# Aria Language, Towards Agent Orientation Paradigm

**Mohsen Lesani**
**Computer Engineering Department**
**Sharif University of Technology**
**Tehran, Iran**
**Mohsen_lesani@ce.sharif.edu**

**Niloufar Montazeri**
**Computer Engineering Department**
**Sharif University of Technology**
**Tehran, Iran**
**montazeri@ce.sharif.edu**

## Abstract

As building large-scale software systems is complex, several software engineering paradigms have been devised. Agent oriented paradigm is one of the most predominant contributions to the field of software engineering and has the potential to significantly improve current practice of the field. The paradigm should be elaborated both practically and conceptually. Most existing agent oriented frameworks do not offer agent definition languages but propose to define agents with the help of agent libraries in existing object oriented languages. Few frameworks that propose languages lack conceptual principles for agent orientation. The contribution of this paper is twofold. Firstly, an agent oriented language called Aria and its compiler are proposed. Aria language is a superset of Java language and the compiler compiles a program in Aria to an equivalent program in Java. These enable Aria to fully integrate with and preserve all the existing knowledge and code in Java. Secondly, the three well-known object oriented principles of abstraction, inheritance and polymorphism are redefined for agent orientation. As chat room is a distributed application, it is selected as a sample case, designed and developed successfully in Aria. In addition, agent MVC architecture is offered as the second case.

## 1. Introduction

Programming languages seem to be the heart of computer science. Programming is to define a problem for a machine to get the solution. The difficulty is that the underlying hardware supports the machine to understand a primitive language while human speaks the natural language. Programming languages are devised to fill this gap. The distance from the

programming language to the machine language is the way that the compilers bypass for human. It first became possible to make practical use of high level programming languages in the 1950s. Since then, successive innovations in programming languages such as procedural, functional and object oriented approaches all were attempts to make the machine understand human more. Programming languages advance to help the programmer to code a problem definition as it really is in the problem domain or as people view and abstract it, not to code it how the machine simply understands. In other words, we attempt to make the programming language as close as possible to the natural language or human mind abstractions. The closer the code can be written to human mind abstractions and parlance, the more easily it can be written, understood, maintained and reused.

Object orientation was a great step toward modeling systems close to how people conceive them. Object orientation breaks a problem domain from the data point of view. It can well design systems that are data oriented, but process oriented systems that require multiple entities to execute concurrently, especially distributed systems, are hard to be neatly designed in object oriented paradigm. The problem is that objects are passive entities and the thread of execution is not a primary concept in object orientation. On the other hand, increasing multi core architectures provide hardware support for concurrency. Multi threaded applications should be developed in order to make use of computational power of such architectures [16]. These facts reveal the need for a more powerful modeling than object orientation.

Systems are most naturally viewed as a team of cooperating subsystems. A software system is best designed as multiple interacting proactive elements, called agents. Agents that were first proposed in AI community [12] represent high level abstractions of active entities in a software system [9][17]. The behavior of a multi-agent software system is the emergence of cooperation of the agents. Agents are desired to be capable of autonomously making intelligent decisions to perform actions in furtherance of their responsibilities. This grounds the need for a language paradigm that supports multi-agent software design. Languages are required for agent specification and compilers are necessary to compile the agent specifications. Platforms and tools are also required to support multi-agent software designers and developers.

The term “agent-oriented programming language” was coined by Shoham from Stanford University in the late 1980s. In an influential paper, he proposed Agent-0 language [13] and presented the concept of agent-oriented programming. Agent-0 is a fundamentally logic based declarative language. Some of the ideas that Shoham offered in his language especially the BDI (Belief Desire Intention) [14] agent model became central to some languages emerged later. Some of these languages such as 3APL[1] [6][7] and Jason[2] [3][4] have close similarities to

[1] http://www.cs.uu.nl/3apl/
[2] http://jason.sourceforge.net

Shoham's language. By contrast, other languages such as Jade[3] [1][2] (that is more precisely a library not a language), Jadex[4] [10][11] and Jack[5] [5][8] share some similar ideas but are fundamentally not logic based and are the result of object oriented practitioners' shift towards agent orientation.

One important criterion of new languages is their backward compatibility. An agent oriented language is expected to support successful features and grammars of previous popular languages (such as object oriented languages) while granting new features for defining agent oriented concepts. In other words, the developer's attitude to shift to the new language and hence the success of language is highly dependent on preserving the developer's existing knowledge and code. Both Aagent-0 and 3APL are logic based languages. The employed logic syntax is not object oriented and linking to available object oriented libraries is impossible or hard. Hence the two languages have gained interest not in practical but research projects. On the other hand, Jade provides programming agents in Java but is not a standalone agent oriented language. It is a Java library for agent modeling. So the user should try to map agent oriented concepts into the object oriented definitions while at best, the user is expected to be able to program in a language that maintains exact features for definition of agent oriented concepts. Jadex can integrate with existing Java libraries but agents should be specified in an XML file that is hardly readable. Jack is an agent oriented language and integrates with Java but the syntax can get more developed i.e. all the constructs should currently begin with a dummy sharp sign in Jack agent specifications. More importantly, object orientation is known to have three principles of encapsulation, inheritance and polymorphism. The publications from neither of the previously mentioned languages including Jack have theoretical discussions of the new paradigm. It is expected to preserve at least the principles of object orientation but even agent inheritance is not present in Jack.

While there are few agent oriented languages available, much way is remained to a mature agent oriented language. This research offers an agent oriented language called Aria[6] that besides providing features to specify agent concepts, also supports full integration with Java. The three object orientation principles are conceptually overridden in Aria. The language is employed for developing chat room application and MVC architecture.

In the remainder of this paper, an explanation of agent orientation precedes the description of Aria language. The three principles are explained along the language features. The paper proceeds with some remarks about the compiler followed by the implemented cases. Finally the conclusion and future works conclude the paper.

---

[3] http://jade.tilab.com

[4] http://vsis-www.informatik.uni-hamburg.de/projects/jadex/

[5] http://www.agent-software.com/

[6] Copyright Lesani 2007

## 2. Aria Agent Orientation

In a sheer agent oriented approach, a system is designed as a team of autonomous agents interacting and cooperating to achieve the system goals. In contrast to an object that is a passive entity, an agent is modeled as a proactive social entity.

An object does not change state or make any other object to change state unless it is told through methods to do so. On the contrary, an agent has different concurrent behaviors in action without the need for any external authority. An agent is alive and its behaviors are autonomously effective regardless of other agents. Agents are social as they interact by sending messages to and receiving messages from each other.

Aria agent oriented language proposes language built-in support for specification of autonomous agents. It conceptually overrides and extends the three well known object orientation principles.

- Abstraction: Everything is an agent or an object. Agents interact by sending messages to each other.
- Inheritance: An agent can inherit message servicing and behaviors from a parent agent.
- Polymorphism: An agent can override its parent definitions for message servicing and behaviors and the overriding definitions are effective even when the agent is referenced as of its parent type.

The principles are explained in the following sections.

## 3. Aria Language

### Abstraction

An agent is specified in Aria as syntax in Code Snippet 1.

```
public agent AgentType
        specializes parentAgentType
        services ServiceType1, ServiceType2, ...
{

        atBirth
        {

        }

        perceive(Massageclass1 message)
        {

        }

        behavior behaviorName
        {

        }

        behavior behaviorName processes (MessageClass2 message)
        {

        }

        atDeath
        {

        }

        // Any OO field or method is also supported.
}
```

Code Snippet 1

An agent is abstracted to perceive messages of definite types and have several concurrent behaviors.

Aria proposes a superior abstraction of messaging concept in comparison to the abstraction of messaging proposed by object orientation. The object orientation abstraction for messaging is to call methods on objects that is conceptually a blocking message passing mechanism. Inter-agent communication is possible through not only blocking but also non-blocking and polling mechanisms. An agent can send messages to other agents in three different ways that are through tell, tellAndWait and tellAndPoll message passing methods. An agent can send a message and proceed with its current behavior. An agent can also send a message and block for its reply before advancing. The third approach is to send a message and iteratively check for the reply. This approach called polling allows for situations when a behavior should be sustained while a message reply is also waited for. These approaches are syntactically specified as in Code Snippet 2, Code Snippet 3 and Code Snippet 4.

```
agentName.tell(messageName)
```

Code Snippet 2

```
Message message = agentName.tellAndWait(messageName)
```

Code Snippet 3

```
MessageWaitedFor messageWaitedfor = agentName.tellAndPoll(messageName)
while (!messageWaitedFor.isMessageReplied())
{
        // Do some tasks
}
Message replyMessage = messageWaitedFor.getMessage();
```

Code Snippet 4

A message class can be a subtype of RequestMessage, OrderMessage or InfoMessage classes that are themselves subtypes of Message class. A message of type RequestMessage is a message that requests a service from the receiving agent. A message of type OrderMessage requests a service that finally results in a reply message. A reply message is usually a message of type InfoMessage which contains a bit of information.

An agent may receive messages of different class types. The *perceive* block for a definite message class is where every received message that is an instance of that class is directed to.

Every agent has a hidden thread-safe message queue. All the messages that other agents send to the agent are put into its message queue. When an agent specification is compiled, a hidden message dispatching *behavior* is added to the agent *behavior*s. To be more precise, the message dispatching *behavior* is a thread that continuously iterates the message queue, identifies the type of each message and runs the *perceive* block of the identified message type with the message as the parameter. All the message queuing and runtime type identification issues are handled behind the scene by the code that is generated and inserted by the compiler into the user code and in part by classes of Aria core package. Hence the user is not needed to be concerned about queuing and other low level issues but focuses on the concerns of the problem domain.

As it was explained all the *perceive* blocks of an agent are executed sequentially in a single thread; hence, a *perceive* block can only contain a short processing on the message. For instance it can contain the action that a simple reflex agent[7] performs in realizing a message of a definite type. This is most common for user interface agents. As most of the messages they receive are requests for presentation and such requests can be carried out rapidly, user interface agents are usually reflex agents.

*Behavior*s are where the agent's intelligent processing should be coded. The code snippet of a *behavior* is translated to a repeatedly running code. Every *behavior* is executed on a separate thread by default and this supports concurrent behaviors in an agent. While the user only specifies the behavior, the threading and scheduling issues are handled by Aria.

[7] The *Simple reflex* term is from [1].

Processing needed to reply some message types may be time consuming and messages of such types can not be promptly answered. As *perceive* blocks should only contain short processing tasks, there is a need that messages of time consuming types be directed to a *behavior* to be further processed. To this end, if there were no specific construct for defining *message processing behaviors* in the language, the user was required to define a thread-safe queue to hold messages of the type and a *perceive* block for the message type to save the messages of the type to that queue. A *behavior* could then get a message from the queue to be further processed. This could be coded in Aria as coded in Code Snippet 5.

```
private ThreadSafeQueue<MessageClassType> ariaMessageClassTypeQueue =
        new ThreadSafeQueue<MessageClassType>();

percieve(MessageClass message)
{
        ariaMessageClassQueue.add(message);
}

behavior behaviorName
{
        try
        {
                MessageClass message = ariaMessageClassQueue.remove();
                // behavior code to process message
        }
        catch (Exception e)
        {
                idle();
        }
}
```

Code Snippet 5

To support the user to accomplish this much easier, Aria allows defining *message processing behavior*s. A *behavior* can declare to process a definite message type; the message queue, the *perceive* block which queues the messages and the code that dequeues messages in the *behavior* block are automatically generated by the compiler. This means that Code Snippet 6 has exactly the same effect as Code Snippet 5.

```
behavior behaviorName processes (MessageClassType message)
{
        // behavior code to process message
}
```

Code Snippet 6

A message that is being processed either in a *perceive* or *behavior* block can be replied by the reply keyword as shown in Code Snippet 7 and Code Snippet 8.

```
reply message
```

Code Snippet 7

```
reply
```

Code Snippet 8

When a message is replied, if the sender is waiting for the reply, the sender unblocks and gets the reply message as the return value of tell method. But if the sender is not waiting for the reply, the reply is simply sent to it to be queued and processed later. A reply statement without a message is exactly similar to replying with a message of type DoneMessage. Reply statements without an explicit message are usually used for unblocking sender agents that are waiting for a requested task to be finished. All the needed synchronizations are handled by Aria core package.

An agent is created and made alive as shown in Code Snippet 9.

```
AgentType agentName = new AgentType();
agentname.becomeLive();
```

Code Snippet 9

Agent's (hidden) message dispatching *behavior* and all the *behavior*s in the agent definition are started when the agent is commanded to become live. The *atBirth* block is executed when the agent is becoming live just before any of the *behavior*s are started. An agent can be requested to terminate by telling it a message of TerminateRequestMessage class. When a message of TerminateRequestMessage class is received, the agent dies by default by terminating all its *behavior*s and executing the *atDeath* block when all the *behavior*s are terminated. This default reaction to TerminateRequestMessage can also be overridden easily by providing a *perceive* block for it. An agent can terminate itself not only by sending a TerminateRequestMessage to itself but also by calling die() method.

Agent definitions support all the constructs that can be defined inside class definitions. Fields can be defined for agent information storage. As fields are accessible from all the *perceive* and *behavior* blocks and agent *behavior*s can execute concurrently, care should be taken for synchronizing field access. Methods are also allowed to be defined for an agent but rarely an agent's method has a public access. Using one of the mentioned telling approaches is proposed for inter agent message passing in contrast to object oriented method calls for message passing; Hence, methods are expected to have private or protected rather than public or package access specifiers. Private and protected methods can be employed to break the functionality in *perceive* and *behavior* blocks to manageable functions and subprocedures.

Composition serves as a way to reuse existing agents and built more high-level agents with greater capabilities from them. An agent can obviously be composed of other agents. Each

subagent can be capable of performing a part of the agent's responsibilities. The agent itself can act as a manager or coordinator.

## Inheritance

Aria agent specification supports agent specialization and servicing declarations.

### Agent Specialization

In addition to agent composition, agent specialization can be employed to achieve software reuse principle. All the capabilities present in an existing agent type can be reused by specializing a new agent from it and then the new agent can be supplemented with further capabilities.

Agent specialization is the counterpart to object inheritance. While object oriented inheritance involves fields and methods, agent orientated specialization concerns also message processing mechanisms and behaviors. When an agent inherits from a parent agent, the entire parent's *perceive* and *behavior* blocks are inherited by the agent. The child agent can perceive all the message types that its parent could perceive and has all the behaviors that its parent has. The child agent has all the parent agent's functionalities in addition to new specific functionalities that can be added.

### Service Provision

A service is specified in Aria in the syntax shown in Code Snippet 10.

```
public service ServiceType extends AnotherServiceType1, AnotherServiceType2, ...
{
        servicesTo(MessageClass1);
        servicesTo(MessageClass2);
}
```

Code Snippet 10

A service specification formally defines a service that agents may provide. A service specification declares some message or request types. An agent that declares to support a service should be able to perceive all the message types declared in the service specification. An agent that has declared the *perceive* block for a message type is able to perceive messages of that type. An agent that has a *message processing behavior* for a definite message type is also considered to be able to perceive messages of that type. This is because a *message processing behavior* automatically generates a *perceive* block for the message type it is processing. Service provision is the counterpart of interface realization in object orientation.

A service can declare to extend other services. An agent that declares to service a definite Service B that extends another Service A should be able not only to perceive all the messages

declared in Service B but also all the messages declared in A. An agent can offer different services. Different agents can provide a unique service with different policies.

## Polymorphism

A general agent with definite capabilities can be specialized to have the capabilities more specifically defined. Different specialized agents can expertize different capabilities of a generally defined parent agent. A *perceive* or *behavior* block can be overridden by an inheriting agent.

An overriding *perceive* or *behavior* block contains the child agent's specific definitions that are different from that of its parent. A *behavior* defined in a child agent that has the same name as of a *behavior* in its parent agent overrides the parent's *behavior*. An overriding *behavior* replaces the overridden *behavior* and will be active instead of it.

A *perceive* block defined in a child agent for a specific message class overrides the *perceive* block for the same message class in the parent agent. Sending a message to an upcasted child agent is polymorphic. This means that the *perceive* block defined in the child agent specification is executed rather than the *perceive* block defined in the parent agent specification.

An agent can add a *perceive* or *behavior* block to itself at runtime. The added *perceive* or *behavior* block can be a new or an overriding one. An agent can not only override the *perceive* and *behavior* blocks of its ancestors at compile time, but it can also override inherited and even its own *perceive* and *behavior* blocks at runtime. This supports an agent to change its behaviors in the course of its life as a result of learning or adaptation. Emerging a new or changing an existing *perceive* or *behavior* block can be performed at runtime as coded in Code Snippet 11, Code Snippet 12 and Code Snippet 13.

```
addPerceiver(
	new Perceiver<MessageClass>()
	{
		public void perceive(MessageClass message)
		{
			//Perceive code
		}
	}
);
```

Code Snippet 11

```
addBehavior(
        new Behavior("BehaviorName")
        {
                public void behavior()
                {
                        //Behavior code
                }
        }
);
```

Code Snippet 12

```
addBehavior(
        new MessageProcessingBehavior<MessageClass>("BehaviorName")
        {
                public void behavior(MessageClass messasge)
                {
                        //Message Processing behavior code
                }
        }
);
```

Code Snippet 13

# 4. Aria Compiler (Ariac)

Aria compiler is developed employing Antlr v.3 tool. Antlr[8] has been the most prominent compiler development tool for at least 5 past years. Ariac translates a program in Aria language to a semantically equivalent program in Java language that is then compiled to the (platform independently executable) Java bytecode. Besides agent and service specifications, Ariac compiler also accepts all the Java language constructs. It means that Aria language is a superset of Java language and Aria code is fully integrable with Java code. This maintains two favored software engineering practices that are backward compatibility and code reuse. All the existing Java packages can be still benefited from while agent definition constructs are also available. The compiler has successfully passed compiling two sample cases implemented in Aria.

# 5. Sample Cases

## Chat Room

Chat room application is selected as a case to employ Aria because of its distributed nature. The application is straightly designed though a sheer agent oriented approach as shown in Figure 1.

[8] http://www.antlr.org

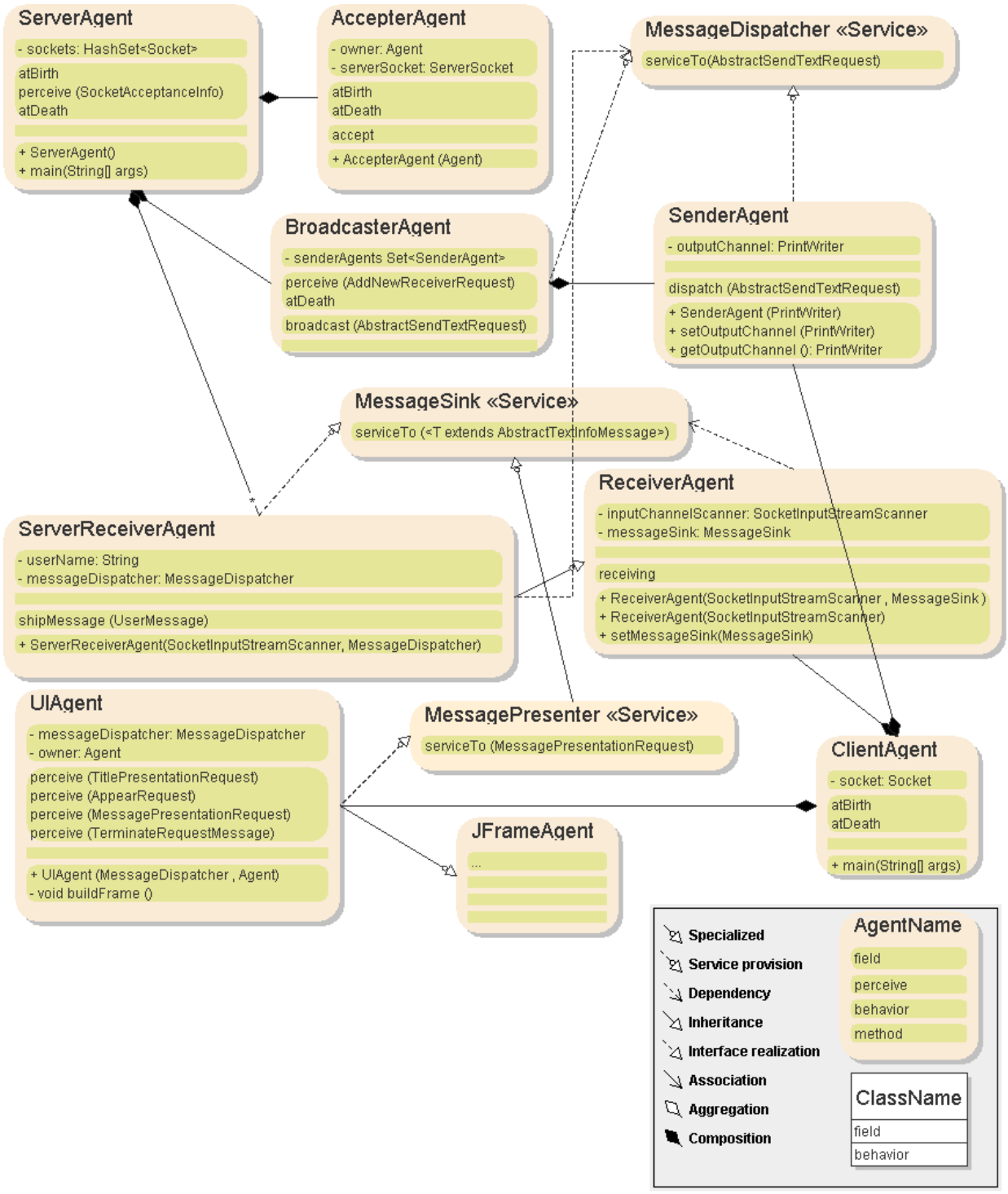


**Figure 1 Chat Room Application Architecture**

The chat room application consists of two subcomponents: ClientAgent and ServerAgent.

ClientAgent is composed of three different agents: a UIAgent, a Sender agent and a Receiver agent. The UIAgent interacts with the user. It gets chat messages from the user and sends them to the Sender agent. The Sender agent sends any message it receives to the server. The Receiver agent gets all the messages coming from the server and directs them to the UIAgent to be presented to the user.

Supporting codes are provided in Aria core package for defining user interface frames as agents while all Swing features are still present. JFrameAgent that is the parent of all the frame agents is capable of receiving requests to appear, change its title and get a line of text from the user. It can be specialized for a particular application to be capable of receiving application specific messages as well.

ServerAgent consists of an AccepterAgent, and a number of ReceiverAgents (one for each connected client) and a BroadcasterAgent. The accepter agent is responsible for accepting new client connections. A ReceiverAgent receives messages from the client and directs them to the BroadcasterAgent. The broadcaster agent consists of SenderAgents that are responsible for sending messages to the connected clients. The BroadcasterAgent sends every message it receives to all the sender agents and the sender agents send the messages to their associated clients. Hence, all of the chat users can see any message that any of them writes.

Figure 2 depicts three chat client agents all running on the same machine.

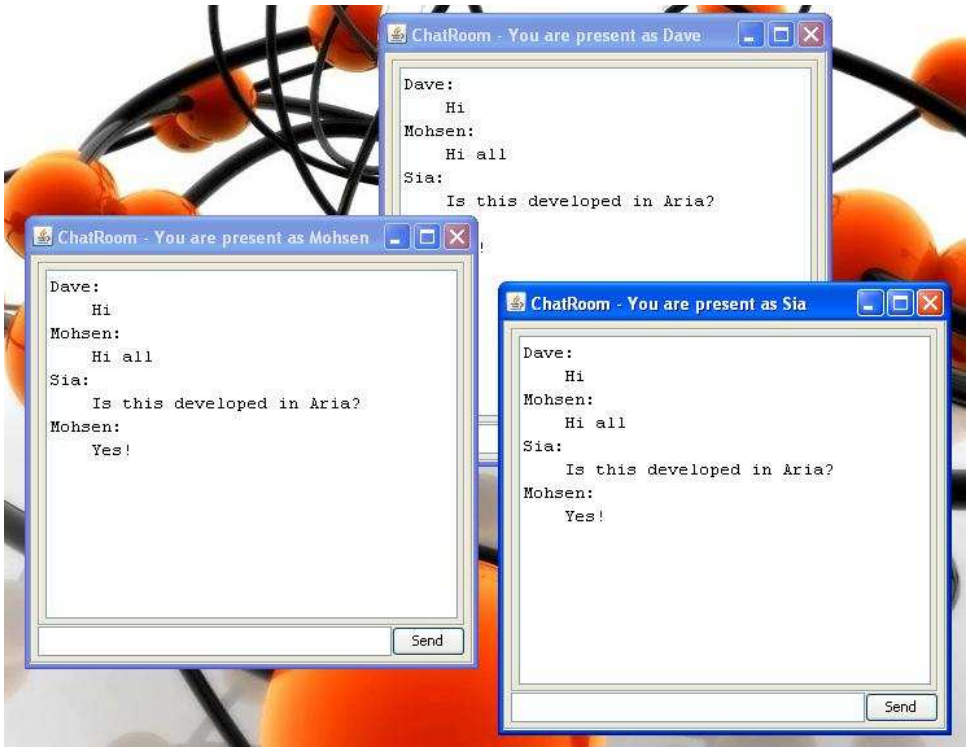


**Figure 2 Chat Room Application Snapshot**

## Agent MVC (Model View Controller)

MVC architecture offers an elegant design for a complex element in terms of three logical sub elements: [15]

- Model: The element's state, and means for changing the state.
- View: The representation of the element (visual or non-visual). There may be various views representing the model differently.
- Controller: The element's control functionality, mapping actions on the view to their impact on the model.

This separation makes it easier to modify or customize each part. As the separated parts are autonomous message passing entities, aria offers agents as ideal modeling elements for the parts. In addition to better modeling and design, Aria helps for responsiveness of views.

There is only one thread of execution that runs all the three parts in an object oriented implementation of MVC. All the process should execute in a single thread and all the message passings are through method calls. When a view should send a message to the controller as a result of user action on the view, the view calls a method on the controller. If the controller procedure is time-consuming the view freezes and becomes unresponsive until the controller finishes the called method. The problem gets worse when the controller has to update the view successively in order to show an animation, for example. The view remains frozen until the controller finishes and then all the updates that the controller has requested for are executed on the view rapidly in sequence. But in fact, this is not the gradual effect the controller intended to make. Besides, Update messages from the model to the views are also forced to execute sequentially i.e. one updates only after another even if the first is very time-consuming.

If the parts are modeled as autonomous agents, a view can send a message to the controller and then just continue its behaviors. Hence the view can always respond to the user and never freezes. The agent oriented views are autonomous and concurrent in responding to messages they receive. All views can start updating concurrently as soon as the model broadcasts a change to the views and no view waits for another view to finish updating.

The selection sort animation is implemented in Aria with MVC architecture as shown in Figure 3.

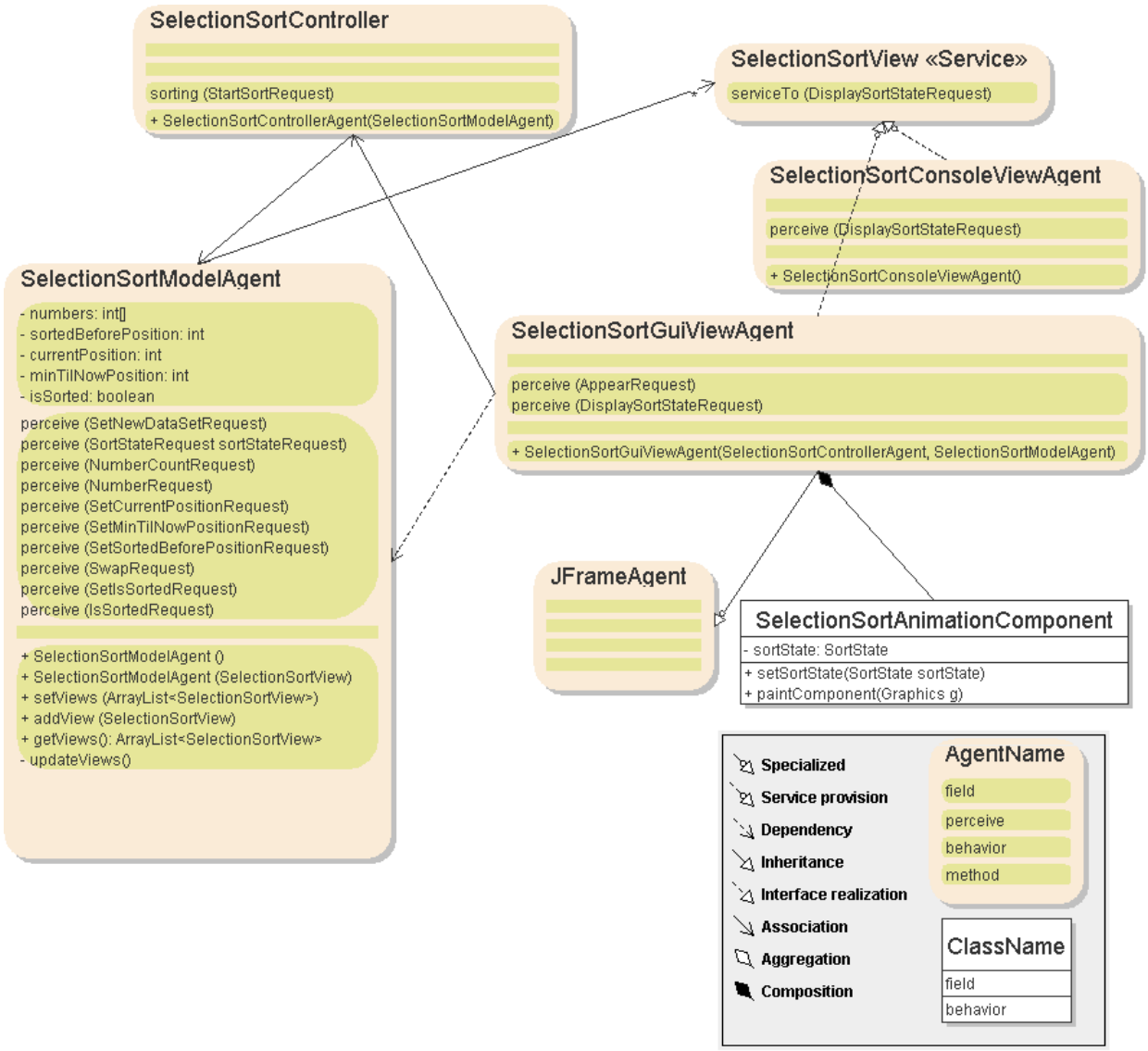


Figure 3 Agent MVC Architecture

The selection sort application consists of these three MVC sub elements:

- SelectionSortModelAgent
  receives requests to offer or change the sort state elements
- SelectionSortGuiViewAgent and SelectionSortConsoleViewAgent
  are different representations of selection sort state (a graphical and a text based). They receive similar requests to display sort state. SelectionSortGuiViewAgent sends a request to the SelectionSortControllerAgent to run the sort algorithm when the start (or restart button is pressed.
- SelectionSortControllerAgent

runs the selection sort algorithm when a message is received from the view to start the sort. It sends change requests to the model with a predefined delay between requests. The model informs any known views of any change it experiences and the views reflect the model change to the user. As the changes are made gradually to the model, updates in the view animate the sort procedure.

Figure 4 depicts a snapshot of the selection sort animation application were both the graphical and console views are demonstrating sort states.

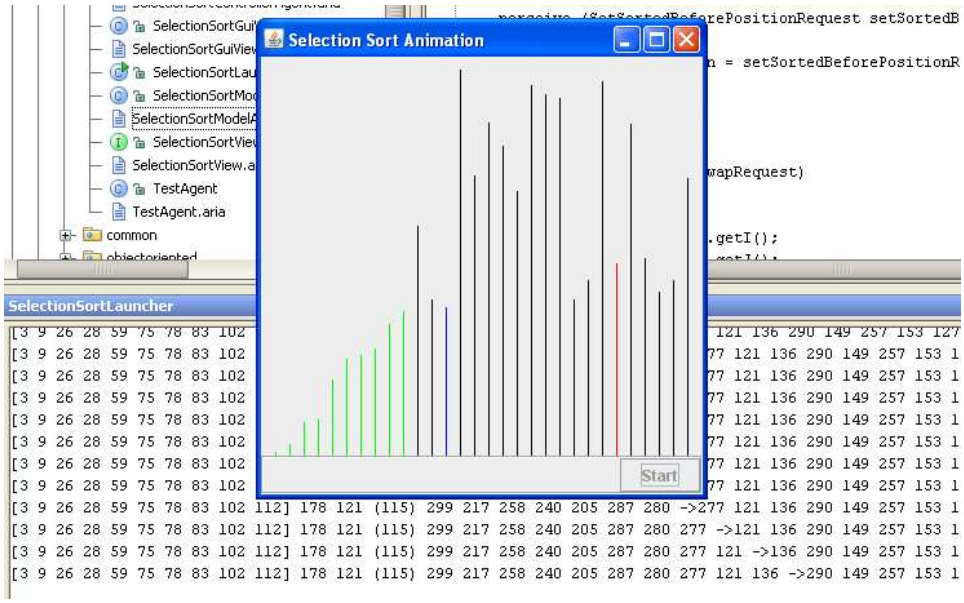


Figure 4 Selection Sort Animation Snapshot

## 6. Conclusion and Future Works

Multi agent systems represent a natural abstraction for architecture of complex systems. Agents as live entities can elegantly represent interacting elements of the system. As organizational processes are essentially interactions between individuals or organizations, constructing organizational software systems with agent oriented architectures can be straightforward. To make agent orientation a concrete paradigm, it should be developed in practice with well defined semantics. This research proposed Aria language and its compiler which supports defining agents at the language level. The offered language features are accompanied with proposed semantics. Three eminent object oriented principles are redefined in terms of the language features. A chat room application and MVC architecture are selected as cases to employ Aria. The cases are easily designed as interacting agents and the Ariac compiler successfully compiled the implemented codes.

As a future work, the compiler should be improved to provide better error handling. The language is flexible enough to support implementation of the well known BDI (Belief, Desire and Intention) [14] agent architecture and also get FIPA[9] compliant. To demonstrate the implementation of BDI agents in Aria besides maintaining FIPA standards are other future works of this paper. In addition, a plug-in should be developed for IntelliJ Idea or Eclipse IDEs in order to facilitate Aria developer's job. The agent diagrams are obtained from an editor for Aria agent oriented design developed in this research as an extension of Violet[10] UML editor. The editor and the modeling language should get more developed.

## 7. References


[1] Bellifemine, F., Bergenti, F., Caire, G. and Poggi, A. 2005. JADE - A Java Agent Development Framework. In Multi-Agent Programming: Languages, Platforms and Applications, Bordini, R. H., Dastani, M., Dix, J., and El Fallah Seghrouchni, A., Ed. Springer-Verlag. Chapter 5.

[2] Bellifemine, F., Rimassa, G., and Poggi, A. 1999. Jade - a Fipa-compliant agent framework. In Proceedings of 4th International Conference on the Practical Applications of Intelligent Agents and Multi-Agent Technology (London, 1999).

[3] Bordini, R. H., Hübner, J. F., and Vieira, R. 2005. Jason and the Golden Fleece of agent-oriented programming. In Multi-Agent Programming: Languages, Platforms and Applications, Bordini, R. H., Dastani, M., Dix, J., and El Fallah Seghrouchni, A., Ed. Springer-Verlag. Chapter 1, 3-37.

[4] Bordini, R. H., Hübner, J. F., and Wooldridge, M. 2007. Programming Multi-Agent Systems in AgentSpeak Using Jason. John Wiley & Sons, Ltd.

[5] Busetta, R., Ronnqulst, R., Hodgson, A., and Lucas, A. 1998 JACK Intelligent Agents - Components for Intelligent Agents in Java. Technical report. Agent Oriented Software Pty. Ltd, Melbourne, Australia.

[6] Dastani, M., Dignum, F., Meyer, J.J., 2003. 3APL: A Programming Language for Cognitive Agents. ERCIM News, European Research Consortium for Informatics and Mathematics, Special issue on Cognitive Systems. 53 (2003).

[7] Hindriks, K.V., de Boer, F.S., van der Hoek, W. and Meyer, J.-J.Ch., 1999. Agent programming in 3APL. Autonomous Agents and Multi-Agent Systems. 2, 4 (1999), 357-401.

[8] Howden, N., Ronnquist, R., Hodgson, A., and Lucas, A. 2001. JACK - Summary of an Agent Infrastructure. In proceedings of 5th International Conference on Autonomous Agents (2001).

[9] Jennings, N. R. An agent-based approach for building complex software systems. Communications of the ACM, 44, 4 (2001), 35-41.


[9] http://www.fipa.org/

[10] http://horstmann.com/violet

[10] Pokahr, A., Braubach, L. and Lamersdorf, W. 2003. Jadex: Implementing a BDI-Infrastructure for JADE Agents. EXP - In Search of Innovation (Special Issue on JADE, Telecom Italia Lab, Turin, Italy). 3, 3 (September 2003), 76-85.

[11] Pokahr, A., Braubach, L., and Lamersdorf, W. 2005. JADEX: A BDI Reasoning Engine. In Multi-Agent Programming: Languages, Platforms and Applications, Bordini, R. H., Dastani, M., Dix, J., and El Fallah Seghrouchni, A., Ed. Springer-Verlag. Chapter 6.

[12] Russell, S., Norvig, P. 2003. Artificial Intelligence, A modern Approach. Prentice Hall.

[13] Shoham, Y. 1991. AGENT-0: A Simple Agent Language and its Interpreter. In Proceedings of 9th National Conference of Artificial Intelligence (Anaheim, CA, 1991). MIT Press.

[14] Shoham, Y. 1993. Agent-oriented programming. Artificial Intelligence, 60, 1 (1993), 51–92.

[15] Stelting, S., Maassen, O. Applied Java Patterns. Prentice Hall, 2001, 598 pages

[16] Sutter. H. (2005) The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal, 30, 3 (March 2005).

[17] Yu, E. 2001. Agent-Oriented Modeling: Software Versus the World. In Proceedings of Agent-Oriented Software Engineering AOSE-2001 Workshop (2001). Springer Verlag, 206-225.