# JACK Intelligent Agents - Components for Intelligent Agents in Java*

Paolo Busetta, Ralph Rönnquist, Andrew Hodgson and Andrew Lucas
Agent Oriented Software Pty. Ltd., Melbourne, Australia.
{paolo|ralph|ash|cal}@agent-software.com

December, 1999

**Abstract**

Intelligent Agents are being used for modelling simple rational behaviours in a wide range of distributed applications. In particular, multi-agent architectures based on the Belief-Desire-Intention (BDI) model have been used successfully in situations where modelling of human reasoning and team behaviour are needed, such as simulating tactical decision-making in air operations and command and control structures. Other applications include intelligent decision support, telephone call centres, and air traffic management.

The JACK Intelligent Agents$^{TM}$ framework by Agent Oriented Software brings the concept of intelligent agents into the mainstream of commercial software engineering and Java. JACK Intelligent Agents is a third-generation agent framework, designed as a set of light-weight components with high performance and strong data typing.

We present the design approach and major technical characteristics of JACK Intelligent Agents. An outline of a typical development process involving the framework is given. Also, we discuss the benefits of the component-based approach, for both the software engineer developing sophisticated distributed applications, and for the researcher exploring agent models and architectures.

**Keywords** agents, multi-agent systems, BDI, Java, components

## 1 Introduction

Intelligent Agents are being used for modelling simple rational behaviours in a wide range of distributed applications. Intelligent agents have received various, possibly contradictory, definitions; by general consensus, they must show some degree of autonomy, social ability, and combine pro-active and reactive behaviour [WJ95]. One of the better known and most successful architectures for agents is the Belief-Desire-Intention (BDI) architecture, which has seen a number of academic and industrial applications.

Agent Oriented Software Pty. Ltd. (AOS), based in Melbourne, Australia, has built JACK Intelligent Agents$^{TM}$, a framework in Java for multi-agent system development. The company's aim is to provide a platform for commercial, industrial and research applications. To this end, its framework supplies a high performance,

[1] This article first appeared in the AgentLink Newsletter. It has since been edited for clarity and updated to reflect the current JACK Intelligent Agents distribution.

light-weight implementation of the BDI architecture, and can be easily extended to support different agent models or specific application requirements.

This paper is organised as follows. Section 2 introduces JACK Intelligent Agents, presenting the approach taken by AOS to its design and outlining its major engineering characteristics. The BDI model is discussed briefly in Section 3. Section 4 gives an outline of how to build an application with JACK Intelligent Agents. Finally, in Section 5 we discuss how the use of this framework can be beneficial to both engineers and researchers.

For brevity we will refer to JACK Intelligent Agents simply as 'JACK'.

# 2 JACK Intelligent Agents

In this section, we present JACK by first highlighting the goals set by its designers, then we provide an overview of the engineering characteristics of the framework.

## 2.1 Approach

Major design goals for JACK were:

- to provide developers with a robust, stable, light-weight product;
- to satisfy a variety of practical application needs;
- to ease technology transfer from research to industry; and
- to enable further applied research.

Whilst applications can be built from the ground up adopting an agent oriented methodology and an appropriate framework, most organisations already possess and depend upon large legacy software systems. Thus, JACK agents have been designed mainly for use as components of larger environments. Consequently, an agent must coexist and be visible as simply another object by non-agent software. Conversely, a JACK programmer must be allowed to easily access any other component of a system. Type safeness when accessing data, reliability and support for a proper engineering process are then key requirements in this kind of environment.

For similar reasons, JACK agents are not bound to any specific agent communications language. Nothing prevents the adoption of a high-level symbolic protocol such as KQML, possibly by integrating software already existing in the public domain. However, JACK has been geared towards industrial object-oriented middleware (such as CORBA) and message passing infrastructures (for instance, PVM or DIS in simulation environments). In addition, JACK provides a native light-weight communications infrastructure for situations where high performance is required.

JACK itself has been designed for extension by properly trained engineers familiar with agent concepts and with a sound understanding of concurrent object-oriented programming.

## 2.2 Overview of the framework

From an engineering perspective, JACK consists of architecture-independent facilities, plus a set of plug-in components that address the requirements of specific agent architectures. An example of such a plug-in is the default BDI reasoning model supplied with JACK.

To an application programmer, JACK currently consists of three main extensions to Java. The first is a set of syntactical additions to its host language, which can be divided as follows:

- a small number of keywords for the identification of the main components of an agent (such as *agent*, *plan* and *event*);
- a set of statements for the declaration of attributes and other characteristics of the components (for instance, the information contained in beliefs or carried by events). All attributes are strongly typed;
- a set of statements for the definition of static relationships (for instance, which plans can be adopted to react to a certain event);
- a set of statements for the manipulation of an agent's state (for instance, additions of new goals or sub-goals to be achieved, changes of beliefs, interaction with other agents).

Furthermore, the programmer can use Java statements within the components of an agent.

For the convenience of programmers, in particular those with a background in AI, JACK also supports *logical variables* and *cursors*. These are particularly helpful when querying the state of an agent's beliefs. Their semantics are mid-way between logic programming languages (with the addition of type checking Java style) and embedded SQL.

The second extension to Java is a compiler that converts the syntactic additions described above into pure Java classes and statements that can be loaded with, and be called by, other Java code. The compiler also partially transforms the code of plans in order to obtain the correct semantics of the BDI architecture.

Finally, a set of classes (called the *kernel*) provides the required run-time support to the generated code. This includes:

- the automatic management of concurrency among tasks being pursued in parallel (*intentions* in the BDI terminology);
- the default behaviour of the agent in reaction to events, failure of actions and tasks, and so on; and
- a native light-weight, high performance communications infrastructure for multi-agent applications.

Importantly, the JACK kernel supports multiple agents within a single process. This is particularly convenient for saving system resources. For instance, agents that perform only short computations or share most of their code or data can be grouped together.

A JACK programmer can extend or change the architecture of an agent by providing new plug-ins. In most cases, this simply means overriding the default Java methods provided by the kernel or supplying new classes for run-time support. However, it is possible to add further syntactic extensions to be handled by the JACK compiler.

Similarly, a different communications infrastructure can be supplied by overriding the appropriate run-time methods.

Future versions of JACK will extend the base BDI model with new plug-ins and will add a number of development and monitoring tools.

# 3 Belief-Desire-Intention Agents

The Belief-Desire-Intention (BDI) agent model supported by JACK has its roots in philosophy and cognitive science, and in particular in the work of Bratman on rational agents [Bra87]. A rational agent has bounded resources, limited understanding

and incomplete knowledge of what happens in the environment in which it lives. Such an agent has *beliefs* about the world and *desires* to satisfy, driving it to form *intentions* to act. An intention is a commitment to perform a plan. In general, a plan is only partially specified at the time of its formulation since the exact steps to be performed may depend on the state of the environment when they are eventually executed. The activity of a rational agent consists of performing the actions that it intended to execute without any further reasoning, until it is forced into a revision of its own intentions by changes to its beliefs or desires. Beliefs, desires and intentions are called the mental attitudes (or mental states) of an agent.

Note that BDI agents depart from purely deductive systems and other traditional AI models because of the concept of intentionality, which significantly reduces the extent of deliberation required. BDI has been shown to be well suited to modelling certain types of behaviour, such as the application of standard operational procedures by trained staff. It has been successfully adopted in fields as diverse as simulation of military tactics, application of business rules in workflows, and diagnostics in telecommunications networks.

Based on previous research and practical applications, Rao and Georgeff [RG92] have described a computational model for a generic software system implementing a BDI agent. Such a system is an example of event-driven programming. In reaction to an event, for instance a change in the environment or its own beliefs, a BDI agent adopts an appropriate plan as one of its intentions. Plans are precompiled procedures that each have a set of conditions that are used to determine their applicability. The process of adopting a plan as one of the agent's intentions may require a selection from a set of candidate plans.

The agent executes the steps of the plans that it has adopted as intentions until further deliberation is required; this may happen because of new events or the failure or successful conclusion of existing intentions.

A step in a plan can consist of adding a goal (that is, a desire to achieve a certain objective) to the agent itself, changing its beliefs, interacting with other agents, and any other atomic action on the agent's own state or the external world.

The abstract BDI architecture has been implemented in a number of systems. Of these, two are of particular relevance to JACK since they represent its immediate predecessors. The first generation is typified by the Procedural Reasoning System (PRS) [GI89], developed by SRI International in the mid '80s. dMARS [dKLW98], built in the mid '90s by the Australian Artificial Intelligence Institute in Melbourne, Australia, is a second generation system. dMARS has been used as the development platform for a number of technology demonstrator applications, including simulations of tactical decision-making in air operations and air traffic management.

# 4 Application development with JACK

In an ideal setting, a developer building an application with JACK should start by identifying the distributed components of the system. The design of a multi agent application requires a sound understanding of distributed system development and distributed AI principles that we cannot discuss here. However, observe that in practical situations the decision as to how to distribute functionality may be dictated by a number of external constraints, such as the existence of legacy systems or a specific communications infrastructure.

For this discussion, let us assume that the functionality that has to be provided by an agent has been identified and that the BDI model has been chosen. At this stage, two main activities that have to be performed are (not necessarily in the order given below):

- identifying the elementary classes (that is, abstract data types and the operation allowed on them) that are required to manipulate the resources used by the agent. These could be external (relational databases, the Internet, the arms of a robot, a GUI and so on) as well as internal (for instance, specific mathematic data structures to represent financial or spatial information);
- identifying those elements that constitute the mental states of the agent. This involves finding:
  - which external events drive the agent (including messages from other agents);
  - which goals the agent can set for itself;
  - which beliefs influence the adoption of plans; and finally
  - the procedures (that is, the plans in BDI terms) required to accomplish tasks, achieve goals and react to events in the various possible contexts.

The implementation of an agent is then a combination of normal Java code for the elementary classes and extended Java, described in Section 2.2, for the agent-specific components. The plans of a JACK agent are, in general, sequences of operations on elementary objects, manipulations of the mental states (e.g., submitting sub-goals or changing beliefs) and interactions with other agents.

Thus a JACK plan could be represented as procedural logic in a flow diagram, state diagram, coordination diagram or other similar notation in an object-oriented methodology such as UML. This is to say that JACK could be used as an extended object-oriented framework supporting event driven and procedural logic in a concurrent execution environment, with the additional benefits of sensitivity to the context and sophisticated management of failure provided by the BDI architecture.

## 4.1 An example

To give a sample of the code of a JACK agent, we have extracted an example from one of the tutorials that are part of the JACK documentation set. The purpose is not to show agent programming but to illustrate how JACK code looks as a straightforward Java extension. This section can be passed over by those not familiar with Java.

This example has agents that "ping" each other, that is, exchange empty messages. The message being exchanged is represented as an event that originates with one agent and is received by another:

```
event PingEvent extends MessageEvent {
   int value;
   #posted as
   ping(int value)
   {
       this.value = value;
   }
}
```

Note the `event` keyword, in place of `class` in Java. The event is also declared as a `MessageEvent`, which means that it can be sent to another agent; this drives JACK to bring in all the required communications support. The `#posted as` statement declares how the event is generated (in this case, by invoking a Java method, `ping()` with one integer parameter).

The following plan handles the notification of the event above and replies to its sender by "bouncing" the event back. This simple example is not sensitive to the

context, i.e., there is no restriction on the state of the beliefs of the agent for its applicability.

```
plan BouncingPlan extends Plan {
   #handles event PingEvent ev;
   #sends event PingEvent pev;

   body()
   {
      @send ( ev.from, pev.ping(ev.value + 1) );
      // Reply to the sender of the event
   }
}
```

A trivial "ping agent" is defined below. It has a single plan and handles a single event. When its method `ping (String other)` is called, it notifies the `PingEvent` with value 1 to the agent `other`. If the latter is another `PingAgent`, then `BouncingPlan` above is invoked, a `PingEvent` with value 2 is sent back, and so on to infinity.

```
agent PingAgent extends Agent {
   #handles event PingEvent;
   #uses plan PingPlan;
   #posts event PingEvent pev;

   void ping (String other)
   {
      send(other, pev.ping(1));
   }
}
```

The application can instantiate as many `PingAgent` agents as it desires. The names of the agents and their network addresses are determined by the communications mechanism in use; as stated before, JACK provides a high performance messaging system with a simple naming scheme.

# 5 Benefits of JACK

The approach taken by JACK has a number of advantages in comparison with both other agent frameworks coming from the artificial intelligence world and standard object-oriented architectures.

The adoption of Java guarantees a widely available, well-supported execution environment. In addition to the promises of the language (summarised by the well known slogan "compile once, run everywhere" by Sun Microsystems), we expect that an increasing number of software components, tools and trained engineers will be available in the next few years.

To the AI researcher, the adoption of an imperative, relatively low-level language such as Java means losing some of the expressive power offered by frameworks based on logic or functional languages. However this is compensated for, not only by the universal availability mentioned above, but also by the modular approach of JACK. As stated in the previous sections, most components of the framework can be tuned and tailored. This makes JACK particularly suited to experimentation with new agent architectures in order to try out new functionality (new mental attitudes,

different semantics, additional types of knowledge bases, and so on) or to study performance characteristics in specific contexts.

Moreover, when compared with frameworks based on traditional AI languages, JACK has distinctive advantages due to a proper utilisation of the intrinsic characteristics of Java. The most important is strong typing, which reduces the chances of programming errors introduced by simple mistyping. It also provides a very basic version control by making sure that interfaces are compatible at run-time. Next is performance, which makes the execution speed of agent code written in JACK comparable to a direct implementation in C or C++.

For the engineer developing a sophisticated distributed application, JACK has several interesting aspects, for instance:

- an efficient way to express high level procedural logic within an object-oriented environment.

  This also helps in rapid application development by allowing a clear distinction between abstract data types and their operations, and application-specific behaviour requiring fine-tuning or evolution when the system is already operational. While the former should be based on high performance, well tested, highly reusable and ultimately expensive code, the latter is better expressed as plans which can be easily modified;

- the context sensitivity and sophisticated semantics of mental attitudes of the BDI architecture.

  These characteristics enable some levels of adaptability to changing conditions;

- ease of integration with legacy systems.

  This enables, among other things, an incremental approach to distributed system development.

When compared with frameworks originating from research environments, JACK has the clear advantages of being light-weight, industrial strength and accessible to a large community of engineers trained in object-oriented programming.

# 6 Conclusions

JACK Intelligent Agents is a multi-agent framework that extends the Java language. The current version supports the BDI model, and its modularity enables extensions and different models to be easily supported.

JACK is an industry-strength product, providing a framework that takes a solution founded in artificial intelligence research into practical use. Compared with its "predecessors", e.g., the PRS and dMARS systems mentioned above and other similar agent frameworks available in the academic world, JACK is not a 'pure' AI system. Instead, it constitutes a successful marriage between the vision of agent research and the needs of software engineering, bringing the power of agent technology to and enriching the host language, Java.

We are confident that JACK will provide benefits both to the software engineer developing distributed systems and to the academic researcher.

# 7 Acknowledgments

The authors would like to thank Prof. Ramamohanarao Kotagiri of the University of Melbourne for his valuable suggestions.

## References


[Bra87] Michael Bratman. *Intention, plans, and practical reason.* Harvard University Press, Cambridge, MA (USA), 1987.

[dKLW98] Mark d'Inverno, David Kinny, Michael Luck, and Michael Wooldrige. A formal specification of dMARS. In Munindar Singh, Michael Wooldrige, and Anand Rao, editors, *Intelligent Agents IV: Agent Theories, Architectures, and Languages.* Springer-Verlag, 1998.

[GI89] Michael Georgeff and F. Ingrand. Decision-making in an embedded reasoning system. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1989.

[RG92] Anand Rao and Michael Georgeff. An abstract architecture for rational agents. In C. Rich, W. Swartout, and B. Nebel, editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR'92).* Morgan Kaufmann Publishers, 1992.

[WJ95] Michael Wooldridge and Nicholas Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(12):115–152, 1995.