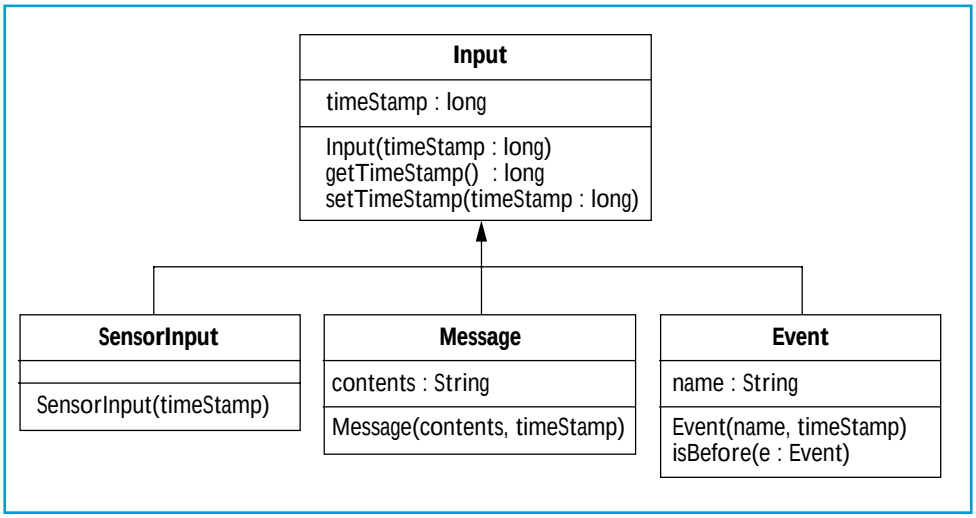

Figure 2. An agent's input can be a piece of sensory information, a message from another agent, or an event defined by the agent.

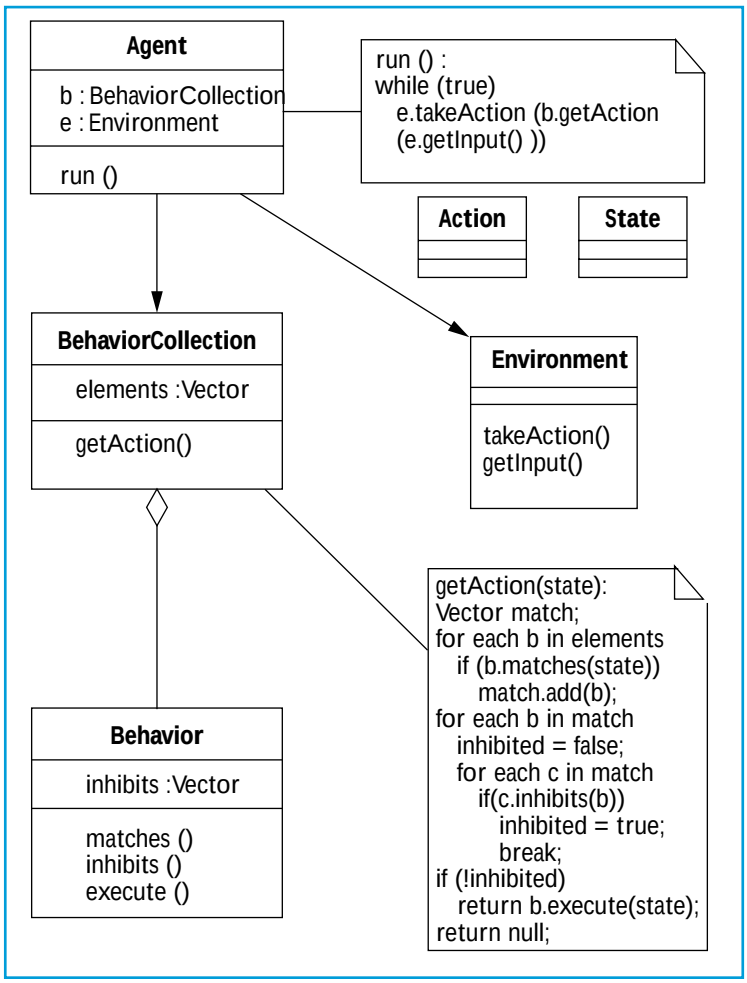

Figure 3. Diagram of a simple reactive agent.

deadlock resulting from the plan's having some resource reserved when it was stopped. The pseudocode in Figure 5 illustrates the two main loops, one for each thread, of our BDI architecture.

"This is a Space for a small quote"

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

Finally, the environment thread's sleep time can be modified, depending on the system's real-time requirements. If we don't need the agent to change plans rapidly when the environment changes, the thread can sleep longer. Otherwise, a short sleep will make the agent check the environment more frequently, using more computation. A more efficient call-back mechanism could easily replace the current run method if the agent's input mechanism supported it.

Behaviors and Activity Management

Most popular agent architectures, including the two we diagrammed, 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 multiagent systems, we can also distinguish between physical behaviors that generate actions, and conversations between agents. 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.

This general activity manager design lends itself to the implementation of many popular agent architectures while maintaining the proper encapsulation and decomposability required in good object-oriented programming. Specifically, activity is an abstract class that defines the interface to be implemented by all behaviors and conversations. The behavior class can implement any helper functions needed in the particular domain (for example, subroutines for triangulating the agent's position). The conversation class can implement a finite-state machine for use by the particular conversations. For example, by simply filling in the appropriate states and adding functions to handle the transitions, an agent can define a contracting protocol as a class that inherits from conversation. Details of how this is done depend on how the conversation class implements a finite-state machine, which varies depending on the system's real-time requirements.

Defining each activity as its own independent object and implementing a separate activity manager has several advantages. The most important is the separation between domain and control knowledge, a feature first popularized by blackboard systems. The activities will embody all the knowledge about the particular domain the agent inhabits, while the activity manager embodies knowledge about the deadlines and other scheduling constraints the agent faces. By implementing each activity as a separate class, we compel the programmer to separate the agent's abilities into

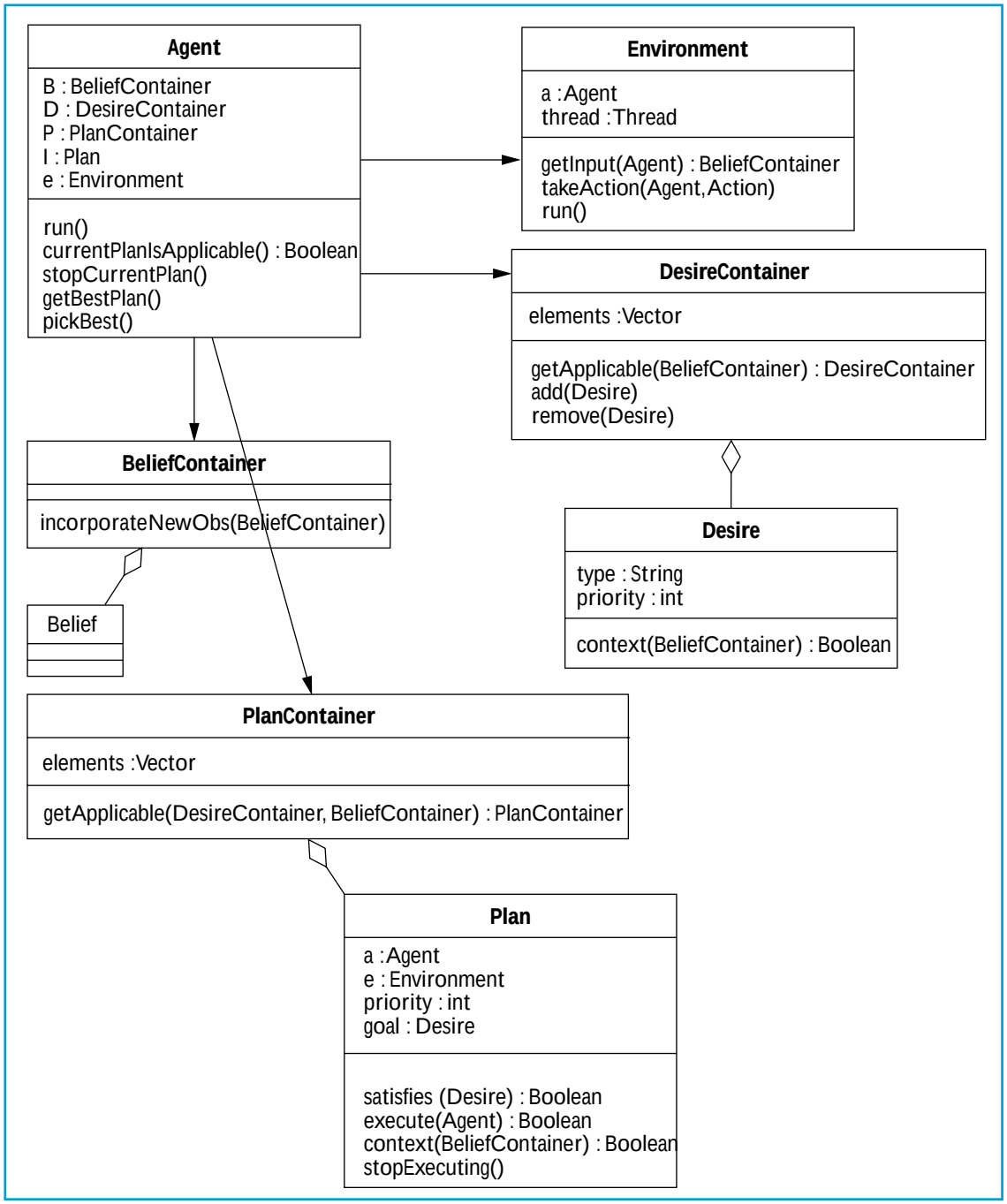

Figure 4. Diagram of a belief-desire-intention architecture.

encapsulated objects that other activities can then reuse. The activity hierarchy forces all activities to implement a minimal interface, which also facilitates reuse. Finally, placing the activities within the hierarchy provides many opportunities for reuse through inheritance. For example, the conversation class can implement a general lost-message error-handling procedure that all the conversations can use.

Architectural Support

Figures 3 and 4 provide general guidelines for implementing agent architectures using an object-oriented language. As agents become more complex, you will likely have to expand upon our techniques. We believe these guidelines are general enough that it won't be necessary to rewrite the entire agent from scratch when adding new functionality.

Of course, a complete agent-based system

```
Agent::run() {
Environment e;
e.run(); //start environment in its own thread

while (true) {
  I = getBestPlan();
  if (I.execute()) // true if goal was achieved
    D.remove(I.goal);
}

Environment::run(){
while (true) {
  a.B.incorporateNewObservations(getInput(w));
  if (! a.currentPlanIsApplicable())
    a.stopCurrentPlan();
  sleep(someShortTime);
}
```

Figure 5. Pseudocode for voluntary multitasking in the BDI architecture.

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,[6] Bluetooth,[7] and FIPA's (the Foundation of Intelligent Physical Agents[8]) emerging standards.

Additional information about agent tools and architectures is available at http://www.multiagent.com/, a site maintained by José Vidal.

References

1. P.M. Ricordel and Y. Demazeau, "From Analysis to Deployment: A Multi-Agent Platform Survey," *Proc. Workshop on Engineering Societies in the Agents' World*, Springer-Verlag, Berlin, 2000; http://lia.deis.unibo.it/confs/ESAW00/.
2. M. Fowler, *UML Distilled, 2nd Edition: A Brief Guide to the Standard Object Modeling Language*, Addison-Wesley, Reading, Mass., 2000.
3. S.J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, Englewood Cliffs, N.J., 1995.
4. Weiss, Gerhard, ed., *Multiagent Systems*, MIT Press, Cambridge, Mass., 1999.
5. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Mass., 1995.
6. Further information available at http://www.sun.com/jini/.
7. Further information available at http://www.bluetooth.com/.
8. Further information available at http://www.fipa.org/.

José M. Vidal is an assistant professor of computer science and engineering at the University of South Carolina, where he is conducting research in multiagent systems and distributed software systems.

Paul A. Buhler is an instructor of computer science at the College of Charleston and a PhD candidate at the University of South Carolina, where his primary research interest is agent-oriented software engineering.

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