Scientific Journal of Riga Technical University
Computer Science. Applied Computer Systems
2011
Volume 47

Aspect-Oriented Approach to Implement Message Handler in Multi-agent Systems

Aleksis Liekna[1], Jānis Grundspeņķis[2], [1-2]*Riga Technical University*

***Abstract* – This paper focuses on message handling in multi-agent systems. The proposed approach uses aspect-oriented programming to separate message handling from other agent concerns, thus increasing system's modularity and simplifying modification and expansion. To illustrate the proposed approach in practice, a prototype of a simple knowledge base agent model is implemented. The prototype is built on top of JADE platform. AspectJ is used for aspect-oriented implementation.**

Keywords: **AspectJ, Aspect-Oriented Programming, JADE, Message Handling, Multi-Agent Systems.**

I. Introduction

Sending and receiving messages is an important factor in multi-agent system communication process. Agents communicate with each other via message exchange. Some of these agents may have message receiving and message sending as their only available sensors and actuators. A number of international standards (provided by FIPA [1]) try to describe the structure of messages and to make interaction protocols applicable, and they are doing well. FIPA's ACL [2] is now a standard for multi-agent system communication [3], and development platforms (such as JADE [4], SPADE [5] and JACK [6]) support it. With relative ease standard-compliant messages can be composed, sent, received on the other end and information contained in them obtained.

The trouble comes when there is a need to figure out what to do with that information. For instance, if we receive message A, we need to perform action X, but if we receive message B, the action to be performed is Y and so on. This often results in lots of if-then statements in the main message receiving cycle, which in turn leads to the need of implementation of multiple agent concerns in a messaging module. This ruins the separation of concerns – a messaging module is responsible for messaging, not for application logic.

Although it is crucial to separate different agent concerns in order to develop and implement a modular and maintainable multi-agent system [7]. From our point of view, this is especially true for message handling. If messaging is not separated from other agent concerns, but hard coded in them, system modularity is violated and maintainability becomes a problem. Traditional object-oriented approach alone does not offer a solution for this problem. As it is described in this paper, a solution can be provided combining object-oriented and aspect-oriented approach.

The rest of this paper is organized as follows. In Section II the current situation is described, that is, message handling without aspect-orientation and problems to be solved are discussed. Section III covers related work on using aspects in multi-agent systems. In Section IV the approach for message handler implementation is proposed. Section V is dedicated to the implemented prototype. In Section VI conclusions and information about future work are given.

II. Current situation

There are two common techniques for implementation of message handling [8]. The first one is the functional approach – there is one main message receiving handler per agent. It typically has lots of if-then statements for analyzing the message and then performing the appropriate action based on the message contents.

The second one uses object-oriented approach. There are multiple message receiving handlers per agent, each of them focusing on concrete message type. This is implemented using message filters and polymorphism. In this case each type of message is received by the appropriate message handler.

In both cases it is necessary to determine what kind of message is received and then the appropriate action is executed. There are two common approaches to do this. In the first case, message handler passes the received message to the appropriate agent component which extracts message contents and performs the required action (if such action exists). To do that, it is needed to include messaging-specific data structures and logic into all agent components to which the messages are passed (see Fig. 1). It is assumed here that message handler (or message handlers if there is more than one) resides in the messaging component.

The problem is that messaging is now implemented not only in the messaging component, but also in all related agent components (such as learning, adaptation and so on). This ruins the separation of agent concerns – one agent concern is implemented in multiple component modules. Here the term "module" stands for the implementation of the component.

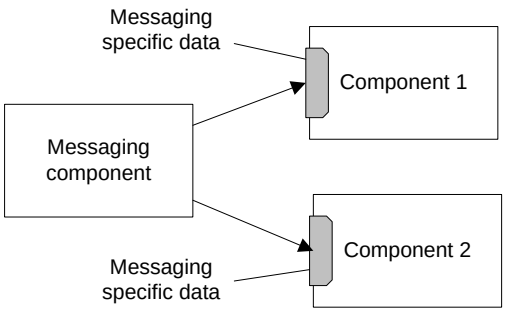

Fig. 1. Messaging component passes received message to other components

This situation is described in [7] as *architectural shattering*– implementation of one agent concern (messaging in our case) is shattered among other agent concerns. This also affects system evolution – if messaging component needs to be changed, other components might need to be changed too. From the architectural point of view there is no single module where messaging is implemented and it is not possible to reason about messaging while looking at the messaging module alone – one needs to take into account all modules of other agent components where messaging is implemented.

In the other case a message handler extracts message contents, performs data translation operations (if necessary) from message-specific to appropriate component-specific format and passes that data to the corresponding agent component (or components). In this case it is needed to include data structures and (possibly) logic of all the related components in the messaging component (see Fig. 2). The problem is that messaging component module not only implements messaging, but also partly implements other agent components. Such situation is described in [7] as *architectural tangling* – implementation of multiple agent concerns in a single module. This also burdens system modification, extension and evolution, as it is not possible to plan, design and implement changes in the messaging component without looking at all other components that are partly implemented in it.

The problem is illustrated by the following example. Suppose a knowledge base agent. This agent stores knowledge as key–value pairs (for example – key: "football", value: "result 5:2"). Agent can receive knowledge and respond to knowledge queries from other agents. Such an agent has two components – a knowledge component and a messaging component. The knowledge component is responsible for knowledge storing and retrieving. The messaging component is used for communication.

A typical implementation of such agent is depicted in Fig. 3. A simple event listening approach, derived from [9] is used in this example. In this approach there are components that produce events (called producers) and components that listen to these events (called listeners). A listener must subscribe to events it needs no listen. When an event fires all listeners subscribed to this event are notified of it.

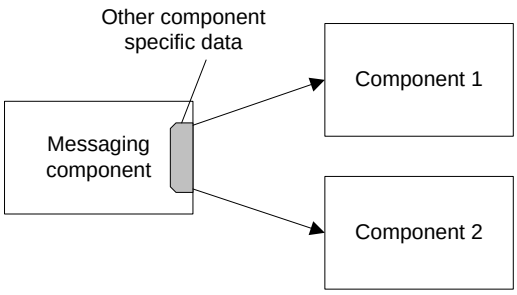

Fig. 2. Messaging component passes component-specific data to other components

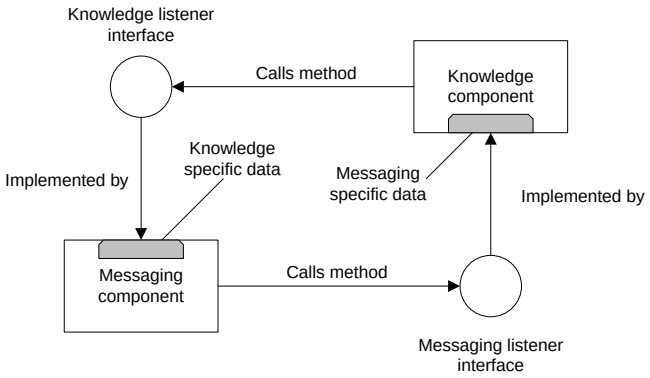

Fig. 3. A typical implementation of the knowledge base agent

There is a listener interface associated with each agent component. This interface declares methods that are called when specific events take place. Listener subscribes to those events by implementing such an interface and adding itself to that component's event listeners list. Event listeners list is implemented as a vector of listener interfaces. Adding an event listener incorporates adding an agent component instance that implements such listener interface to this vector.

When the appropriate event fires (e.g., a message is received), an interface method of each listener from the list is called. An example of messaging listener interface is given below. Java programming language is used in this and further examples.

```
public interface IMessagingListener {
  public void notifyMessageReceived(Object message);
}
```

The messaging listener interface declares a method "notifyMessageReceived" to be implemented in potential event listeners that need to be informed of a message arrival.

Knowledge component needs such information, as it has to store the information received and respond to information queries. A simplified example of knowledge component implementation is given below. It implements the messaging listener interface and overrides the method "notifyMessageReceived" to get notifications about message arrival.

```
public class KnowledgeComponent implements
IMessagingListener {
  @Override
  public void notifyMessageReceived(Object message){
    //process received message
  }
...
```

Finally, there is a main loop in the message receiving component (message handler), where messages are received and passed to all listeners. The example code is given below. Note that the message is passed to listeners unmodified. So each listener has to extract message contents and decide what to do with that message.

```
public void receiveMessages(){
  while(true){
    Object message = this.getNewMessage();
    for(IMessagingListener listener :
this.listeners){
      listener.notifyMessageReceived(message);
    }
  }
}
```

It means that message-specific data and message extracting logic must be included in all listeners. Within the example it means that knowledge component not only has to deal with knowledge-specific, but also with messaging-specific logic and data.

This is a typical example of both architectural shattering and tangling described earlier in this section. From the messaging component point of view architectural shattering takes place, since messaging is implemented not only in the messaging component, but also in the knowledge component. From the knowledge component point of view, architectural tangling occurs, since knowledge component implements both knowledge and (partly) messaging. This situation is depicted in Fig. 3, where grey fields in messaging and knowledge components show the presence of tangled code.

So, what happens if messaging (i.e. ontology, message content language, the mechanism of extracting message contents, etc.) needs to be changed? The knowledge component needs to be changed, too! If there is only one component related to messaging, this is not a big problem, but what if there are tens or even hundreds of components? Each of them has to be changed. It is unacceptable from the architectural point of view – changes in one component of the system cause changes in a number of other components. This is the problem that needs to be solved in order to develop evolvable and maintainable multi-agent systems.

III. Related work

Generally, separation of different concerns can be achieved in a number of ways. According to [10] this includes frameworks, code generation, design patterns, dynamic languages and aspect-oriented programming. From our point of view aspect-oriented programming is the most suitable solution for this problem because of reasons described below. Other alternatives either provide solutions for specific problems (such as frameworks that offer filters for dealing with HTTP requests), solve the problem only partially (this is the case with design patterns), or is too complex (lots of configuration needed and low-level syntax detail understanding required in the case of code generation) to be useful. Aspect-orientation, on the other hand, is a general approach and it can be combined with other candidates from the list to solve the problem even better.

Garcia et al. [7], [11], [12], [13], [14] offer separation of multi-agent system concerns using aspect-oriented programming. The main idea behind this approach is the use of aspects to separate crosscutting concerns (such as interaction adaptation, autonomy, etc.) from the agent's basic functionality.

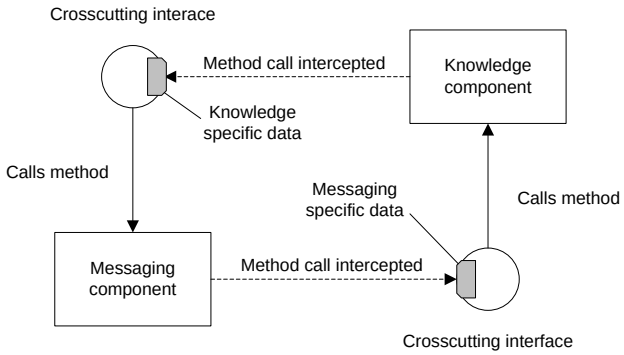

Fig. 4. Implementation of knowledge base agent example using the approach proposed by Garcia et al

Each crosscutting concern is encapsulated in an aspect and represented by a crosscutting interface [7]. Crosscutting interface provides services to the system and specifies the interaction with other components of the system, as it is described in [7] and [12]. Crosscutting interfaces (implemented as aspects) intercept dynamic behavior (such as method calls and object creation) in other components of the system and perform the appropriate actions (such as calling a method) in the component they represent.

To illustrate the approach presented by Garcia, let's return to the knowledge base agent example introduced in the previous section. The implementation of this example (shown in Fig. 3), using the Garcia's approach is depicted in Fig. 4. The idea is as follows.

When a message arrives, the method "receiveMessage()" in the messaging component class (described in previous section) returns. This behavior is intercepted by the crosscutting interface of the knowledge component. It uses the method's context, such as parameters and the value returned, to obtain the received message. Afterwards, it extracts message content, converts the latter to knowledge-specific format and calls the appropriate method of the knowledge component. The messaging component is unaware of such activities, so it does not have to include any knowledge-specific data or refer to some listener interfaces. It simply receives messages while crosscutting interface of knowledge component "takes what it needs". Message sending can be implemented in a similar way. In this case the crosscutting interface of the messaging component intercepts method call in the knowledge component, converts knowledge-specific to messaging-specific data and calls the message sending method in the messaging component.

This technique really works – components do not have to include any crosscutting code (code implementing other components), so both architectural shattering and tangling are eliminated at the component level. Crosscutting interfaces take care of all the crosscutting logic.

Nevertheless, from our point of view, the problem is far from being solved – it is just moved one level higher – from components to their crosscutting interfaces. Fig. 4 shows that knowledge-specific data is present in the crosscutting interface of the messaging component, as well as crosscutting interface

of the knowledge messaging component includes some messaging-specific data.

The drawback of this approach is that each crosscutting interface is directly accessing the internal structure (intercepting method calls) of other agent components. Suppose that there is a component whose internal structure is accessed by 100 aspects. What happens when the internal structure of this component must be changed? All 100 aspects must be changed, too!

Furthermore, this approach threats the basic functionality of an agent as a special case of component – it does not have a crosscutting interface – only the "regular" one. In this case the problem described in the previous section is not solved at all, because other components still have to refer directly to the interface of the basic functionality. Hereby, the basic functionality of an agent is shattered all over the system in the crosscutting interfaces of other components. So, when the basic functionality changes the crosscutting interfaces of all other components might also need to be changed.

Another approach promoting the separation of concerns in multi-agent systems using the aspect–oriented approach is presented by Amor et al. [15], [16]. They introduce *Maleca* – an architecture that combines both component-based and aspect-oriented techniques. Their main idea is to assemble the multi-agent system from commercial-of-the-shelf (*COTS*) components and tie them together using the aspect-orientation. Although this provides a way out for reusability, the whole solution depends on specific solutions (COTS components) already available. Yet, the most specific solutions are almost always designed from scratch. Therefore more general approach is needed, which is proposed in the next section.

Besides the, aspect-orientation can also be used for multi-agent system testing [17] and observing [18] thus separating them from other agent concerns. Discussion on these topics is beyond the scope of this paper.

IV. Proposed approach

The proposed approach determines that every agent component has its own *aspect interface* and *aspect listener*. An aspect interface represents a specific agent component and provides services to other agent components. It has an empty implementation – a class instance that implements this interface and has all of the method bodies empty. It also has an aspect tied to it. This aspect monitors the agent component which it represents and intercepts corresponding method calls in the execution process. Then it translates the method context (such as method parameters, returned value and the object on which the method is executed) from component specific to common data structures. Afterwards it calls the appropriate aspect interface method passing the translated context as parameters.

Aspect listener, on the other hand, monitors aspect interfaces of other agent components and intercepts those method calls in which it is interested. Then it translates common data structures to component-specific ones and calls the appropriate method of the component.

The common data structure introduced here is a data structure that both the aspect listener and the aspect interface of each concerned agent component understand. It is used to overcome both architectural shattering and tangling. Each agent component may have its own internal structure. For cooperation of those components, some kind of mapping is needed. In order not to tangle agent component X into agent component Y and vice versa a common data structure Z is introduced. X can be converted into Z and Z can be converted into Y. So X can be taken from one agent concern and converted to Z. Then Z can be converted to Y and passed to other agent concern. To make the conversion transparent for both concerns a mediator (the converter) is needed. The aspect interface and the aspect listener together provide such a mediator. The aspect interface provides the conversion from X to Z at one end, while the aspect listener converts Z to Y at the other end. The conversion is transparent for both X and Y because the aspect-oriented approach is used. Neither X nor Y calls the conversion process directly – it is encapsulated in aspects.

To illustrate this, let's consider the knowledge base agent example, presented in Section II once again. The implementation of this example using the proposed approach is shown in Fig. 5. Solid arrows denote direct method invocation, while dashed arrows denote method call interception using aspects. Let's examine the data flow between the elements. Consider that the knowledge base agent receives a message that contains a knowledge query – a request for knowledge (simply "request" further in the text). Request is received in the messaging component. Using message filters and polymorphism it is ensured that a specific message handler instance receives this message. When this happens, the messaging interface aspect intercepts the appropriate method call (1 in Fig. 5), extracts the message content and converts the latter into common data structure (such as a request class instance). Then it calls the messaging aspect interface method (2 in Fig. 5) which has an empty body. This call is intercepted by the aspect listener of the knowledge component (3 in Fig. 5). Instead of execution of the method's empty body, it converts common request data to knowledge-specific request data and calls the method (e.g. receiveRequest()) responsible for receiving a request (4 in Fig. 5). Then, the knowledge interface aspect intercepts the completion of request (e.g. when method getKnowledge() returns) (5 in Fig. 5). After that, it converts knowledge-specific data (the knowledge requested) to common data and calls the corresponding knowledge aspect interface method (6 in Fig. 5). This method call is intercepted by the aspect listener of the messaging component (7 in Fig. 5). The latter converts common knowledge data to messaging-specific knowledge data (fills the reply message content with knowledge data). Then it calls the appropriate method in the messaging component to send the reply message containing knowledge initially requested (8 in Fig. 5).

As one can observe, the proposed approach successfully resolves both architectural shattering and tangling problems.

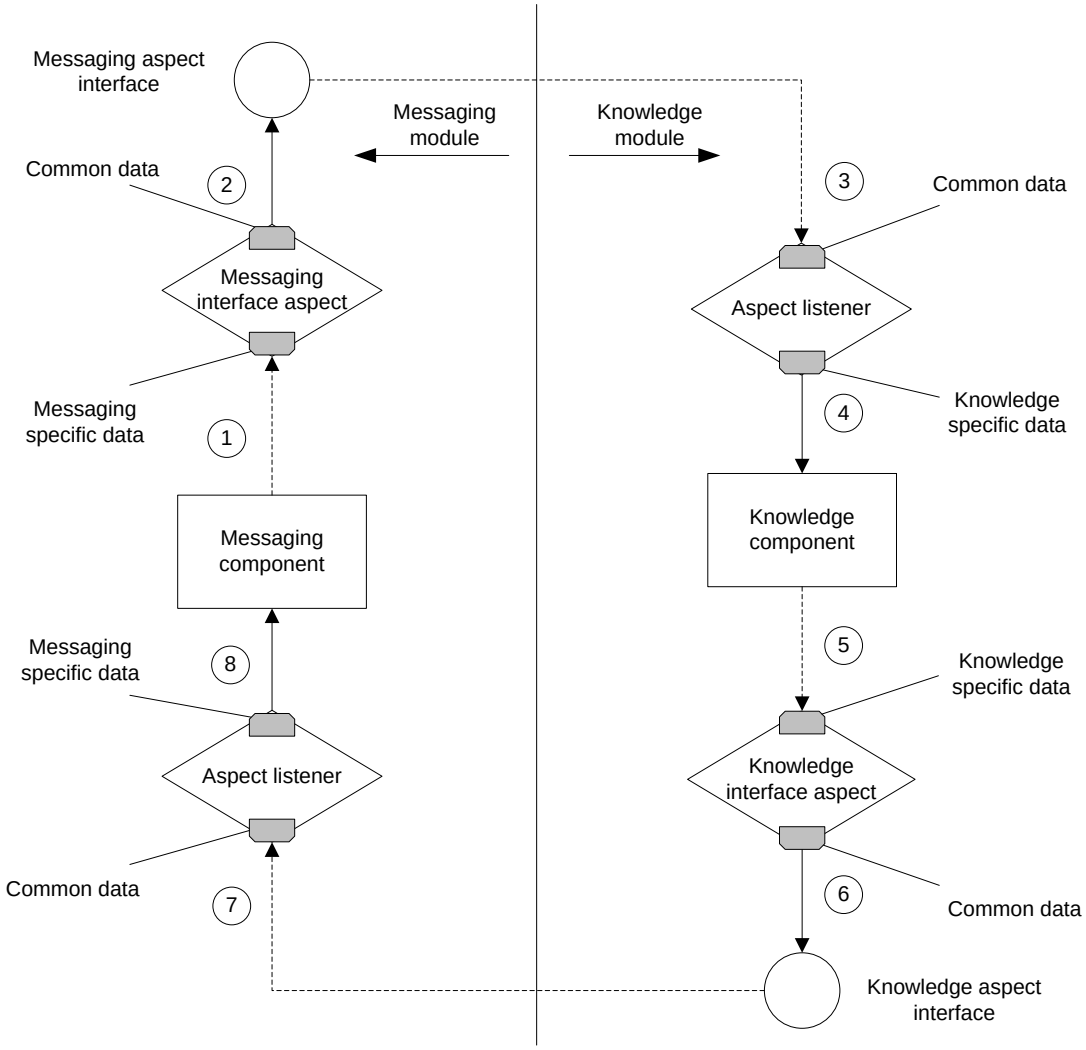

Fig. 5. Implementation of knowledge base agent example using the proposed approach

Let us describe differences between the proposed approach, the event-based approach and the one proposed in [7] and [12] by Garcia et al.

First of all, the proposed approach incorporates no direct references from one agent component to another, which makes it possible to design and implement agent components completely independent of one another. It is also possible to use COTS components described in [15]. Such references, however, exist in both event-based approach and the approach proposed by Garcia. In the case of event-based approach the whole technique is based on direct interface references (as described in Section II) which makes independent component development nearly impossible. In the approach proposed by Garcia direct references exist from other (crosscutting) concerns to the interface of agent basic functionality. It makes developing of other concerns (such as messaging) impossible, if the basic functionality component does not exist.

Second, our approach introduces the common data concept. This allows agent components to be developed independently of the system they are being used in. Integration within the system is provided by the aspect listener and the aspect interface as described earlier in this section. When the integration takes place, one must define a common data format that serves as a mediator between the data structures of different agent components. Neither event-based nor Garcia's approach benefits from such common data concept.

In summary, the proposed approach can be viewed as inverted event-based technique with common data concept. In traditional event-based approach the object that initiates the event calls the method of the event listener (through the interfaces in Java, function pointers in C/C++, delegates in C# etc.). In the proposed approach the listener actually "listens" the object. It is not like: "Hey, I have a new message, now, you get it and you get it!" but more like: "Hey, new message arrived there (in the other component); let me take a look at it!"

V. Implemented prototype

To demonstrate the proposed approach in practice we have implemented a prototype of the knowledge base agent

example introduced in Section II. The main purpose of this prototype is to show that the proposed approach really works, i.e., it can be applied not only in theory, but also in practice. The prototype is build on top of JADE platform using AspectJ to provide aspect-oriented programming support. Some of the implementation details as well as working example of the prototype are presented in this section. The implementation is outlined first while the prototype itself is described later on.

JADE platform is used since it is compliant with FIPA standards, is more or less a general purpose agent development framework and fully respects the idea of autonomous agents that are loosely coupled. Detailed discussion on JADE is beyond the scope of this paper and the reader is encouraged to visit JADE official home page [4] for further reference. The basic understanding of the specifics of JADE is recommended before continuing with this chapter.

The implementation is as follows. An agent should be developed component by component. First the fixed component itself must be implemented (it is also possible to use a COTS component if available). Then the aspect interface is implemented for that component. The aspect listener for that component can be created as soon as all the aspect interfaces of the components it needs to listen are created (the chicken and the egg problem). To illustrate this, the main steps of creating the knowledge base agent are outlined.

The knowledge base agent consists of two components – the messaging component and the knowledge component. Since this paper focuses on message handling the messaging component is discussed in detail. The messaging component itself is created first. Message filters and polymorphism are used to create multiple message-receiving handlers each of them receiving concrete type of message. Two message-receiving handlers are created for knowledge base agent, because it must receive both the knowledge and knowledge requests. Fig. 6 depicts an UML class diagram of the message handlers for the knowledge base agent.

Since the prototype is built on top of JADE platform message handlers are implemented as agent behaviors. The class "MessageReceiverBehaviour" is an abstract base class used for message handling. The main message receiving cycle is defined in the method "receiveMessages", where the method "getNewMessage" is periodically called in order to receive a new message. The method "parseMessage" is called upon the message arrival. The body of this method is left empty since it will be the subject of later aspect interaction. The method "getNewMessage" is implemented in such way that it returns only the messages corresponding to specific message filter returned by method "getMessageFilter". The latter is defined as abstract, so the derived message handlers can specify the message types they are interested in by overriding this method. As the knowledge base agent must receive both knowledge requests and the knowledge itself, two message handlers are defined, respectively the "KnowledgeRequestReceiverBehaviour" and the "KnowledgeReceiverBehaviour" as it is depicted in Fig. 6.

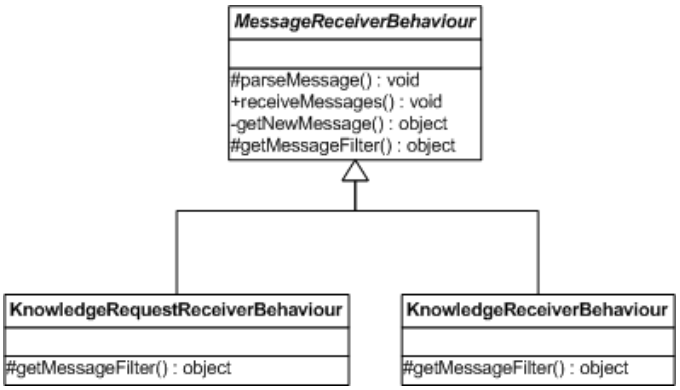

Fig. 6. An UML class diagram of message handlers for the knowledge base agent

Each of these handlers override the "getMessageFilter" method defined in the base class to specify message types they must receive and handle.

Next an aspect interface must be created for the messaging component of the knowledge base agent. This interface is responsible for announcing a specific message arrival and is created as follows. First a regular interface which contains all methods other agent components' aspect listeners will be able to intercept must be defined. In this case the interface will consist of two methods for indicating the arrival of knowledge, and knowledge request respectively. An example of such interface is depicted in Fig. 7 (a).

After creating an interface, it must be implemented. Method bodies are left empty in the implementation, an example of which is shown in Fig. 7 (b). It is ensured that exactly one interface implementation instance per agent will exist at runtime. This is achieved by passing the agent instance to the constructor of the interface implementation. When both the regular interface and its implementation are ready, the interface aspect can be created. The interface aspect defines pointcuts (dynamic points in the component execution that must be intercepted) and the advice (the code that needs to be executed when specific pointcut is reached). The advice code converts component-specific data to common data structures and calls the appropriate interface implementation method passing these structures to the method as parameters. An example of pointcut, which intercepts knowledge message arrival, is shown in Fig. 7 (c). The corresponding advice is shown in Fig. 7 (d).

The creation of the aspect listener is a bit simpler. All that needs to be done is to create pointcuts that specify aspect interface methods of other agent components that must be intercepted, and define advices for them. The aspect listener of the messaging component is responsible for sending messages when specific pointcuts are reached in the aspect interfaces of other agent components (the specific methods of these interfaces are called). In this case the advice converts common data to messaging-specific data, as well as prepares and sends the message by invoking the message sending method of the messaging component. An example pointcut intercepting the call of "knowledgeRequestDone" in the core aspect interface is depicted in Fig. 7 (e).

(a)
```
public interface IMessageReceivingInterface {
    public void knowledgeRequestReceived(KnowledgeRequest knowledgeRequest);
    public void knowledgeReceived(Knowledge knowledge);
}
```

(b)
```
public class MessageReceivingInterface implements IMessageReceivingInterface {
    @Override
    public void knowledgeReceived(Knowledge knowledge) {
    }
    @Override
    public void knowledgeRequestReceived(KnowledgeRequest knowledgeRequest) {
    }
    ...
```

(c)
```
pointcut knowledgeReceived(KnowledgeReceiverBehaviour behaviour, ACLMessage message) :
    execution(* MessageReceiverBehaviour+.parseMessage(ACLMessage))
    && target(behaviour) && args(message);
```

(d)
```
after(KnowledgeReceiverBehaviour behaviour, ACLMessage message) :
    knowledgeReceived(behaviour, message) {
    Knowledge knowledge;
    ...
    //initialize the knowledge from message content
    ...
    behaviour.getKnowledgeBaseAgent().getMessgeReceivingInterface().knowledgeReceived(knowledge);
}
```

(e)
```
pointcut knowledgeRequestDone(CoreInterface coreInterface, Knowledge knowledge) :
    execution(* CoreInterface.knowledgeRequestDone(Knowledge))
    && target(coreInterface) && args(knowledge);
```

(f)
```
after (CoreInterface coreInterface, Knowledge knowledge):
    knowledgeRequestDone(coreInterface, knowledge) {
    ...
    //translate knowledge to message-specific data, put it into message and send the message
    ...
}
```

Fig. 7. Code samples for aspect interface ((a), (b), (c) and (d)) and aspect listener ((e) and (f)) of the messaging component

This method is called by the core interface aspect, when a knowledge request is done. The appropriate advice is shown in Fig. 7 (f). It converts common data to messaging-specific data, puts that data into the message and sends the message. And that's it – a working message handler has been implemented!

If following the instructions above, one can implement the proposed approach and use it in practice to create a modular, maintainable and expandable message handler, which is separated from other agent components. The rest of this section describes the implemented prototype.

The prototype consists of two agents – a knowledge base agent and a knowledge base client agent. The knowledge base agent receives knowledge as key-value pairs and responds to knowledge requests (is asked for knowledge by key). The client agent sends these knowledge requests and the knowledge itself to the knowledge base agent. To simulate request processing workload a delay is introduced. So it takes a second to process the request. The knowledge base agent is implemented in a way that only one request can be served at a time. If a request arrives while another one is in progress it is rejected, otherwise it is accepted, processed and answered later on. The example screen of our implemented prototype is shown in Fig. 8. This is the knowledge base client agent GUI window. Two units of knowledge have been sent to the knowledge base agent: key: "football", value: "5:2" and key: "football", value: "rainy". Then the knowledge base agent is asked for knowledge about football and the response is "5:2; rainy". Of course, one can object that there is no "real" knowledge, but for the sake of simplicity a very basic structure is represented here. The discussion about "real" knowledge structures (such as rules, frames and scripts) is beyond the scope of this paper.

VI. Conclusions and Future Work

The implemented prototype clearly shows that aspect-oriented approach can be successfully applied to multi agent systems in order to implement a modular, maintainable and expandable message handler.

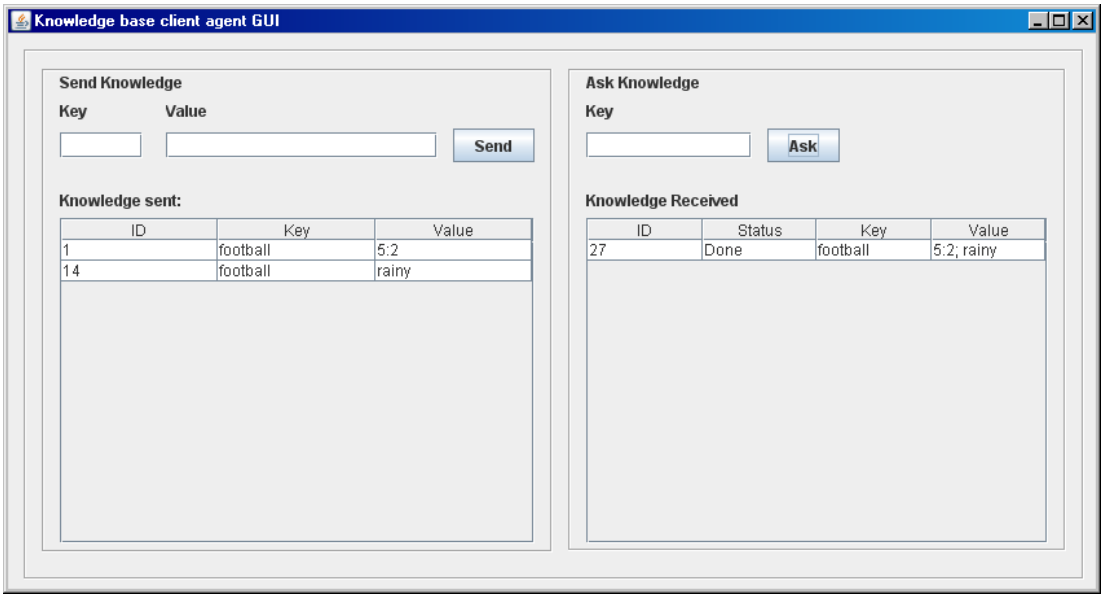

Fig. 8. An example of the knowledge base client agent GUI window

The proposed approach allows the design and implementation of agent components independently of the message handler used. This allows more flexibility in the component design as messaging-specific code is no longer tangled with other agent components and shattered across the system.

The separation of message handler from other components of the agent provides more distinct view of the system as one can observe message sending and message receiving in the aspect interface and the aspect listener of the messaging component. There is no need to look for a message sending code in the knowledge or in the learning component.

Aspects should be used with care as it is very easy to "over-aspectize" the system. Aspect-oriented programming provides constructs that allow placing virtually all of the system code in one module leaving only class declarations as the core concern. One must distinguish between the things really needed to be aspectized and the ones that are better left out. The introduction of an agent component should be encapsulated in an aspect in order to achieve modularity. Nevertheless, the component itself should not be contained in the aspect – it must be a separate entity. An aspect serves only as a mediator between the component and the rest of the system; it should not implement any other functionality.

Inter-aspect dependencies should also be addressed with care. If one makes an aspect that is based on aspect, which is based on other aspect and so on, one can come down to a situation when changing a line of code won't allow the system to compile anymore. It will take a long time before realizing where the real problem is.

Components should be designed with aspects in mind. Although it is possible to apply aspects to any component, our experience shows that it is much easier if the component is designed for it. This eases the creation of pointcuts and keeps the system structure clear. If the component under consideration is poorly structured the application of aspect is not an easy task. It is difficult to identify the pointcuts such as a message arrival if there is only one message handler instance that has only one method. In such case a very specific pointcut definition is required. It is hard to implement such a pointcut and the implementation is not very stable. If the inner structure of the component slightly changes the pointcut will have to be completely re-implemented. If the component is well structured, but is not designed with aspects in mind, the creation of pointcut still depends on the specificity of the inner structure of the component. The implementation of such pointcut is much easier comparing with the case of poorly structured component and is more stable, but not stable enough, because of the dependency on the inner structure of the component. When designing a component with aspects in mind, one can create special methods (possibly with empty bodies) later to be intercepted by aspects. This leads to creation of well structured components in which specific pointcuts are easily identifiable. Even if such component is not used in the context of aspects it is still more structured and, as a consequence, more modular and understandable than a regular one. Changing the inner structure leads to little or no changes in the pointcut implementation as long as there are no major changes in the component. Of course, no technique will help if changes are crucial, but when component is designed with aspects in mind, it is possible to implement minor changes without breaking pointcuts. To achieve this, changes must be implemented in a way they do not affect the special methods created earlier.

This paper focuses on message handling. Nevertheless, the proposed approach can be used for implementing the whole multi-agent system and this is our primary goal – to create a better architecture for multi-agent systems. To successfully

achieve this goal, the development of a tool that facilitates the developer's work is planned. The developer should be focusing on development of the inner structure of agent components, not on implementation of the pattern we present. So a tool that generates component interaction code, given components themselves and the scheme of the interaction is needed. This is the topic of future research.

Another goal of future research is to test the approach in different agent frameworks. Currently it has been tested only with JADE, but further testing with other platforms is needed to discover potential advantages and pitfalls.

REFERENCES

[1] FIPA. (2010, Sept. 10). *FIPA Home Page* [Online]. Available: http://www.fipa.org/
[2] FIPA. (2010, Sept. 10). *Agent Comunication Language Specifications* [Online]. Available: http://www.fipa.org/repository/aclspecs.html
[3] F. Bellifemine, *et al.*, "Developing multi-agent systems with JADE," *Intelligent Agents VII Agent Theories Architectures and Languages*, pp. 42-47, 2001.
[4] JADE. (2010, Sept. 5). *Jade - Java Agent DEvelopment Framework* [Online]. Available: http://jade.tilab.com/
[5] SPADE. (2010, Sept. 5). *The SPADE Agent Platform* [Online]. Available: http://spade.gti-ia.dsic.upv.es/
[6] JACK. (2010, Sept. 5). *JACK home page* [Online]. Available: http://www.agent-software.com.au/products/jack/
[7] A. Garcia and C. Lucena, "Taming Heterogeneous Agent Architectures," *Communications of the ACM*, vol. 51, pp. 75-81, 2008.
[8] F. Bellifemine, *et al.*, *Developing Multi-Agent Systems with JADE*. Chichester: John Wiley & Sons Ltd, 2004.
[9] JavaWorld. (2010, Sept. 9). *Events and listeners* [Online]. Available: http://www.javaworld.com/javaworld/javaqa/2000-08/01-qa-0804-events.html
[10] L. Raminvas, "AspectJ in Action Second Edition," p. 519, 2010.
[11] A. Garcia, *et al.*, "The Interaction Aspect Pattern," in *Tenth European Conference on Pattern Languages of Programs*, Germany, Irsee, 2005, pp. 587-606.
[12] A. Garcia, *et al.*, "Aspectizing Multi-agent Systems: From Architecture to Implementation," in *Software Engineering for Multi-Agent Systems III*. vol. 3390, R. Choren, *et al.*, Eds., ed Heidelberg: Springer, 2005, pp. 121-143.
[13] A. Garcia, *et al.*, "Aspects in Agent-Oriented Software Engineering: Lessons Learned," in *Agent-Oriented Software Engineering VI*. vol. 3950, J. Müller and F. Zambonelli, Eds., ed Heidelberg: Springer, 2006, pp. 231-247.
[14] A. Garcia, *et al.*, "The Mobility Aspect Pattern," in *Proceedings of the 4th Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP '04)*, 2004.
[15] M. Amor and L. Fuentes, "Malaca: A component and aspect-oriented agent architecture," *Information & Software Technology*, vol. 51, pp. 1052-1065, 2009.
[16] M. Amor, *et al.*, "Separating Learning as an Aspect in Malaca Agents," in *Agent and Multi-Agent Systems: Technologies and Applications*. vol. 4953, N. Nguyen, *et al.*, Eds., ed Heidelberg: Springer, 2008, pp. 505-515.
[17] R. Coelho, *et al.*, "Unit testing in multi-agent systems using mock agents and aspects," presented at the Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems, Shanghai, China, 2006.
[18] T. Mehmood, *et al.*, "Framework for Modeling Performance in Multi Agent Systems (MAS) using Aspect Oriented Programming (AOP)," in *The Sixth Australasian Workshop on Software and System Architectures (AWSA 2005)*, Brisbane, 2005, pp. 40-45.

Aleksis Liekna received his Bc.sc.ing degree in 2008 and his Mg.sc.ing. degree in 2010 from Riga Technical University. At the moment he is a PhD student at Riga Technical University. His major field of study is computer science. He is working as a Programmer for SIA ZetCOM and also as a Research Assistant at Riga Technical University. His research interests include artificial intelligence and multi-agent systems.

He is awarded by the Latvian Foundation for Education for his bachelor thesis "Development and Implementation of Reinforcement Learning Model".

Janis Grundspenkis graduated from Riga Politecnical Institute (now Riga Technical University) in 1965. His major was electrical engineer of automation and telemechanics. He received his Dr.sc.ing. degree from Riga Polytechnic Institute in 1972 and his Dr.habil.sc.ing. degree in 1993 from Riga Technical University.

He is a professor of systems theory at Riga Technical University. He is also a Dean of the Faculty of Computer Science and Information Technology, the Director of the Institute of Applied Computer Systems, and the head of the Department of System Theory and Design. His research interests are agent technologies, knowledge engineering and management, structural modeling for diagnostics of complex systems and development of intelligent tutoring systems.

He is a member of Institute of Electrical and Electronics Engineers (IEEE), Association of Computer Machinery (ACM) and International Association for Development of the Information Society (IADIS). He is a full member of Latvian Academy of Science.

Aleksis Liekna, Jānis Grundspeņķis. Daudzaģentu sistēmu ziņojumu apmaiņas mehānisma realizācija, pielietojot aspektorientētu pieeju
Rakstā ir piedāvāts veids, kā realizēt daudzaģentu sistēmas ziņojumu apmaiņas mehānismu, pielietojot aspektorientētu pieeju. Lai izstrādātu uzturamu un labi strukturētu daudzaģentu sistēmu, ir svarīgi savstarpēji nodalīt dažādas daudzaģentu sistēmas komponentes (tādas kā ziņojumu apmaiņa, apmācība, kustība, u.tml.). Tā kā aģentu komunikācija daudzaģentu sistēmās notiek ar ziņojumu apmaiņu, tad tai ir jāpievērš īpaša uzmanība. Programmatūras aģentiem ziņojumu apmaiņa var kalpot par vienīgo sensoru un izpildmehānismu. Ziņojumu apmaiņa ir jānodala no pārējām aģentu komponentēm, lai palielinātu to savstarpējo neatkarību, tādējādi uzlabojot sistēmas kopējo struktūru un palielinot tās attīstības potenciālu (vieglāk ir attīstīt un modificēt strukturētu un modulāru nevis monolītu sistēmu). Daudzaģentu sistēmu izstrādē plaši tiek pielietotas objektorientētas tehnoloģijas, taču ar to ir par maz, lai viennozīmīgi atdalītu savstarpēji sadarbojošās komponentes. Piemēram, ja ziņojumu apmaiņu izmanto aģenta kustības un apmācības komponentēs, tad tajās ir jāiekļauj interfeisa realizācija ar ziņojumu apmaiņu. Līdz ar to rodas problēma – kustība un apmācība kļūst atkarīgas no ziņojumu apmaiņas, un, veicot izmaiņas ziņojumu apmaiņā, pastāv iespēja, ka izmaiņas būs jāveic arī kustības un apmācības komponenšu realizācijā. Šo problēmu var risināt, pielietojot aspektorientētu pieeju. Lai gan pētījumi šajā virzienā jau ir veikti, pēc raksta autoru domām tie nesniedz pietiekoši labu risinājumu. Šajā rakstā ir piedāvāta pieeja, kura risina komponenšu savstarpējās neatkarības problēmu, pielietojot aspektu interfeisus, aspektu notikumu uztvērējus un vienotu datu struktūru. Lai ilustrētu piedāvātās pieejas praktisku pielietojumu, ir izstrādāts un rakstā aprakstīts daudzaģentu sistēmas prototips, kas balstās uz JADE platformu. Aspektorientācijas realizācijai ir izmantots AspectJ. Piedāvātā pieeja sekmīgi risina aģenta komponenšu savstarpējās atdalīšanas problēmu ziņojumu apmaiņas gadījumā, taču to var pielietot arī visu pārējo komponentu savstarpējai atdalīšanai, kas ir viens no turpmāko pētījumu mērķiem. Vēl viens turpmāko pētījumu mērķis ir rīka izstrādāšana, kas atvieglotu piedāvātās pieejas pielietošanu.

Алексис Лиекна, Янис Грундспенькис. Реализация механизма обмена сообщениями многоагентной системы с использованием аспектно-ориентированного подхода
В данной статье предложен способ реализации механизма обмена сообщениями в многоагентной системе с использованием аспектно-ориентированного подхода. Для того, чтобы разработать поддерживаемую и хорошо структурированную многоагентную систему, важно разделить

разные компоненты (такие как обмен сообщениями, обучение, движение и другие). Так как коммуникация агентов происходит с помощью сообщений, отдельное внимание надо уделять именно передаче сообщений. Программным агентам передача сообщений может служить единственным сенсором и механизмом выполнения. Передачу сообщений следует отделить от других частей и компонент, чтобы повысить независимость частей друг от друга, таким образом улучшая общую структуру системы и повышая потенциал её развития (проще развивать и модифицировать структурированную и модулярную систему). В разработке многоагентной системы широко используются объектно-ориентированные технологии, но этого недостаточно, чтобы однозначно отделить вместе работающие компоненты. Допустим, если для передачи сообщений используются компоненты передвижения и обучения, то в них надо включить реализацию интерфейса передачи сообщений. Таким образом, появляется проблема - движение и обучение становятся зависимыми от передачи сообщения и, при изменениях в передаче сообщений, возможно, надо будет также вносить изменения в реализацию компонентов движения и обучения. Эту проблему можно решить с помощью применения аспектно-ориентированного подхода. Хотя исследования в этом направлении уже ведутся, авторы статьи считают, что это не дает достаточно хорошего решения. В статье предложен подход, который решает проблему независимости компонентов друг от друга, используя интерфейсы аспектов, приемники событий аспектов и единую структуру данных. Для иллюстрации предложенного подхода разработан и описан многоагентный прототип системы, который базируется на платформе JADE. Реализация аспектно-ориентированных частей выполнена на AspectJ. Предложенное решение успешно решает проблему отделения различных частей агента в случае передачи сообщений, однако его можно использовать также и для отделения всех остальных компонентов, что является одной из целей дальнейших исследования. Ещё одна цель будущих исследований – разработка программного обеспечения для облегчения внедрения данного подхода.