SMART RESOURCE PROTOTYPE ENVIRONMENT V. 2.0

DELIVERABLE 2.3

Technical report

SmartResource: Proactive Self-Maintained Resources in Semantic Web

12/13/2005

University of Jyväskylä

Agora Center

Author: Industrial Ontologies Group

Contact Information: e-mail: vagan@it.jyu.fi

Title: SmartResource prototype environment, v. 2.0, Proactivity Stage, Deliverable 2.3

Work: Technical report

Status of document: final

Number of Pages: 28

Keywords: Java Agent Development Framework, Agent Behavior, Proactivity of Resources, Agent Implementation, Agent Communication, Adapters, Java Naming and Directory Interface, Prototype environment

.

Abbreviations

JADE	Java Agent Development Framework
FIPA	The Foundation for Intelligent Physical Agents
EJB	Enterprise Java Beans
JNDI	Java Naming and Directory Interface
ACL	Agent Communication Language

Contents

1 Introduction

Reported deliverable D2.3 belongs to Proactivity Stage [1] of SmartResource project [2] and focuses on an architectural design of agent-based resource management framework and on enabling a meaningful resource interaction. Its research and development tasks include adding software agents (Maintenance Agents) to the industrial resources, enabling their proactive behavior. For this purpose, Resource Goal/Behavior Description Framework has been designed, which is a basis for making resource's individual behavioral model. The model is assumed to be processed and executed by the RGBDF engine [3] used by the Maintenance Agents. Agent-based approach for management of various complex processes in the decentralized environments is being adopted and popularized currently in many industrial applications. Presentation of the resources as agents in the multi-agent system and use of technologies and standards developed by the Agent research community is a prospective way of industrial systems development. Creation of framework for enabling resources' proactive behavior and such agent features as self-interestedness, goal-oriented behavior, ability to reason about itself and its environment and to communicate with other agents, can bring a value to the next-generation industrial systems.

According to SmartResource's project implementation plan, D2.3 is meant to automate the scenario of interaction between Device, Expert and Web Service that was implemented in the SmartResource prototype environment v. 1.0 (Adaptation Stage) [4]. The logic of interaction has to be implemented in a multi-agent system involving DeviceAgent, ExpertAgent and WebServiceAgent respectively. This implementation is a practical part of the previous second project year deliverables: Resource Goal/Behavior Description Framework (D2.1) [5] and Design of the SmartResource Platform (D2.2) [6].

2 JADE – a platform for the SmartResource agent scenario

2.1 Choice of a multi-agent system

As a basis for implementation of the interaction scenario between SmartResource agents (see task of this deliverable) Java Agent Development Framework (JADE)[1] has been chosen. Such choice is made, because Java language is the basis for JADE that makes its integration with previous version of the SmartResource Prototype Environment easy. Additionally, the JADE platform is mature in providing a variety of tools for the debugging and deployment phases of the agents. JADE fully follows FIPA[2] specifications, which are important for further ontological description of multi-agent coordination.

In general, the implementation task assumes migration of the scenario's logics from the Control Servlet to the community of agents implemented in JADE (see Figure 1).

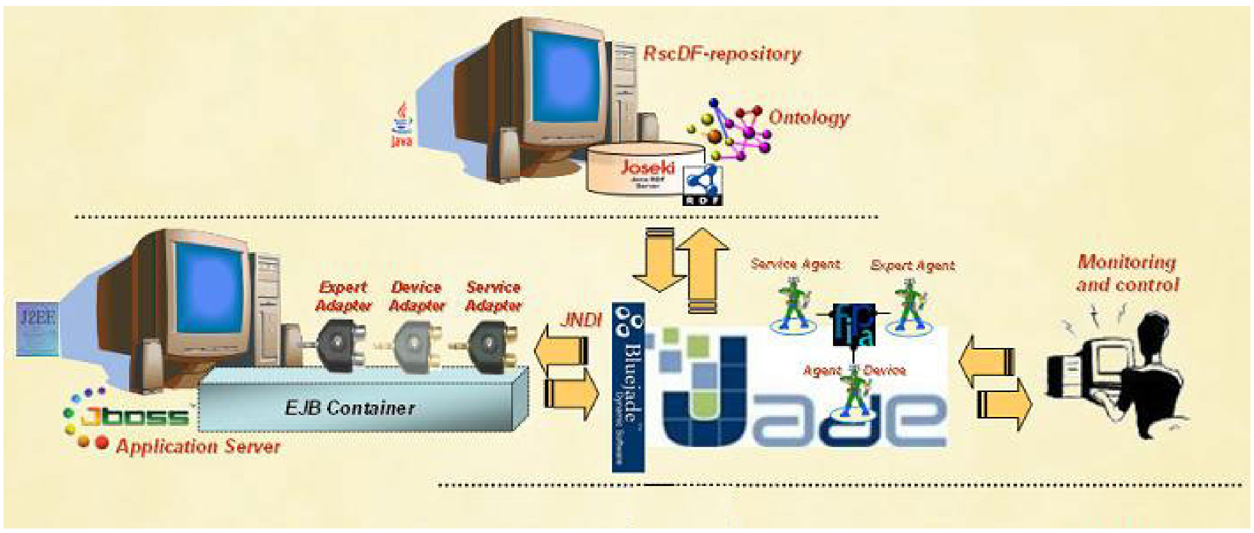


Figure 1 - Evolution of the SmartResource Prototype Environment

On the other hand, adapters that were implemented during previous project year, reside at the JBoss Application Server as they are. It is one of the challenges to implement the access of agents hosted by JADE to the adapters.

As it is shown in the Figure 2, the scenario of interaction between agents includes the following stages:

---

[1] http://jade.tilab.com/
[2] http://www.fipa.org/

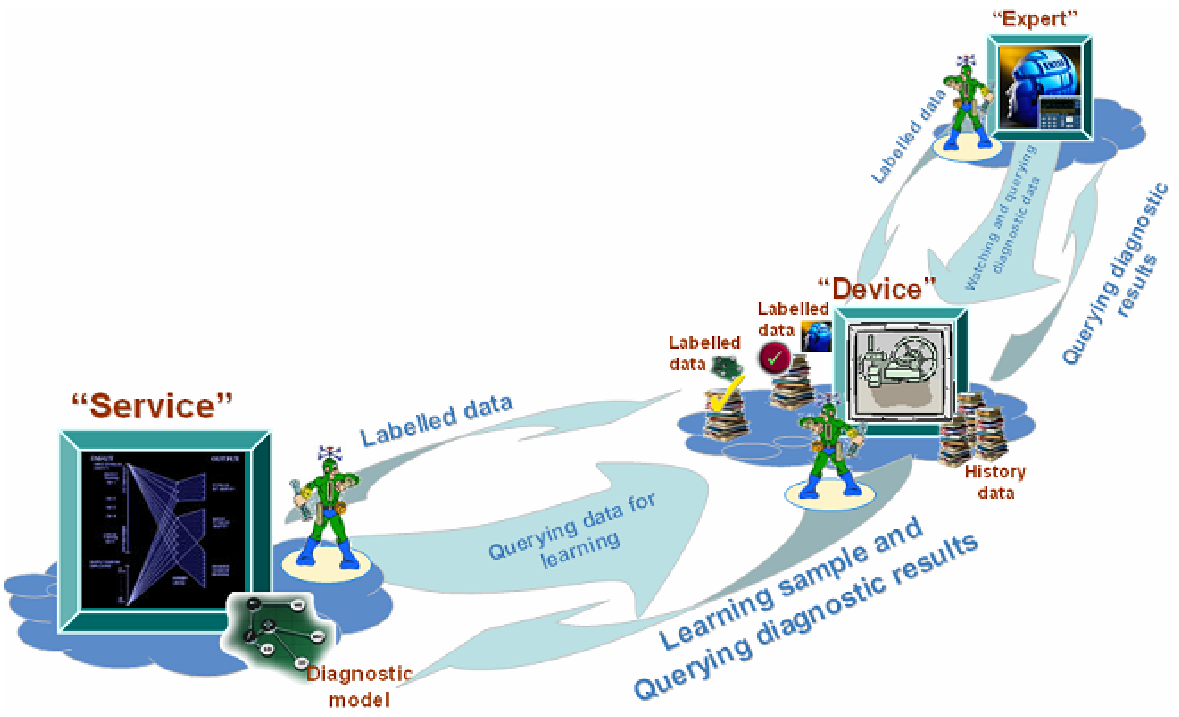


Figure 2 - Scenario of interaction between agents

1. Accumulation of a history of the industrial machine.

2. Diagnostic request processing and response generation by ExpertAgent.

3. Learning of the WebServiceAgent based on labeled data received from ExpertAgent.

4. Diagnostics of alarm situations by WebServiceAgent.

2.2 Access to adapters

As it was planned, the implemented agents access the adapters for data transformation needs. For this purpose, an abstract class ResourceAgent has been designed. It implements the initialization of local history storage of an agent from common history stored at the Joseki server. Additionally the class makes necessary preparations for a successful lookup of the adapters by agents: an instance of a context (JNDI naming directory) that allows for adapters (implemented as EJBs) to be found by their names. See appropriate code below.

```
protected InitialContext getContext(String providerURL) throws NamingException
    Hashtable props = new Hashtable();
    props.put(InitialContext.INITIAL_CONTEXT_FACTORY,
            "org.jnp.interfaces.NamingContextFactory");
    props.put(InitialContext.PROVIDER_URL, providerURL);
    InitialContext initialContext = new InitialContext(props);
    return initialContext;
}

/**
 *
 */
protected synchronized void initHistory() {
    resourceHistory = ModelFactory.createDefaultModel();
    resourceHistory.read(ONTOLOGY_URL);
}
```

The hierarchy of Agent classes implemented in second version of the SmartResource Prototype environment is given in Figure 3.

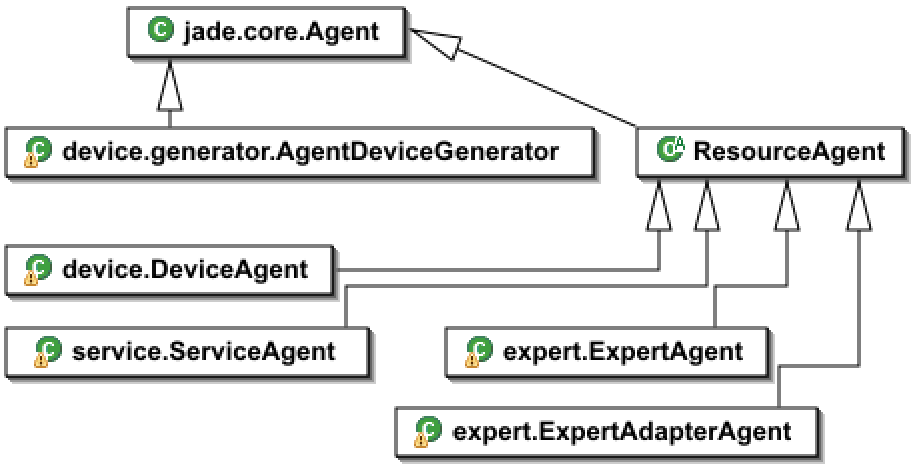


Figure 3 - Hierarchy of agent Java classes

2.3 Behavior of agents in JADE

So far, developers of JADE have provided a possibility to implement behaviors of agents using the hierarchy of classes shown in Figure 4. This structured approach to modeling behaviors makes JADE platform even more suitable for experimental research of the RGBDF schema and RGBDF engine.

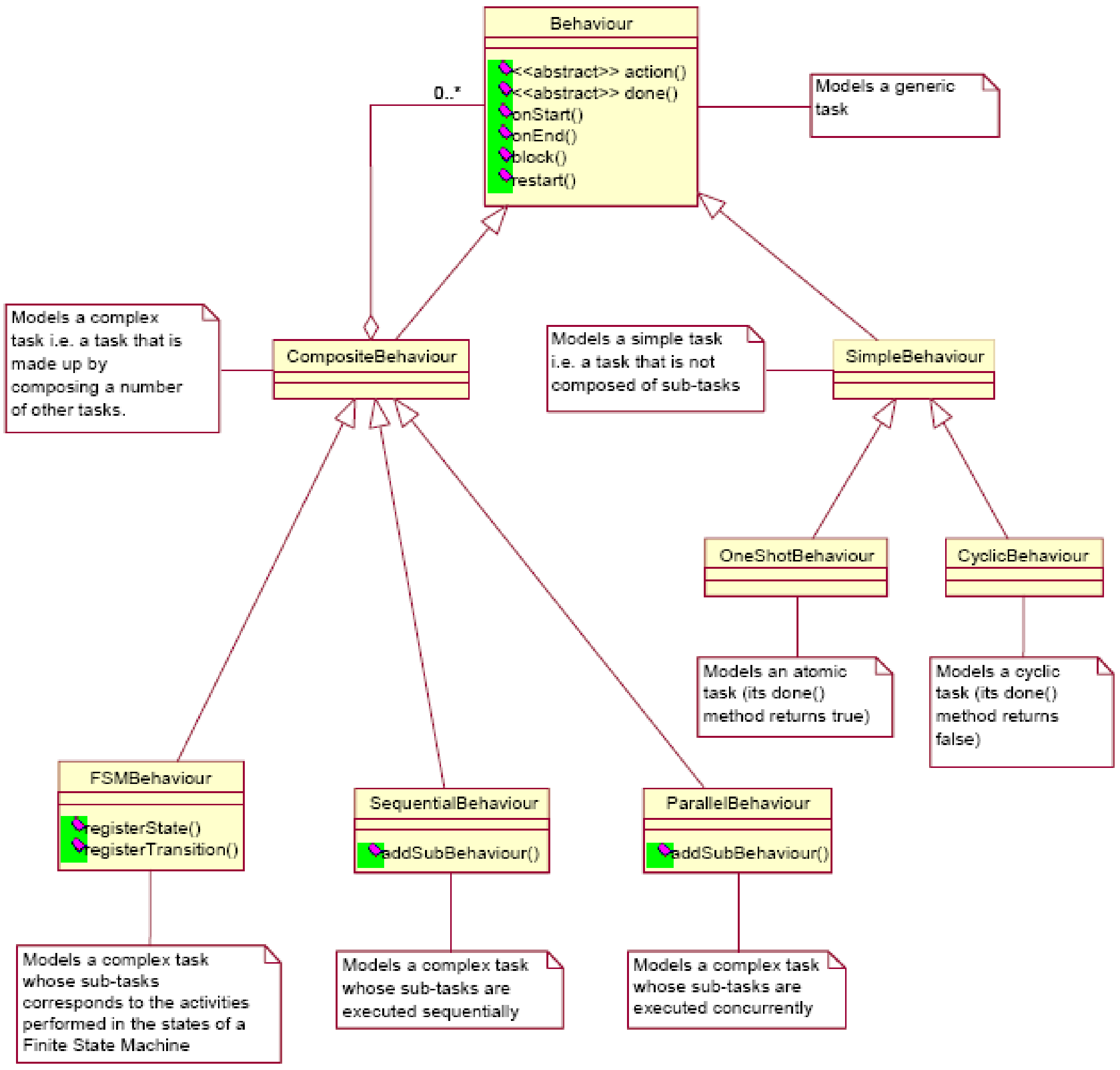

Behaviour
<<abstract>> action()
<<abstract>> done()
onStart()
onEnd()
block()
restart()
0..*
Models a generic task
Models a complex task i.e. a task that is made up by composing a number of other tasks.
CompositeBehaviour
Models a simple task i.e. a task that is not composed of sub-tasks
SimpleBehaviour
OneShotBehaviour
CyclicBehaviour
Models an atomic task (its done() method returns true)
Models a cyclic task (its done() method returns false)
FSMBehaviour
registerState()
registerTransition()
SequentialBehaviour
addSubBehaviour()
ParallelBehaviour
addSubBehaviour()
Models a complex task whose sub-tasks corresponds to the activities performed in the states of a Finite State Machine
Models a complex task whose sub-tasks are executed sequentially
Models a complex task whose sub-tasks are executed concurrently

Figure 4 - Hierarchy of different behaviors of agents in JADE

3 Implementation of the scenario in JADE

The implemented classes have been distributed among the following packages of previous version of the SmartResource prototype environment:

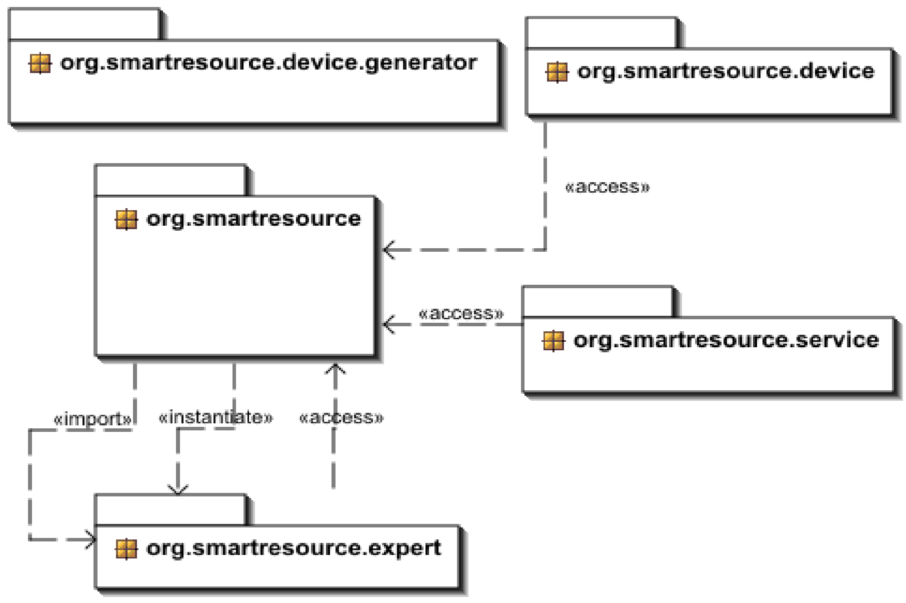


Figure 5 - Map of Java packages in the implementation

The package org.smartresource includes abstract class ResourceAgent mentioned above. As it is shown in Figure 5, packages org.smartresource.device, org.smartresource.service and org.smartresource.expert use classes implemented in org.smartresource. The packages contain main classes: DeviceAgent, ServiceAgent and ExpertAgent respectively (Figure 6).

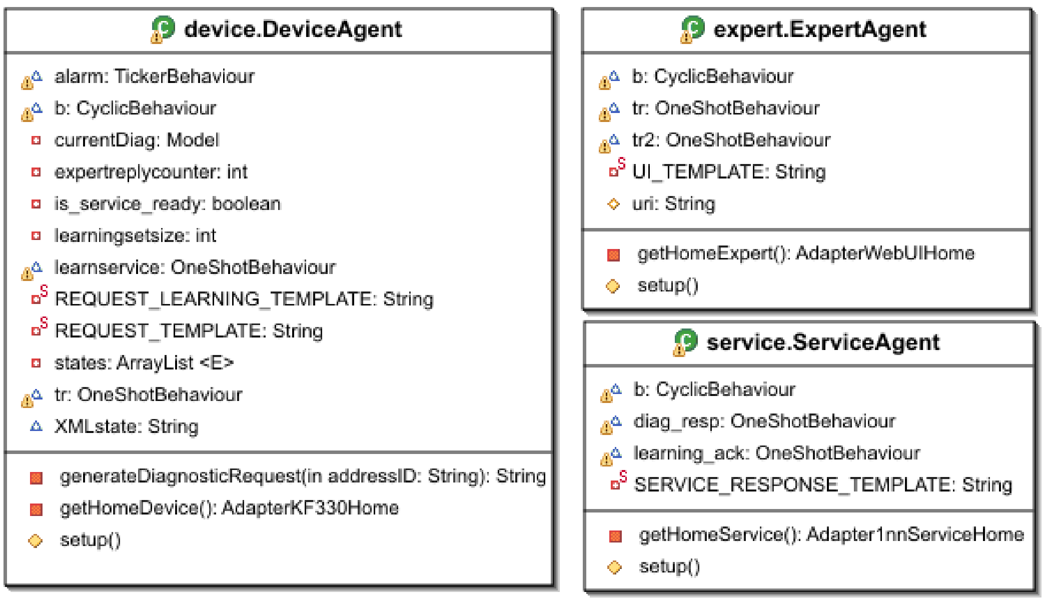
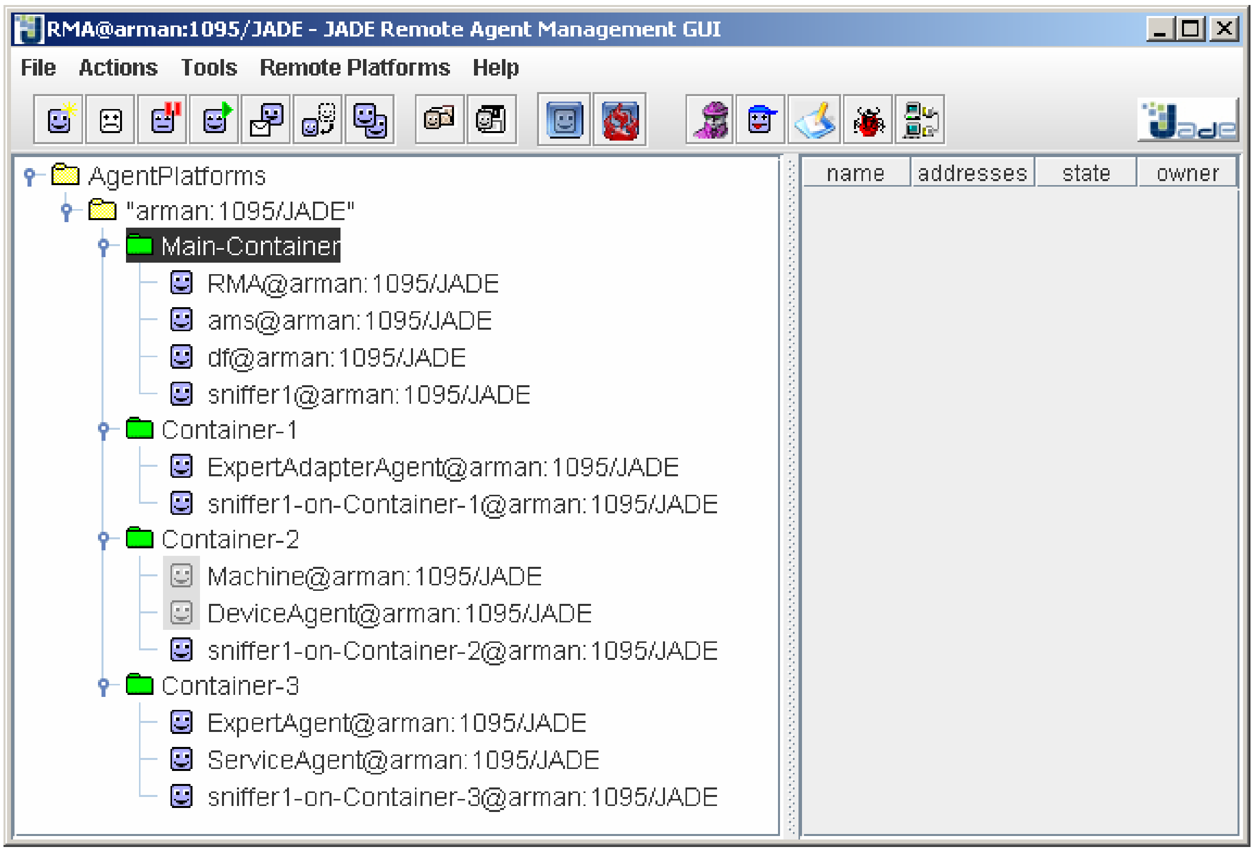


Figure 6 – General view of the DeviceAgent and ExpertAgent classes

3.1 Platform launch

In order to succeed with the interaction of the agents on the platform, we have to start all the platform components in a predefined order. First the Joseki server must be started because every agent initializes an appropriate adapter, and requests for ontology from joseki storage. Next we start JADE platform as such without agents on it. As far as resource adapters are implemented mainly as EJB's, we start the JBoss server with adapters. During the JBoss initialization, an ExpertAdapterAgent (helper agent, which is a part of expert adapter, see subchapter 3.4 for details) is created and deployed to JADE. Then we deploy Service and Expert agents, which are ready to accept incoming request messages. Now the AgentDeviceGenerator and DeviceAgent can be started. These two agents constitute the initial point of the platform operation, as far as they originally generate messages, which go to expert and service agents. When all the agents are running, the configuration of JADE is the following:



```
AgentPlatforms
  "arman:1095/JADE"
    Main-Container
      RMA@arman:1095/JADE
      ams@arman:1095/JADE
      df@arman:1095/JADE
      sniffer1@arman:1095/JADE
    Container-1
      ExpertAdapterAgent@arman:1095/JADE
      sniffer1-on-Container-1@arman:1095/JADE
    Container-2
      Machine@arman:1095/JADE
      DeviceAgent@arman:1095/JADE
      sniffer1-on-Container-2@arman:1095/JADE
    Container-3
      ExpertAgent@arman:1095/JADE
      ServiceAgent@arman:1095/JADE
      sniffer1-on-Container-3@arman:1095/JADE
```

Figure 7 – General view of the JADE platform

The number of containers shows that agents are started from different places such as Java code in a servlet of the JBoss server or a command line.

3.2 AgentDeviceGenerator (Machine)

This agent simulates industrial device generating states of the industrial device in XML format. The frequency of generation is stable with a period of 10 seconds, thus every 10 seconds the agent sends INFORM ACL message to DeviceAgent with the XML state in its content.

See Figure 7 that illustrates the message flow between agents. The monitoring of the messages was performed using SnifferAgent of JADE.

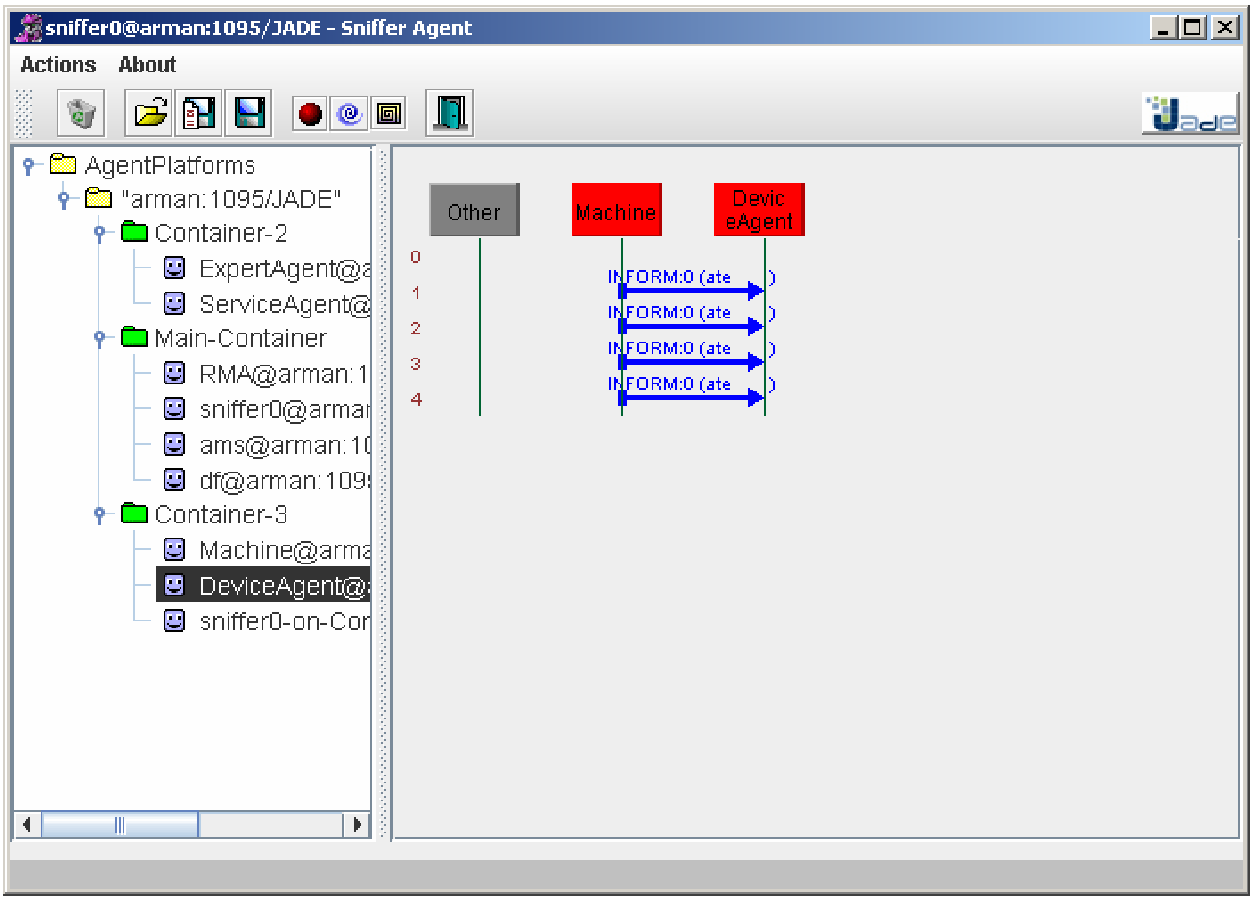
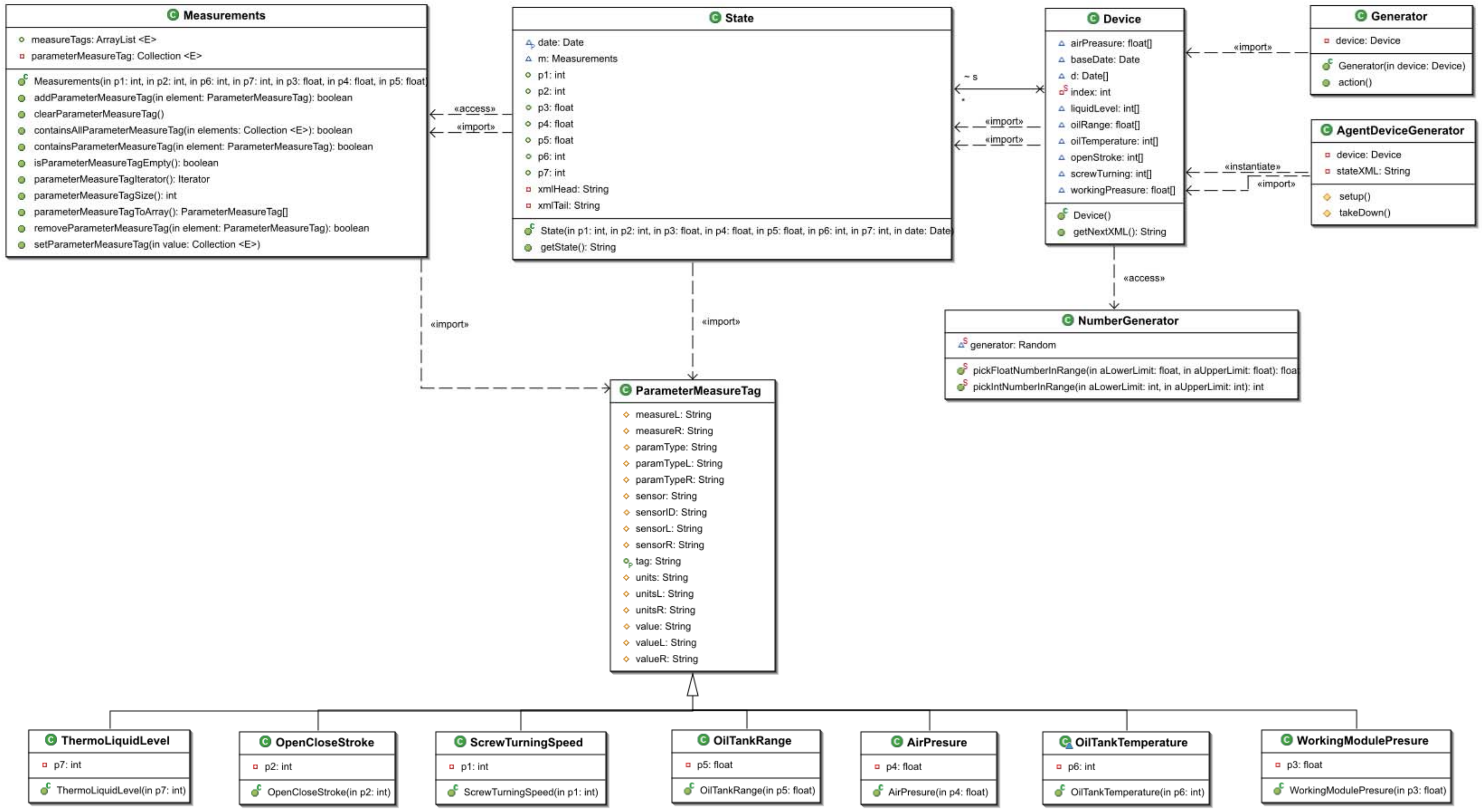


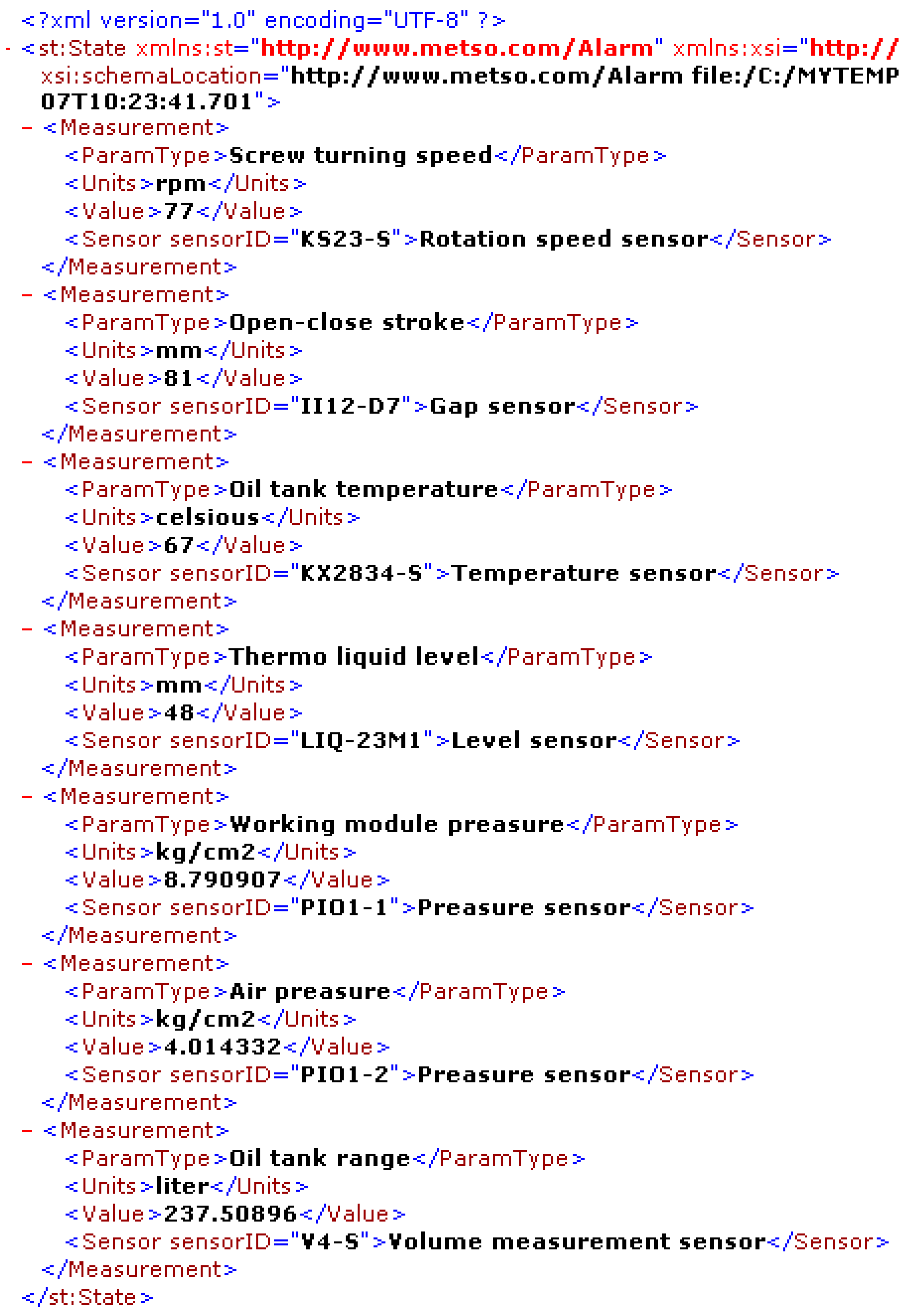
Figure 7 - Message flow from AgentDeviceGenerator to DeviceAgent

AgentDeviceGenerator instantiates the Device class of the org.smartresource.device.generator package developed in a previous version of the prototype (see Figure 8).

Figure 8 – *Generator* package: inheritance, dependence and associataion UML diagram


Measurements
measureTags: ArrayList <E>
parameterMeasureTag: Collection <E>
Measurements(in p1: int, in p2: int, in p6: int, in p7: int, in p3: float, in p4: float, in p5: float
addParameterMeasureTag(in element: ParameterMeasureTag): boolean
clearParameterMeasureTag()
containsAllParameterMeasureTag(in elements: Collection <E>): boolean
containsParameterMeasureTag(in element: ParameterMeasureTag): boolean
isParameterMeasureTagEmpty(): boolean
parameterMeasureTagIterator(): Iterator
parameterMeasureTagSize(): int
parameterMeasureTagToArray(): ParameterMeasureTag[]
removeParameterMeasureTag(in element: ParameterMeasureTag): boolean
setParameterMeasureTag(in value: Collection <E>)
State
date: Date
m: Measurements
p1: int
p2: int
p3: float
p4: float
p5: float
p6: int
p7: int
xmlHead: String
xmlTail: String
State(in p1: int, in p2: int, in p3: float, in p4: float, in p5: float, in p6: int, in p7: int, in date: Date)
getState(): String
Device
airPreasure: float[]
baseDate: Date
d: Date[]
index: int
liquidLevel: int[]
oilRange: float[]
oilTemperature: int[]
openStroke: int[]
screwTurning: int[]
workingPreasure: float[]
Device()
getNextXML(): String
Generator
device: Device
Generator(in device: Device)
action()
AgentDeviceGenerator
device: Device
stateXML: String
setup()
takeDown()
NumberGenerator
generator: Random
pickFloatNumberInRange(in aLowerLimit: float, in aUpperLimit: float): float
pickIntNumberInRange(in aLowerLimit: int, in aUpperLimit: int): int
«access»
«import»
~ s
*
«import»
«import»
«import»
«instantiate»
«import»
«access»
«import»
«import»
ParameterMeasureTag
measureL: String
measureR: String
paramType: String
paramTypeL: String
paramTypeR: String
sensor: String
sensorID: String
sensorL: String
sensorR: String
tag: String
units: String
unitsL: String
unitsR: String
value: String
valueL: String
valueR: String
ThermoLiquidLevel
p7: int
ThermoLiquidLevel(in p7: int)
OpenCloseStroke
p2: int
OpenCloseStroke(in p2: int)
ScrewTurningSpeed
p1: int
ScrewTurningSpeed(in p1: int)
OilTankRange
p5: float
OilTankRange(in p5: float)
AirPresure
p4: float
AirPresure(in p4: float)
OilTankTemperature
p6: int
OilTankTemperature(in p6: int)
WorkingModulePresure
p3: float
WorkingModulePresure(in p3: float)

Below is a sample of the XML message that is sent.

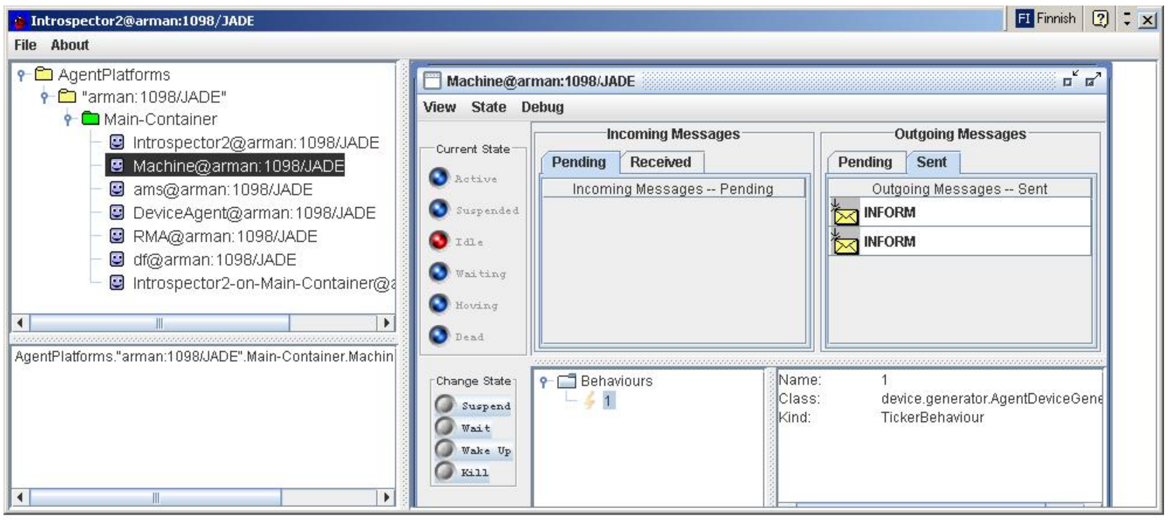


```
  <?xml version="1.0" encoding="UTF-8" ?>
- <st:State xmlns:st="http://www.metso.com/Alarm" xmlns:xsi="http://
    xsi:schemaLocation="http://www.metso.com/Alarm file:/C:/MYTEMP
    07T10:23:41.701">
  - <Measurement>
      <ParamType>Screw turning speed</ParamType>
      <Units>rpm</Units>
      <Value>77</Value>
      <Sensor sensorID="KS23-S">Rotation speed sensor</Sensor>
    </Measurement>
  - <Measurement>
      <ParamType>Open-close stroke</ParamType>
      <Units>mm</Units>
      <Value>81</Value>
      <Sensor sensorID="II12-D7">Gap sensor</Sensor>
    </Measurement>
  - <Measurement>
      <ParamType>Oil tank temperature</ParamType>
      <Units>celsious</Units>
      <Value>67</Value>
      <Sensor sensorID="KX2834-S">Temperature sensor</Sensor>
    </Measurement>
  - <Measurement>
      <ParamType>Thermo liquid level</ParamType>
      <Units>mm</Units>
      <Value>48</Value>
      <Sensor sensorID="LIQ-23M1">Level sensor</Sensor>
    </Measurement>
  - <Measurement>
      <ParamType>Working module preasure</ParamType>
      <Units>kg/cm2</Units>
      <Value>8.790907</Value>
      <Sensor sensorID="PI01-1">Preasure sensor</Sensor>
    </Measurement>
  - <Measurement>
      <ParamType>Air preasure</ParamType>
      <Units>kg/cm2</Units>
      <Value>4.014332</Value>
      <Sensor sensorID="PI01-2">Preasure sensor</Sensor>
    </Measurement>
  - <Measurement>
      <ParamType>Oil tank range</ParamType>
      <Units>liter</Units>
      <Value>237.50896</Value>
      <Sensor sensorID="V4-S">Volume measurement sensor</Sensor>
    </Measurement>
  </st:State>
```

The functionality of the AgentDeviceGenerator is implemented as a TickerBehaviour of JADE. In the fragment of the corresponding code below we can see that frequency of “ticking” is set to 10000 milliseconds. Device class is used inside the implementation of this behavior for generation of the XML messages.

```
protected void setup() {
    TickerBehaviour b = new TickerBehaviour(this,10000){
        protected void onTick(){
            stateXML = device.getNextXML();
            ACLMessage inform = new ACLMessage(ACLMessage.INFORM);
            inform.setLanguage("XML");
            inform.addReceiver(new AID("DeviceAgent",AID.ISLOCALNAME));
            inform.setContent(stateXML);
            inform.setConversationId("DeviceState");
            myAgent.send(inform);
        }
    };
    addBehaviour(b);

}
```

Execution of the behavior by the agent was monitored by the Introspector utility agent in JADE (see Figure 9). Additionally the Introspector agent allows for monitoring incoming and outgoing messages for agents hosted by the JADE platform.



AgentPlatforms
"arman:1098/JADE"
Main-Container
Introspector2@arman:1098/JADE
Machine@arman:1098/JADE
ams@arman:1098/JADE
DeviceAgent@arman:1098/JADE
RMA@arman:1098/JADE
df@arman:1098/JADE
Introspector2-on-Main-Container@a

AgentPlatforms."arman:1098/JADE".Main-Container.Machin

Incoming Messages -- Pending

Outgoing Messages -- Sent
INFORM
INFORM

Behaviours
1

Name: 1
Class: device.generator.AgentDeviceGene
Kind: TickerBehaviour

Figure 8 - Monitoring behavior of AgentDeviceGenerator in JADE

3.3 DeviceAgent

This agent receives messages with states in XML format from AgentDeviceGenerator, uses DeviceAdapter implemented earlier for transformation of the XML message into RscDF representation. The state in the RscDF format is stored in the local history of the agent and further every minute the DeviceAgent generates alarm message to ExpertAgent.

The messages sent from DeviceAgent to ExpertAgent (ACL REQUEST) can be shown using SnifferAgent in JADE (see Figure 10).

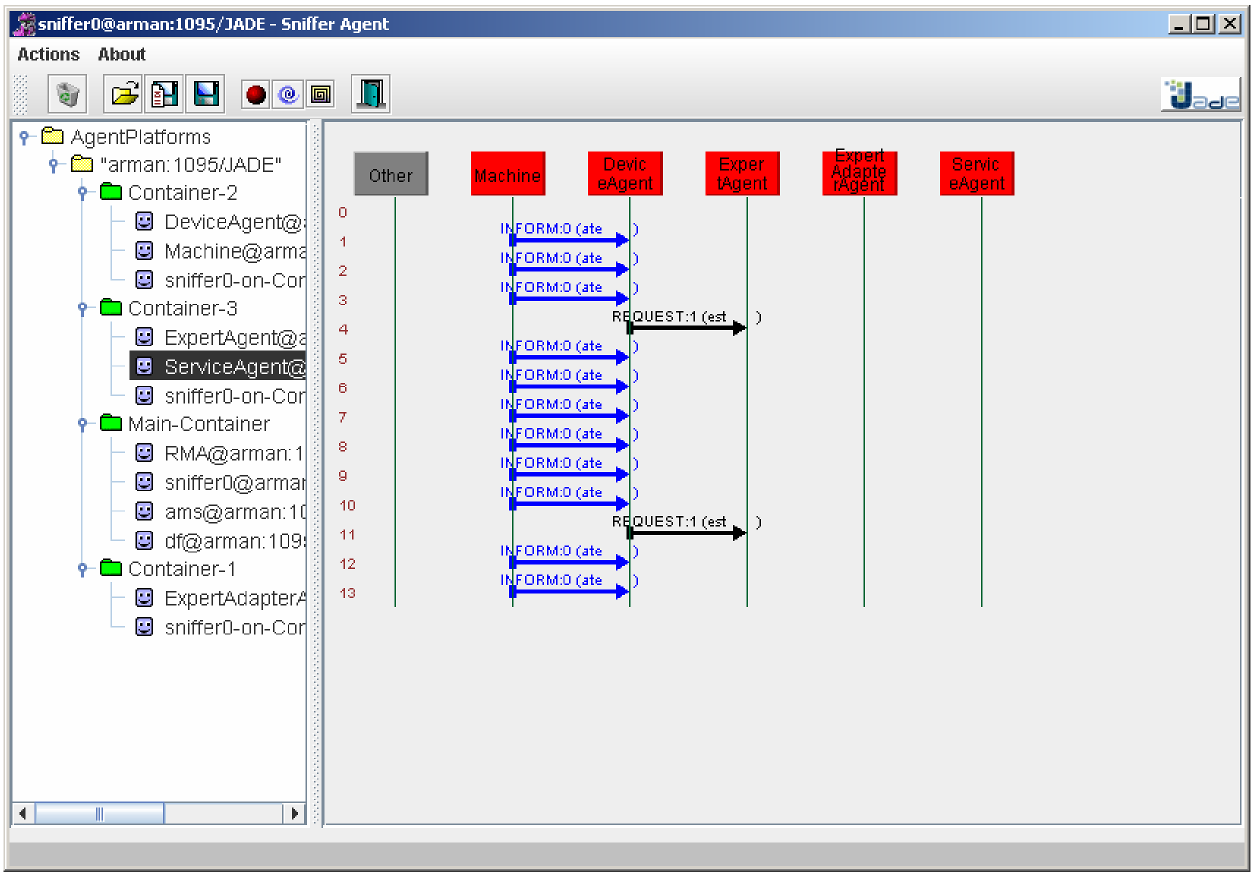


Figure 9 - Monitoring of the messages from DeviceAgent to ExpertAgent

Details of the message can be viewed directly from the SnifferAgent by two clicks (see Figure 12). The message is a request for diagnostics in RscDF format. It contains all the history collected by DeviceAgent so far with the AlarmRequest composed from a template.

The behavior of the DeviceAgent is not such simple as the behavior of the AgentDeviceGenerator. It consists of the composition of three behaviors: CyclicBehavior, OneShotBehavior and TickerBehavior (you can find them in Figure 4). They are composed in the following way (Figure 11):

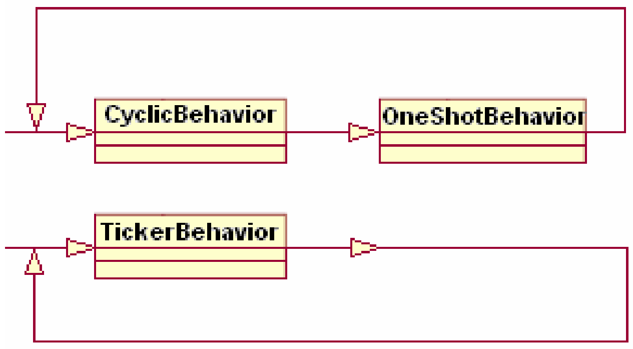
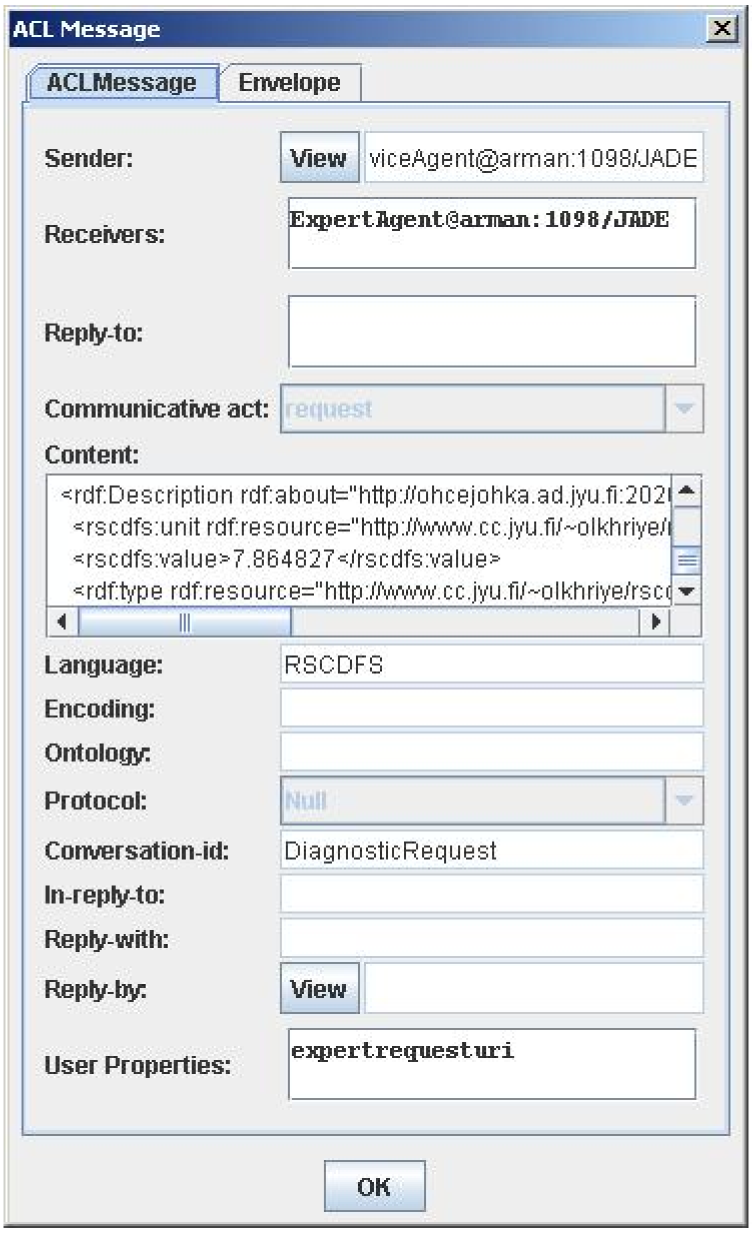


Figure 10 - Behaviors that compose the logics of the DeviceAgent

There are two threads of behavior of the agent: communication with AgentDeviceGenerator and communication with Expert and Service agents.



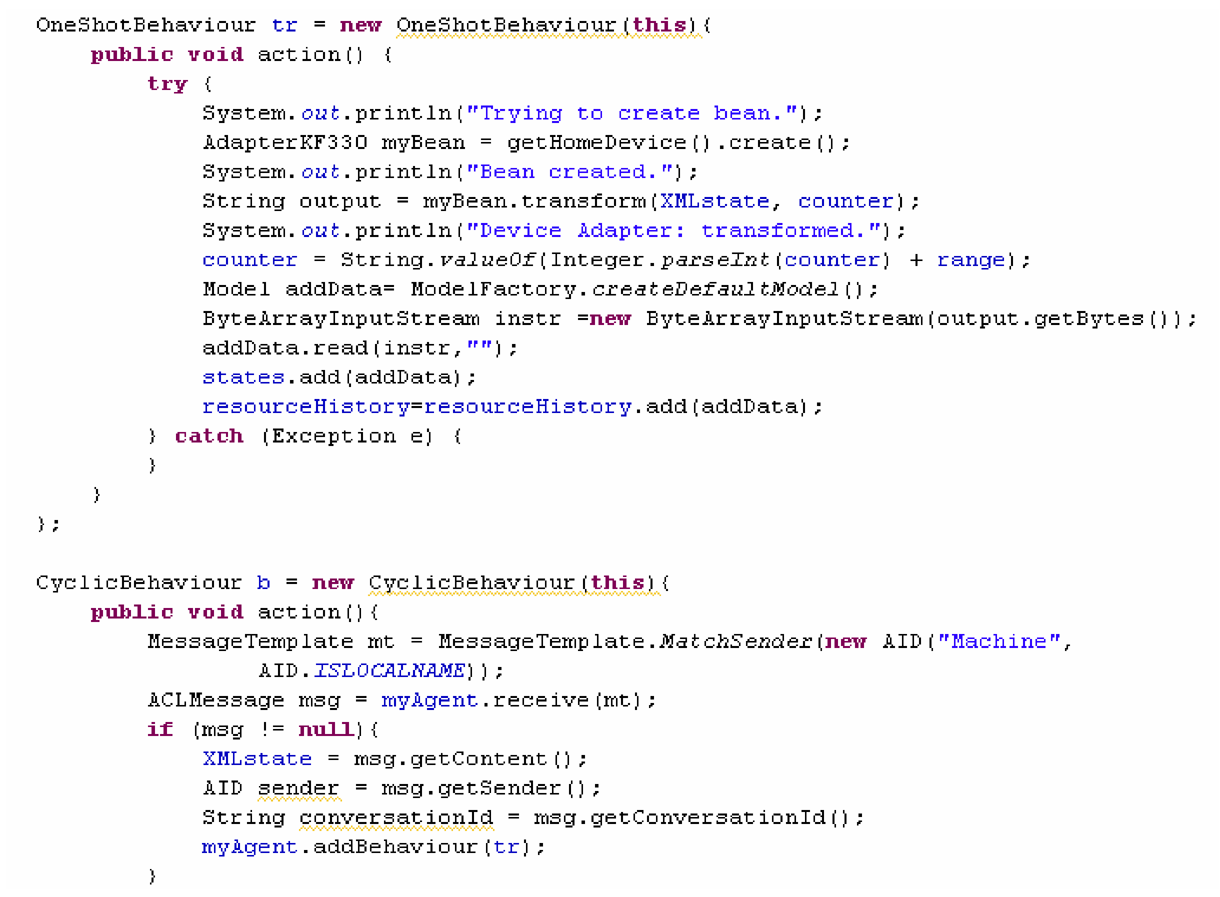
Sender: viceAgent@arman:1098/JADE
Receivers: ExpertAgent@arman:1098/JADE
Reply-to:
Communicative act: request
Content:

```
<rdf:Description rdf:about="http://ohcejohka.ad.jyu.fi:202
  <rscdfs:unit rdf:resource="http://www.cc.jyu.fi/~olkhriye/
  <rscdfs:value>7.864827</rscdfs:value>
  <rdf:type rdf:resource="http://www.cc.jyu.fi/~olkhriye/rsc
```

Language: RSCDFS
Encoding:
Ontology:
Protocol: Null
Conversation-id: DiagnosticRequest
In-reply-to:
Reply-with:
Reply-by:
User Properties: expertrequesturi

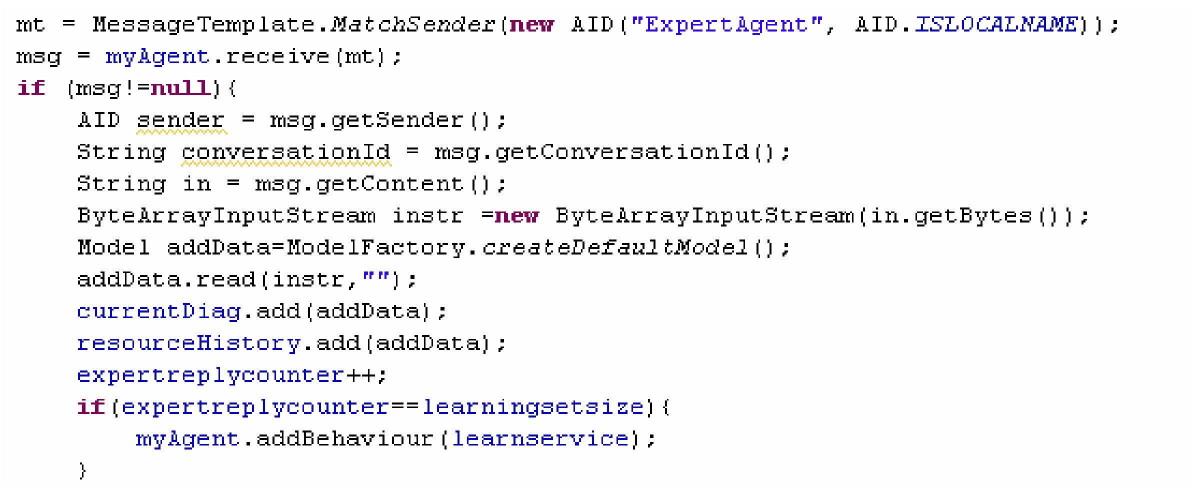
Figure 11 - Message-request from DeviceAgent to ExpertAgent

The cyclic behavior listens to incoming messages from other agents. When it receives the message with a state in XML format from the AgentDeviceGenerator, the agent invokes transformation methods of the adapter and stores the state in the RscDF format to the local history (see the code below).



```
OneShotBehaviour tr = new OneShotBehaviour(this){
    public void action() {
        try {
            System.out.println("Trying to create bean.");
            AdapterKF330 myBean = getHomeDevice().create();
            System.out.println("Bean created.");
            String output = myBean.transform(XMLstate, counter);
            System.out.println("Device Adapter: transformed.");
            counter = String.valueOf(Integer.parseInt(counter) + range);
            Model addData= ModelFactory.createDefaultModel();
            ByteArrayInputStream instr =new ByteArrayInputStream(output.getBytes());
            addData.read(instr,"");
            states.add(addData);
            resourceHistory=resourceHistory.add(addData);
        } catch (Exception e) {
        }
    }
};

CyclicBehaviour b = new CyclicBehaviour(this){
    public void action(){
        MessageTemplate mt = MessageTemplate.MatchSender(new AID("Machine",
                AID.ISLOCALNAME));
        ACLMessage msg = myAgent.receive(mt);
        if (msg != null){
            XMLstate = msg.getContent();
            AID sender = msg.getSender();
            String conversationId = msg.getConversationId();
            myAgent.addBehaviour(tr);
        }
```

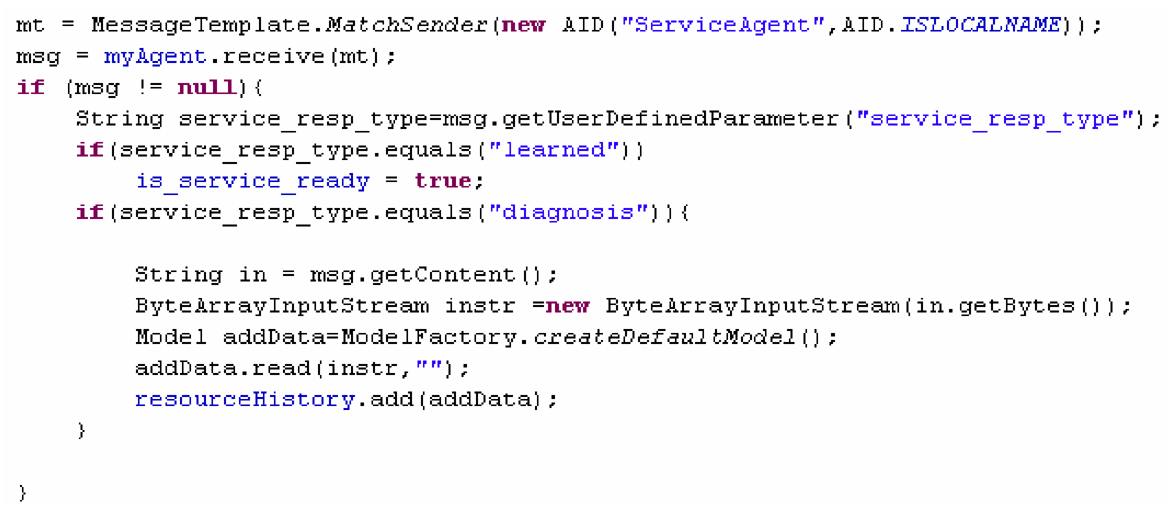
Overall in cyclic behaviour there are 3 processing cases which process incoming messages: one mentioned above, and two procedures for processing messages from ExpertAgent and ServiceAgent respectively. In case of message from Expert Agent the logic is the following:



```
mt = MessageTemplate.MatchSender(new AID("ExpertAgent", AID.ISLOCALNAME));
msg = myAgent.receive(mt);
if (msg!=null){
    AID sender = msg.getSender();
    String conversationId = msg.getConversationId();
    String in = msg.getContent();
    ByteArrayInputStream instr =new ByteArrayInputStream(in.getBytes());
    Model addData=ModelFactory.createDefaultModel();
    addData.read(instr,"");
    currentDiag.add(addData);
    resourceHistory.add(addData);
    expertreplycounter++;
    if(expertreplycounter==learningsetsize){
        myAgent.addBehaviour(learnservice);
    }
```

The agent stores the response from expert to the local storage and increments the counter of the expert replies. After increment it checks whether it is enough learning exaples (i.e. expert replies) for learning a service by comparing the current number of expert replies with the predefined variable *learningsetsize*. If the amount is sufficient for learning, it launches the behavior "*learnservice*".

In case of receiving the message from ServiceAgent, the procedure determines the type of the message received from the ServiceAgent by checking the User Defined Parameter "*service_resp_type*". If the value equals "*learned*", the procedure sets the trigger *is_service_ready* to true. Else if the value equals "*diagnosis*", it stores the diagnosis to the local history. See the code fragment below:



```
mt = MessageTemplate.MatchSender(new AID("ServiceAgent",AID.ISLOCALNAME));
msg = myAgent.receive(mt);
if (msg != null){
    String service_resp_type=msg.getUserDefinedParameter("service_resp_type");
    if(service_resp_type.equals("learned"))
        is_service_ready = true;
    if(service_resp_type.equals("diagnosis")){

        String in = msg.getContent();
        ByteArrayInputStream instr =new ByteArrayInputStream(in.getBytes());
        Model addData=ModelFactory.createDefaultModel();
        addData.read(instr,"");
        resourceHistory.add(addData);
    }

}
```

Ticker behavior implements periodical (once per 60 seconds) requests for diagnostics sent to ExpertAgent or ServiceAgent. It checks for a trigger *is_service_ready* value. While the value is false, all the diagnostic requests are sent to an ExpertAgent, but as soon as service has learned (DeviceAgent has received a confirmation message with the UserDefinedParameter *service_resp_type="learned"*), the trigger value is switched to true and since that moment all the diagnostic requests are sent to the ServiceAgent. The request generation logic includes downloading a template of the request from a server and then filling it by the elements parsed from the local history.

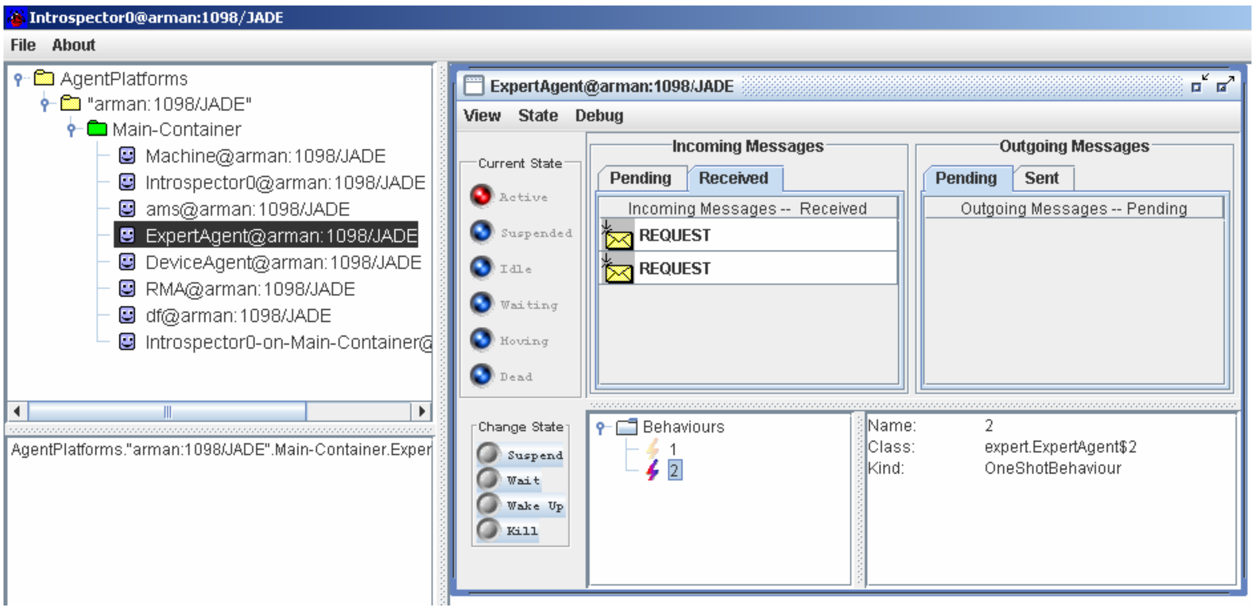
3.4 ExpertAgent

The behavior of the ExpertAgent comprises the receiving message with request for diagnostics from DeviceAgent, then transforming this message to HTML using ExpertAdapter, getting expert's response, transforming to RscDF and sending it back to DeviceAgent. As it was implemented previously, the ExpertAgent sends a request for diagnostics to human expert via e-mail service, which contains link to a diagnostics page. This page is generated on the fly and is published on a web server. When expert opens the diagnostic page and makes a diagnosis, the data of the form is sent to a servlet, which in turn invokes an ExpertAdapterAgent's method to send a message to an ExpertAgent. When ExpertAgent receives a diagnosis from

ExpertAdapterAgent, it transforms it into RscDF using the ExpertAdapter and sends the RscDF to a DeviceAgent

The behavioral implementation of the ExpertAgent is simpler than the DeviceAgents's one. That is, CyclicBehaviour class is used for receiving a request for diagnostics from DeviceAgent, or a message with a diagnosis from ExpertAdapterAgent. After receiving the expected message, the logics inside the CyclicBehavior invoke OneShotBehavior to perform data transformation from RscDF representation to HTML or sending a reply to DeviceAgent.

Figures 13 and 14 show the incoming ACL messages, received from DeviceAgent. When a message from device is received, the OneShotBehaviour is added.



AgentPlatforms
"arman:1098/JADE"
Main-Container
Machine@arman:1098/JADE
Introspector0@arman:1098/JADE
ams@arman:1098/JADE
ExpertAgent@arman:1098/JADE
DeviceAgent@arman:1098/JADE
RMA@arman:1098/JADE
df@arman:1098/JADE
Introspector0-on-Main-Container@

AgentPlatforms."arman:1098/JADE".Main-Container.Exper

Incoming Messages -- Received
REQUEST
REQUEST

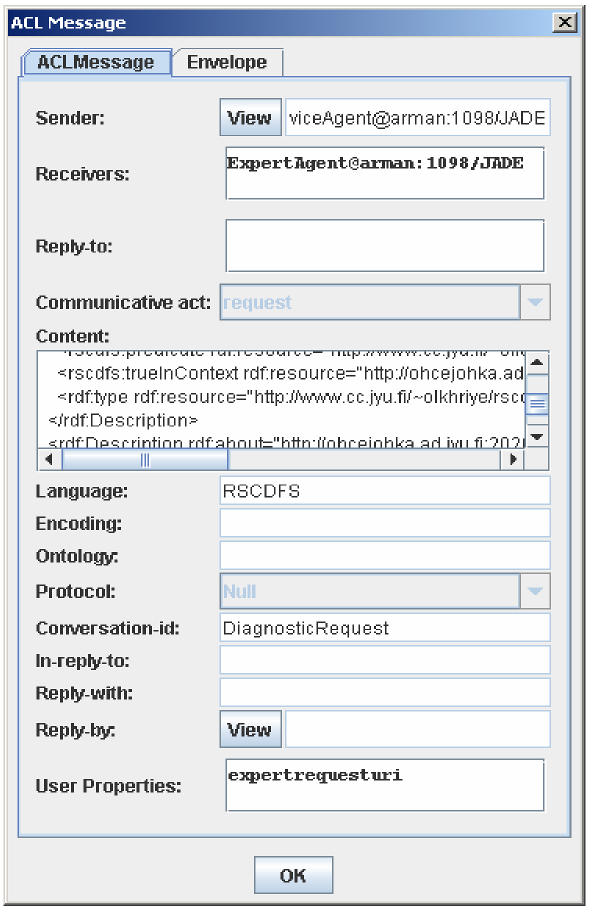
Outgoing Messages -- Pending

Behaviours
1
2

Name: 2
Class: expert.ExpertAgent$2
Kind: OneShotBehaviour

Figure 12 - Monitoring messages and a behavior of the ExpertAgent

Figure 13 shows the moment when OneShotBehavior is active, which means that the message is being transformed by the ExpertAdapter. As you may notice, the message in Figure 14 is the same as the one in Figure 12 that proves the correctness of the implementation.



Sender: viceAgent@arman:1098/JADE

Receivers: ExpertAgent@arman:1098/JADE

Reply-to:

Communicative act: request

Content:

```
  <rscdfs:trueInContext rdf:resource="http://ohcejohka.ad
  <rdf:type rdf:resource="http://www.cc.jyu.fi/~olkhriye/rscd
</rdf:Description>
<rdf:Description rdf:about="http://ohcejohka.ad.jyu.fi:202
```

Language: RSCDFS

Encoding:

Ontology:

Protocol: Null

Conversation-id: DiagnosticRequest

In-reply-to:

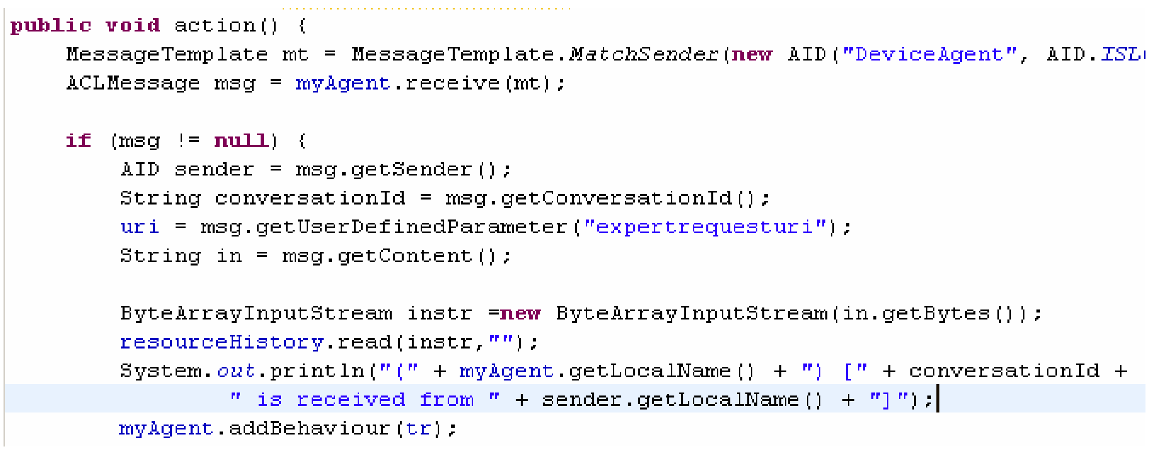
Reply-with:

Reply-by:

User Properties: expertrequesturi

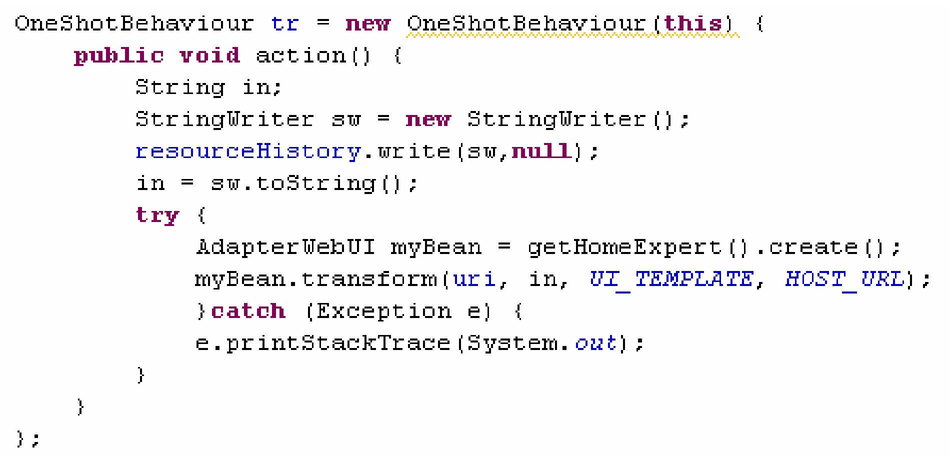
Figure 13 - Details of the ACL message received by ExpertAgent

The code which implements processing logic of the diagnostic request from DeviceAgent is distributed among cyclic and one shot behaviors of the ExpertAgent. The part of the cyclic behaviour is given below.



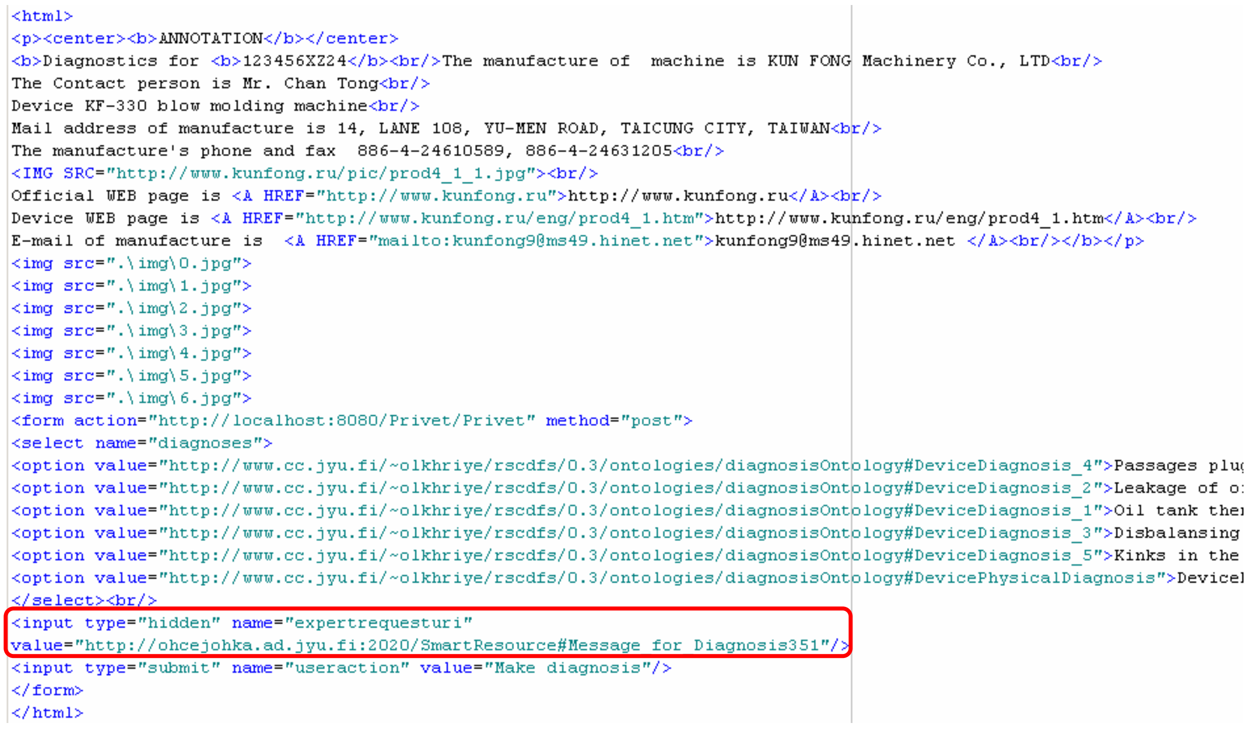
```
public void action() {
    MessageTemplate mt = MessageTemplate.MatchSender(new AID("DeviceAgent", AID.ISL
    ACLMessage msg = myAgent.receive(mt);

    if (msg != null) {
        AID sender = msg.getSender();
        String conversationId = msg.getConversationId();
        uri = msg.getUserDefinedParameter("expertrequesturi");
        String in = msg.getContent();

        ByteArrayInputStream instr =new ByteArrayInputStream(in.getBytes());
        resourceHistory.read(instr,"");
        System.out.println("(" + myAgent.getLocalName() + ") [" + conversationId +
                " is received from " + sender.getLocalName() + "]");
        myAgent.addBehaviour(tr);
```

The last line of the code above adds a new behavior to an ExpertAgent. It is a one shot behavior, which invokes an adapter for transformation from RscDF to HTML.



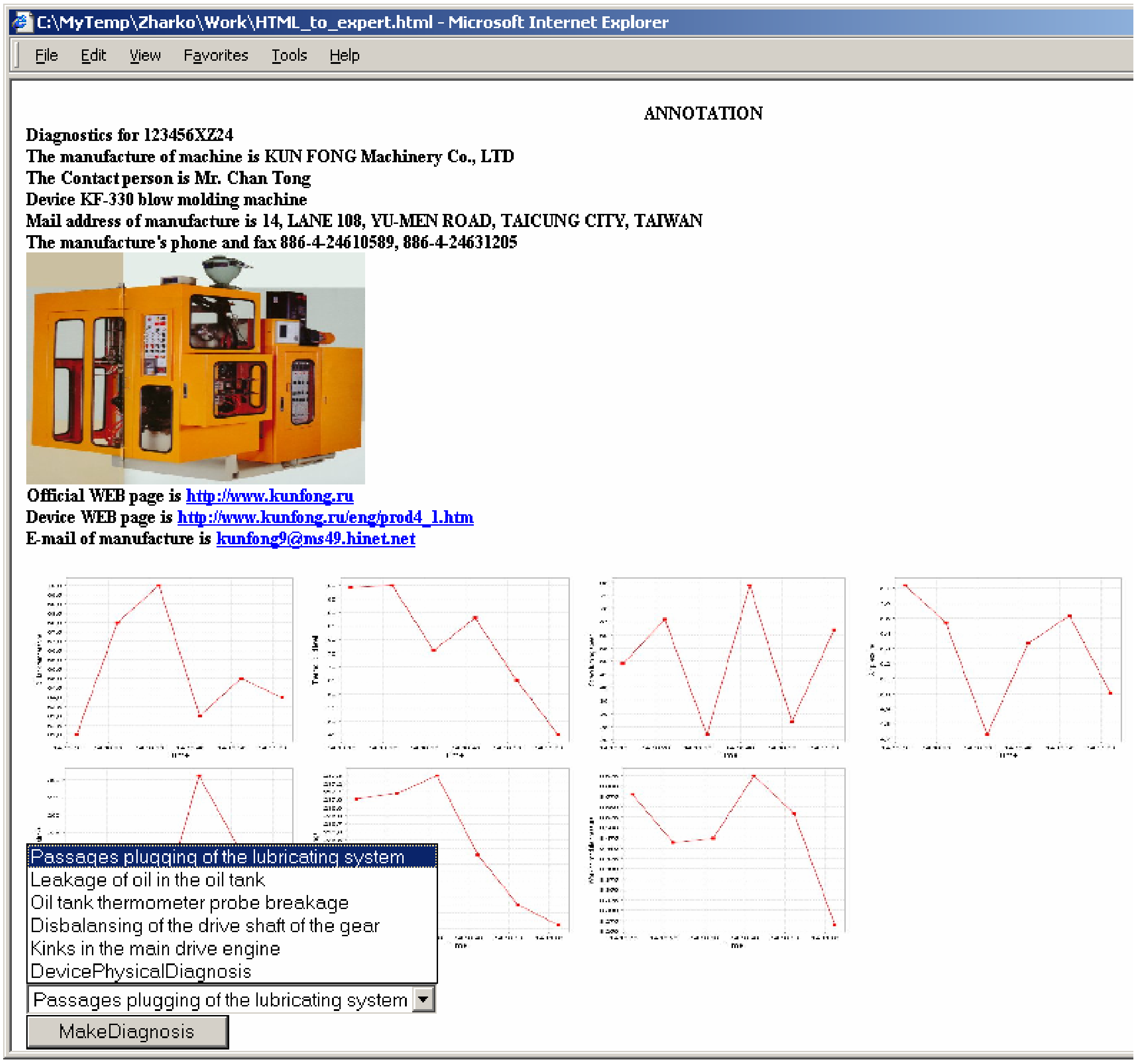
```
OneShotBehaviour tr = new OneShotBehaviour(this) {
    public void action() {
        String in;
        StringWriter sw = new StringWriter();
        resourceHistory.write(sw,null);
        in = sw.toString();
        try {
            AdapterWebUI myBean = getHomeExpert().create();
            myBean.transform(uri, in, UI_TEMPLATE, HOST_URL);
            }catch (Exception e) {
            e.printStackTrace(System.out);
        }
    }
};
```

Below there is an HTML that is a result of the transformation performed by ExpertAdapter.



```
<html>
<p><center><b>ANNOTATION</b></center>
<b>Diagnostics for <b>123456XZ24</b><br/>The manufacture of  machine is KUN FONG Machinery Co., LTD<br/>
The Contact person is Mr. Chan Tong<br/>
Device KF-330 blow molding machine<br/>
Mail address of manufacture is 14, LANE 108, YU-MEN ROAD, TAICUNG CITY, TAIWAN<br/>
The manufacture's phone and fax  886-4-24610589, 886-4-24631205<br/>
<IMG SRC="http://www.kunfong.ru/pic/prod4_1_1.jpg"><br/>
Official WEB page is <A HREF="http://www.kunfong.ru">http://www.kunfong.ru</A><br/>
Device WEB page is <A HREF="http://www.kunfong.ru/eng/prod4_1.htm">http://www.kunfong.ru/eng/prod4_1.htm</A><br/>
E-mail of manufacture is  <A HREF="mailto:kunfong9@ms49.hinet.net">kunfong9@ms49.hinet.net </A><br/></b></p>
<img src=".\img\0.jpg">
<img src=".\img\1.jpg">
<img src=".\img\2.jpg">
<img src=".\img\3.jpg">
<img src=".\img\4.jpg">
<img src=".\img\5.jpg">
<img src=".\img\6.jpg">
<form action="http://localhost:8080/Privet/Privet" method="post">
<select name="diagnoses">
<option value="http://www.cc.jyu.fi/~olkhriye/rscdfs/0.3/ontologies/diagnosisOntology#DeviceDiagnosis_4">Passages plu
<option value="http://www.cc.jyu.fi/~olkhriye/rscdfs/0.3/ontologies/diagnosisOntology#DeviceDiagnosis_2">Leakage of o
<option value="http://www.cc.jyu.fi/~olkhriye/rscdfs/0.3/ontologies/diagnosisOntology#DeviceDiagnosis_1">Oil tank the
<option value="http://www.cc.jyu.fi/~olkhriye/rscdfs/0.3/ontologies/diagnosisOntology#DeviceDiagnosis_3">Disbalansing
<option value="http://www.cc.jyu.fi/~olkhriye/rscdfs/0.3/ontologies/diagnosisOntology#DeviceDiagnosis_5">Kinks in the
<option value="http://www.cc.jyu.fi/~olkhriye/rscdfs/0.3/ontologies/diagnosisOntology#DevicePhysicalDiagnosis">Device
</select><br/>
<input type="hidden" name="expertrequesturi"
value="http://ohcejohka.ad.jyu.fi:2020/SmartResource#Message_for_Diagnosis351"/>
<input type="submit" name="useraction" value="Make diagnosis"/>
</form>
</html>
```

However ExpertAdapter produces not only an HTML page, it generates the pictures of the parameter values. An important element of the generated HTML page is a hidden field, containing the unique URI of the diagnostic request statement (see the highlighted code above). This unique URI allows for completely asynchronous interaction with expert. An expert can process requests in a random order, because the diagnosis statement is unambiguously associated with the diagnostic request by the URI. The interface provided to human expert for carrying out a diagnostics is given on Figure 15.


C:\MyTemp\Zharko\Work\HTML_to_expert.html - Microsoft Internet Explorer

File Edit View Favorites Tools Help

ANNOTATION

Diagnostics for 123456XZ24
The manufacture of machine is KUN FONG Machinery Co., LTD
The Contact person is Mr. Chan Tong
Device KF-330 blow molding machine
Mail address of manufacture is 14, LANE 108, YU-MEN ROAD, TAICUNG CITY, TAIWAN
The manufacture's phone and fax 886-4-24610589, 886-4-24631205

Official WEB page is http://www.kunfong.ru
Device WEB page is http://www.kunfong.ru/eng/prod4_1.htm
E-mail of manufacture is kunfong9@ms49.hinet.net

Passages plugging of the lubricating system
Leakage of oil in the oil tank
Oil tank thermometer probe breakage
Disbalansing of the drive shaft of the gear
Kinks in the main drive engine
DevicePhysicalDiagnosis

Passages plugging of the lubricating system

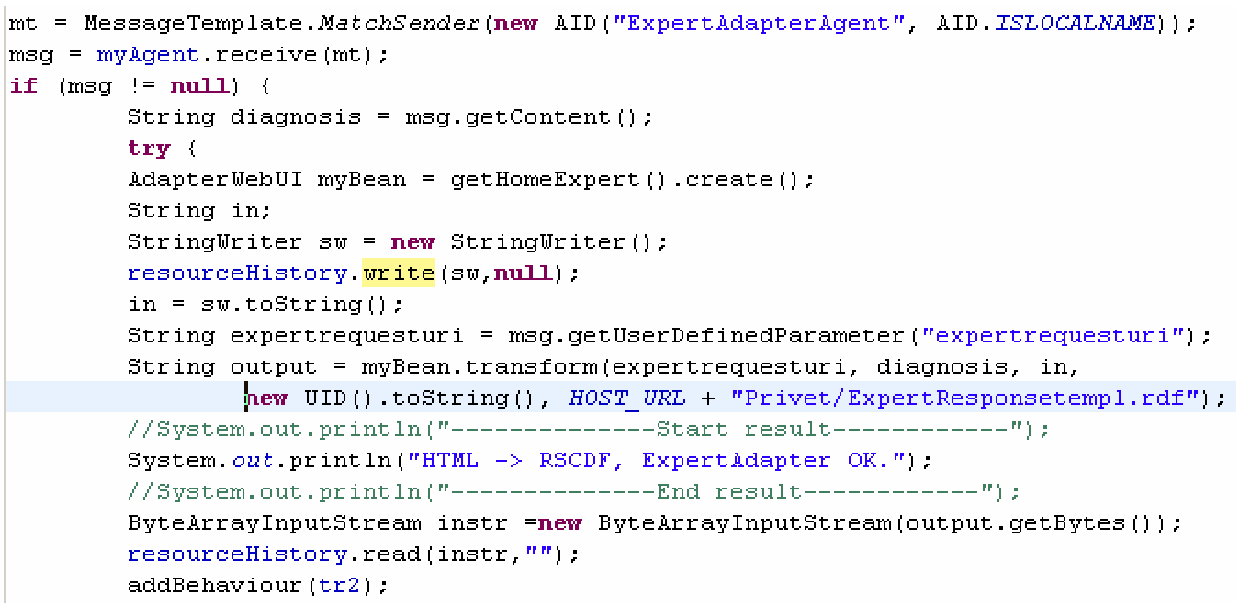
MakeDiagnosis

Figure 145 – Interface generated by ExpertAdapter

After an expert has selected an appropriate diagnosis, the data is sent to a processing servlet, which logically is a part of the adapter. The servlet processes the HTTP request and invokes method *sendDiagnosis* of ExpertAdapterAgent. The role of ExpertAdapterAgent is to be the bridging element between the servlet and ExpertAgent. Below is fragment of code with the sendDiagnosis method:

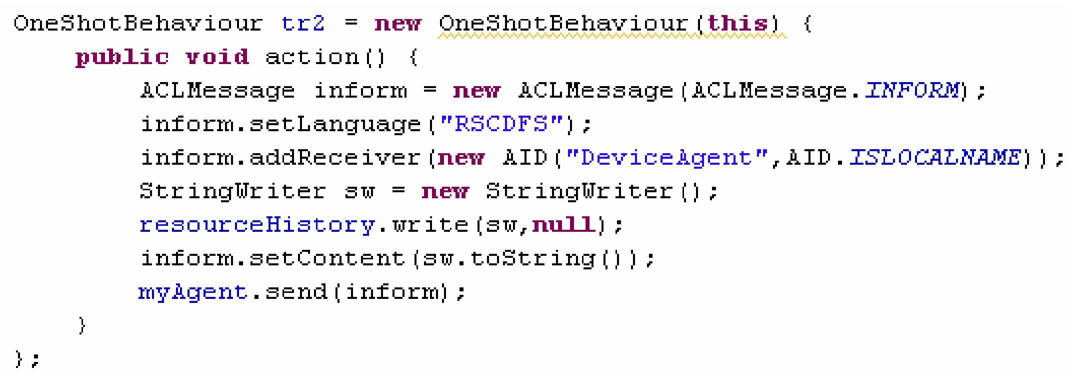
```
public void sendDiagnosis(String diagnosis, String expertrequesturi){
    ACLMessage inform = new ACLMessage(ACLMessage.INFORM);
    inform.setLanguage("RSCDF");
    inform.addReceiver(new AID("ExpertAgent",AID.ISLOCALNAME));
    inform.addUserDefinedParameter("expertrequesturi", expertrequesturi);
    inform.setContent(diagnosis);
    send(inform);
}
```

The message from ExpertAdapterAgent is processed by ExpertAgent as follows:



```
mt = MessageTemplate.MatchSender(new AID("ExpertAdapterAgent", AID.ISLOCALNAME));
msg = myAgent.receive(mt);
if (msg != null) {
        String diagnosis = msg.getContent();
        try {
        AdapterWebUI myBean = getHomeExpert().create();
        String in;
        StringWriter sw = new StringWriter();
        resourceHistory.write(sw,null);
        in = sw.toString();
        String expertrequesturi = msg.getUserDefinedParameter("expertrequesturi");
        String output = myBean.transform(expertrequesturi, diagnosis, in,
                new UID().toString(), HOST_URL + "Privet/ExpertResponsetempl.rdf");
        //System.out.println("--------------Start result------------");
        System.out.println("HTML -> RSCDF, ExpertAdapter OK.");
        //System.out.println("--------------End result------------");
        ByteArrayInputStream instr =new ByteArrayInputStream(output.getBytes());
        resourceHistory.read(instr,"");
        addBehaviour(tr2);
```

The behavior added in the last line of code below is a OneShotBehaviour. It sends the message with the diagnosis in RscDF format to a DeviceAgent:

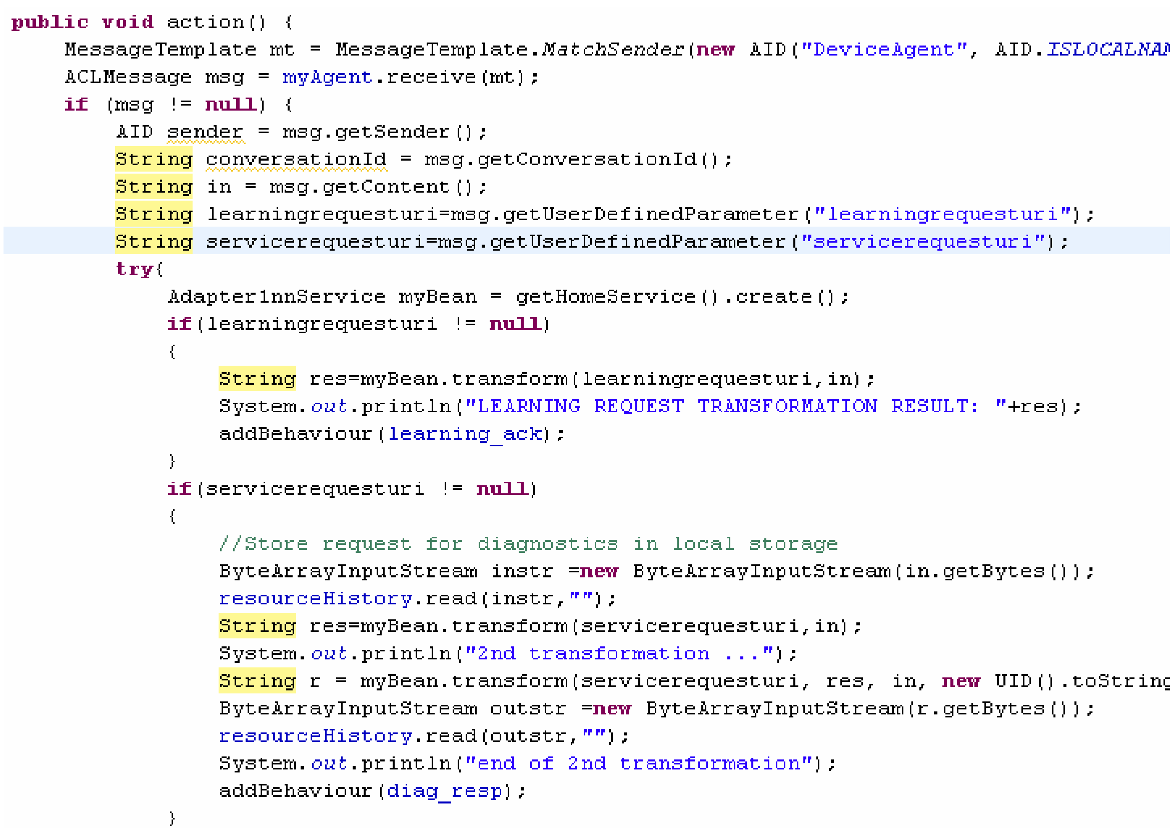


```
OneShotBehaviour tr2 = new OneShotBehaviour(this) {
    public void action() {
        ACLMessage inform = new ACLMessage(ACLMessage.INFORM);
        inform.setLanguage("RSCDFS");
        inform.addReceiver(new AID("DeviceAgent",AID.ISLOCALNAME));
        StringWriter sw = new StringWriter();
        resourceHistory.write(sw,null);
        inform.setContent(sw.toString());
        myAgent.send(inform);
    }
};
```

3.5 ServiceAgent

The ServiceAgent logic is quite simple. It implements one cyclic behavior for accepting messages from device. The messages can be of two types – request for diagnostics or request for learning. The messages are distinguished by UserDefinedParamter "*learningrequesturi*" and "*servicerequesturi*". In case of learning request, the adapter performs transformation and then the behavior is added, which sends a confirmation to a DeviceAgent. When the diagnostic request is received, it is processed in two transformation steps. On the first step the classification as such is done, which retuns diagnosis URI. On the second stage, the diagnosis URI is wrapped into an RscDF message using appropriate template. After the second transformation step, the

OneShotBehavior is added, which sends the response with diagnosis to a DeviceAgent. For implementation details see the fragment of the code below:



```
public void action() {
    MessageTemplate mt = MessageTemplate.MatchSender(new AID("DeviceAgent", AID.ISLOCALNAM
    ACLMessage msg = myAgent.receive(mt);
    if (msg != null) {
        AID sender = msg.getSender();
        String conversationId = msg.getConversationId();
        String in = msg.getContent();
        String learningrequesturi=msg.getUserDefinedParameter("learningrequesturi");
        String servicerequesturi=msg.getUserDefinedParameter("servicerequesturi");
        try{
            Adapter1nnService myBean = getHomeService().create();
            if(learningrequesturi != null)
            {
                String res=myBean.transform(learningrequesturi,in);
                System.out.println("LEARNING REQUEST TRANSFORMATION RESULT: "+res);
                addBehaviour(learning_ack);
            }
            if(servicerequesturi != null)
            {
                //Store request for diagnostics in local storage
                ByteArrayInputStream instr =new ByteArrayInputStream(in.getBytes());
                resourceHistory.read(instr,"");
                String res=myBean.transform(servicerequesturi,in);
                System.out.println("2nd transformation ...");
                String r = myBean.transform(servicerequesturi, res, in, new UID().toString
                ByteArrayInputStream outstr =new ByteArrayInputStream(r.getBytes());
                resourceHistory.read(outstr,"");
                System.out.println("end of 2nd transformation");
                addBehaviour(diag_resp);
            }
```

3.6 Platform in Runtime

The platform in runtime performs in asynchronous mode. The messages flow can easily be reconfigured if we want to change the logic of some of the components. In current implementation, the request for diagnostics is sent to an expert until enough learning examples for service are collected. After the service has learned, all the diagnostic requests are sent to ServiceAgent. The logic can be changed e.g. that every second example goes to Expert. This kind of configuration may be reasonable, when the service is not reliable enough. Another possibility is to send the diagnostic request to both Expert and Service. This kind of logic can be implemented for Service quality verification. Thanks to the fact, that the platform consists of highly independent agent enitites and unified semantic interchange format, the modification of logic becomes fairly simple and converges to modification of one java method in the DeviceAgent. Figure 16 depicts the sequence of messages passed before the first request for learning is sent. Figure 17 shows that from now the classification requests are passed from DeviceAgent to ServiceAgent because a Service has learned.

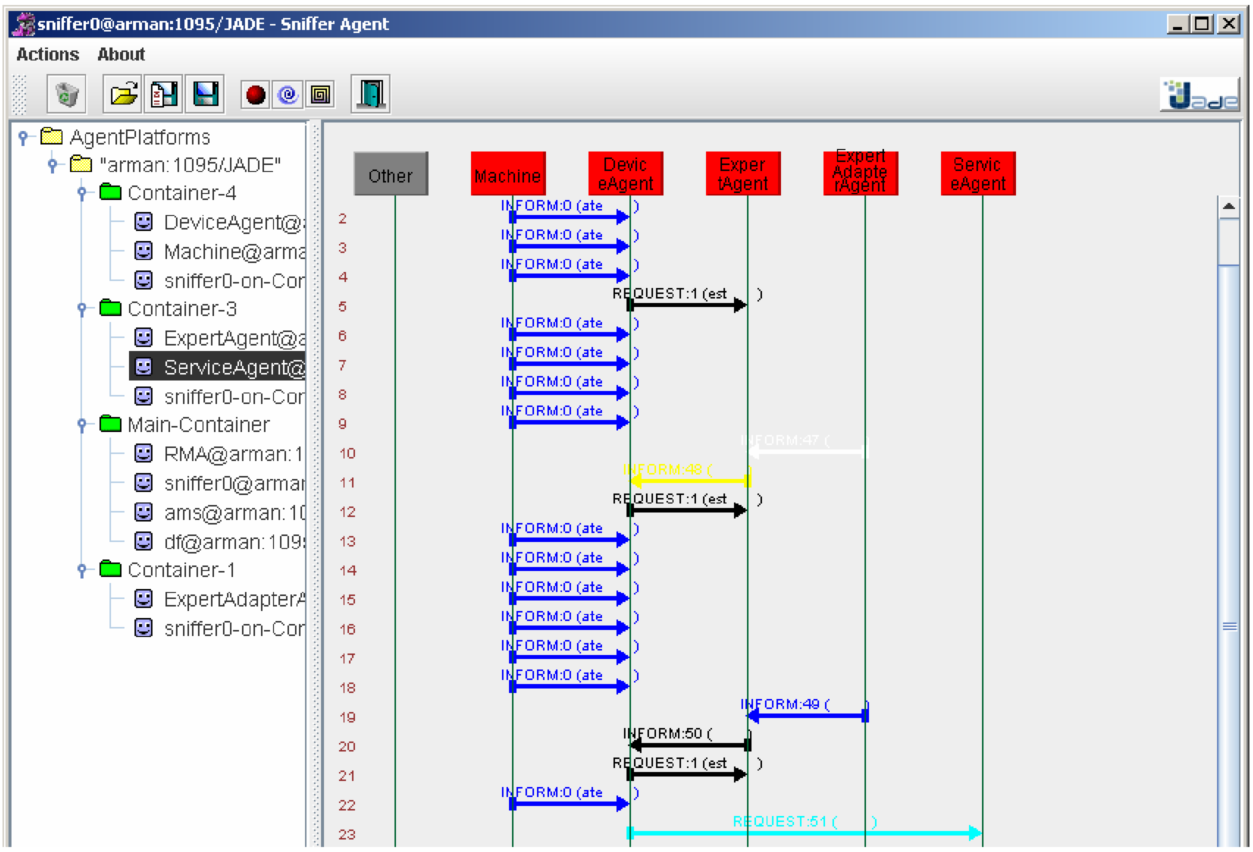


Figure 156 – Message Sequence before ServiceLearningRequest

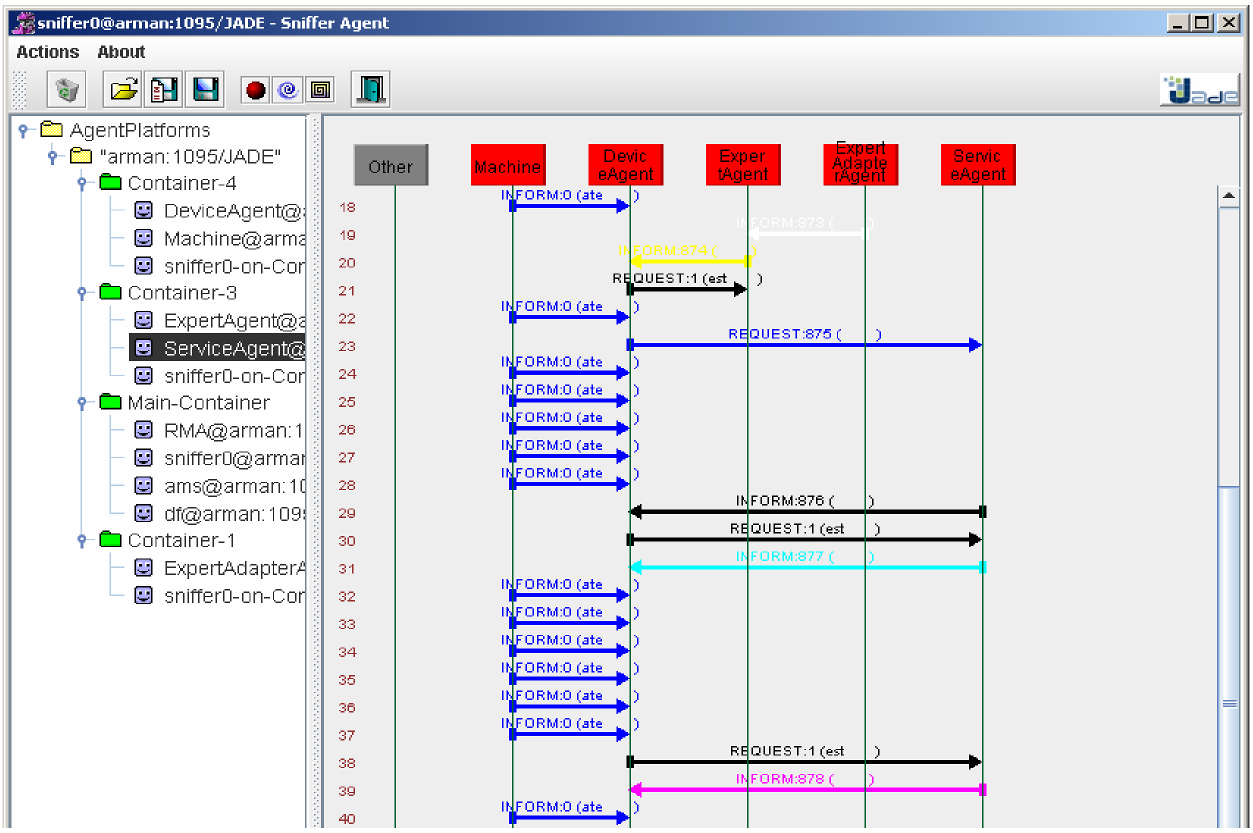


Figure 167 – Message Sequence up to ServiceLearningRequest

Conclusions

The transformation of the prototype environment v1.0 to v2.0 was aimed at automating the platform in terms of agent technology, which would act as a set of agents representing resources behind them. Supplying every resource by its own agent, we have explored the implementation specifics of the asynchronous message exchange applied to the use case of the knowledge transfer from expert to service. The implementation has discovered a new type of interaction with the resources which have undetermined response time and web-based interface. An expert as the most complex resource for adaptation required a lot of efforts to be done towards weaving different technologies into logically bundled component. The complex interoperation tasks between JADE and application server included EJB invocation from an agent platform, on-the-fly creation of the html data, dynamic processing of the expert response and artificial bridging from the web server to an agent platform via creation and posting an agent from the servlet to the running platform. We named this kind of interaction as resource choreography.

References

[1] Kaykova O., Khriyenko O., Terziyan V., Zharko A., General Proactivity Framework (Pro-GAF) , Technical Report (Deliverable D 2.1), SmartResource Tekes Project, Agora Center, University of Jyvaskyla, February-May, 2005.

[2] Official web-page of SmartResource TEKES project, http://www.cs.jyu.fi/ai/OntoGroup/SmartResource_details.htm

[3] Khriyenko O., Proactivity Layer of the Smart Resource in Semantic Web, In: Proceedings of the 3-rd European Semantic Web Conference ESWC06, June 11-14, 2006, Budva, Montenegro, Springer, LNAI, 14 pp. (submitted 18 November, 2005).

[4] Kaykova O., Khriyenko O., Kovtun D., Marttinen J., Naumenko A., Nikitin S., Terziyan V., Tsaruk Y., Zharko A., SmartResource Prototype Environment, Version 1.0, “Adaptation Stage”, Technical Report (Deliverable D 1.3), SmartResource Tekes Project, Agora Center, University of Jyvaskyla, July-December, 2004.

[5] Kaykova O., Khriyenko O., Terziyan V., Zharko A., RGBDF: Resource Goal and Behaviour Description Framework, In: M. Bramer and V. Terziyan (Eds.): Industrial Applications of Semantic Web, *Proceedings of the 1-st International IFIP/WG12.5 Working Conference IASW-2005* , August 25-27, 2005, Jyvaskyla, Finland, Springer, IFIP, pp. 83-99.

[6] Kaykova O., Khriyenko O., Terziyan V., Zharko A., Design of the SmartResource Platform, Technical Report (Deliverable D 2.2), SmartResource Tekes Project, Agora Center, University of Jyvaskyla, June-October, 2005.