Deliverable D1.4
Implementation of multi-agent based infrastructure

Work Package 1
Multi-agent Architecture

Deliverable D1.4
Implementation of multi-agent based infrastructure

Document type: Deliverable
Document version: Draft version 1
Document Preparation Date: Friday, September 28, 2012
Classification: Public
Author(s): IPB, Whirlpool, AEA, SINTEF, UNIVPM
File Name: Deliverable D1-4_0.7.pdf

Project Ref. Number: 246203
Project Start Date: 01/07/2010
Project Duration: 36 months
Website: www.grace-project.org
<table>
<thead>
<tr>
<th>Rev.</th>
<th>Content</th>
<th>Resp. Partner</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Structure of the document and preliminary description of some chapters</td>
<td>IPB</td>
<td>23/07/2012</td>
</tr>
<tr>
<td>0.2</td>
<td>The contents of all chapters are included</td>
<td>IPB</td>
<td>14/09/2012</td>
</tr>
<tr>
<td>0.3</td>
<td>Improvement of the document with the inclusion of the description of pieces of code. Revision of the text.</td>
<td>IPB</td>
<td>26/09/2012</td>
</tr>
</tbody>
</table>
# Table of Contents

Table of Figures ........................................................................................................................................ 4  
Acronyms ................................................................................................................................................. 6  
1. Introduction ............................................................................................................................................. 7  
2. Agent-based development frameworks ................................................................................................. 8  
   2.1. Existing development frameworks ...................................................................................................... 8  
   2.2. JADE development framework ........................................................................................................... 9  
3. Implementation of the behaviours of GRACE agents .............................................................................. 14  
   3.1. Package Structure of GRACE Multi-agent System .............................................................................. 14  
   3.2. Structure of a GRACE Agent ............................................................................................................. 16  
   3.3. Parameterization of the Agent ........................................................................................................... 19  
   3.4. Registration, Searching and Deregistration in the Director Facilitator .............................................. 21  
   3.5. Local Database ................................................................................................................................... 23  
   3.6. Interaction Protocols .......................................................................................................................... 23  
4. Integration of the ontology within the agents ......................................................................................... 26  
   4.1. Available Solutions to Translate Ontologies for JADE ......................................................................... 26  
   4.2. Bean Generator Plug-in ...................................................................................................................... 29  
   4.3. Implementation of the generated classes .............................................................................................. 31  
   4.4. Usage of the GRACE Ontology ......................................................................................................... 34  
5. Integration with legacy systems ............................................................................................................... 40  
   5.1. Interface with LabView applications .................................................................................................... 40  
      5.1.1. Interface Strategy ............................................................................................................................. 40  
      5.1.2. Adaptation Rules ............................................................................................................................. 43  
   5.2. Interface with the production database ............................................................................................... 46  
6. Development of the Graphical User Interfaces ..................................................................................... 48  
7. Lessons learned and Conclusions ......................................................................................................... 53  
References ................................................................................................................................................... 54
Table of Figures

Figure 1. Articulation of Task 1.4 with other tasks from WP1 and WP5 ........................................ 7
Figure 2. Graphical User Interface of the Director Facilitator ......................................................... 10
Figure 3. Containers in the JADE Platform [1] .............................................................................. 10
Figure 4. Graphical User Interface of the Remote Monitoring Agent ............................................. 11
Figure 5. Graphical User Interface of the Dummy Agent ............................................................... 12
Figure 6. Graphical User Interface of the Sniffer Agent ............................................................... 12
Figure 7. Graphical User Interface of the Instrospector Agent ..................................................... 13
Figure 8. GRACE project packages .............................................................................................. 14
Figure 9. Classes included in the PA package .................................................................................. 15
Figure 10. Basic Services package .................................................................................................... 16
Figure 11. Agent execution cycle [5] ............................................................................................... 17
Figure 12 – Each agent consult its own XML file to get the skills information ......................... 20
Figure 13 Example of a conversation using the ontology and the FIPA Protocol .......................... 24
Figure 14. Possible approaches for integration of the ontology in multi-agent systems .......... 27
Figure 15. JADE Abstract Ontology for OntologyBeanGenerator .............................................. 29
Figure 16. Partial view of the GRACE ontology with OntologyBeanGenerator concepts ........ 30
Figure 17. Screen shot of the OntologyBeanGenerator plug-in for JADE ................................. 30
Figure 18. Example of some concepts generated by the OntologyBeanGenerator ..................... 31
Figure 19. Agents using ontologies to exchange knowledge .......................................................... 35
Figure 20. Excerpt from the process of sending the Processplan object from PA to RA ............ 37
Figure 21. ACL message exchanged between PA and RA agents ................................................ 38
Figure 22. XML structure used to exchange data from the QCA and the QCS ......................... 40
Figure 23. “in process” approach for the communication between Java and LabVIEW™ ........ 41
Figure 24. “out process” approach for the communication between Java and LabVIEW™ using web services ................................................................................................. 41
Figure 25. Classes used on GRACE to handle the interfaces with LabView applications .......... 42
Figure 26. Example of the XML (input and output) files .............................................................. 43
Figure 27. Screenshot of the Product Type Agent .......................................................................... 48
Figure 28. Screenshot of the Product Agent .................................................................................... 49
Figure 29. Screenshot of the Resource Agent ................................................................................ 49
Figure 30. Screenshot of the Resource Agent executing an Operation after Applying Adaptation Procedures .......................................................... 50

Figure 31. Screenshot of the Interdependent Meta Agent ............................................ 50
Deliverable D1.4
Implementation of multi-agent based infrastructure

Acronyms

| ACC | Agent Communication Channel |
| ACL | Agent Communication Language |
| AMS | Agent Management System |
| DF  | Directory Facilitator |
| FIFO| First In, First Out |
| FIPA| Foundation for Intelligent Physical Agents |
| GRACE | inteGration of pRocess and quAlity Control using multi-agEnt technology |
| GUI | Graphical User Interface |
| JADE | Java Agent DEvelopment Framework |
| JNI | Java Native Interface |
| OWL | Web Ontology Language |
| PA  | Product Agent |
| QCA | Quality Controller Agent |
| QCS | Quality Control Station |
| RA  | Resource Agent |
| RDF | Resource Description Framework |
| RMA | Remote Management Agent |
| RMI | Remote Method Invocation |
| SL  | Semantic Language |
| SPARQL | Protocol and RDF Query Language |
| SWRL | Semantic Web Rule Language |
| UDDI | Universal Description Discovery and Integration |
| UML | Unified Modelling Language |
| WSDL | Web Service Definition Language |
| JDBC | Java Database Connectivity |
1. Introduction

This deliverable contains the outcome of the Task 1.4, entitled “Development of the multi-agent architecture”, which objective is to develop the multi-agent system infrastructure according to the specifications established in the previous tasks (mainly tasks 1.2 and 1.3).

For this purpose, and aiming at simplifying the implementation of the multi-agent system infrastructure, the Java Agent DEvelopment Framework (JADE) [1] was used. JADE is totally implemented in Java and provides a set of services to support the easy development and management of agent based solutions, such as white and yellow pages services and debugging tools, and is compliant with the Foundation for Intelligent Physical Agents (FIPA) specifications [2].

As illustrated in Figure 1, this agent-based infrastructure will be used in the rest of the project to host the mechanisms and algorithms developed during the others WPs, and the basis for the prototype demonstration that will be set-up in the WP5 (including integration, customization and validation).

![Figure 1. Articulation of Task 1.4 with other tasks from WP1 and WP5](image)

The document is divided into seven chapters. After this brief introduction, chapter 2 will provide an overview of the existing agent development frameworks and provides some details of the JADE framework which is used in this work. Chapter 3 is devoted to the description of the implementation of the behaviours of GRACE agents and Chapter 4 presents the integration of the ontology within the GRACE multi-agent system to represent the shared knowledge. Chapter 5 describes the details about the integration of the legacy systems, namely the production database and quality control applications, and Chapter 6 provides details about the developed graphical user interfaces. Finally, Chapter 7 is devoted to the discussion of the lessons learned during the implementation process.
2. Agent-based development frameworks

This chapter overviews the currently available technologies to implement multi-agent systems based solutions and describes in particular the selected development framework used to implement the GRACE multi-agent system infra-structure.

2.1. Existing development frameworks

Multi-agent systems can be adequately developed using usual object-oriented languages, such as Java and C++. However, the development of multi-agent system solutions requires the implementation of features not supported by usual programming languages, such as message transport, encoding and parsing, white and yellow pages services, ontologies for common understanding and agent life-cycle management services, which increases the programming effort. The use of agent development platforms, that provide the previously referred features and services, makes easier the development of agent-based applications and reduces the programming effort. Additionally, they also provide a higher level of abstraction of some technical details.

A significant set of platforms for the development of agent-based solutions is available on commercial and scientific domain, providing a variety of services and agent models, which differences reflect the philosophy and the target problems envisioned by the platform developers. Among the broad number of available agent development platforms, the following platforms were analysed: ZEUS, FIPA-OS\(^1\), Java Agent Development Framework (JADE)\(^2\), Grasshopper, JACK, April. Table 1 summarizes a comparison among these platforms, taking into account several parameters, such as the type of agent model, FIPA compliance and license type. More details can be found in the Agent Builder website (http://www.agentbuilder.com/AgentTools/) and some surveys can be found in [3] and [4].

<table>
<thead>
<tr>
<th>Platform</th>
<th>Licence</th>
<th>Agent model</th>
<th>FIPA compliance</th>
<th>Still Maintained</th>
<th>Being Developed</th>
</tr>
</thead>
<tbody>
<tr>
<td>April</td>
<td>Freeware</td>
<td>Reactive</td>
<td>✔</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>FIPA-OS</td>
<td>Freeware</td>
<td>Reactive</td>
<td>✔</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>Grasshopper</td>
<td>Freeware</td>
<td>Reactive</td>
<td>✔</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>JACK</td>
<td>Commercial</td>
<td>BDI</td>
<td>✔(^3)</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>JADE</td>
<td>Freeware</td>
<td>Reactive</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Madkit</td>
<td>Freeware</td>
<td>Reactive</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>ZEUS</td>
<td>Freeware</td>
<td>Reactive</td>
<td>✔</td>
<td>❌</td>
<td>❌</td>
</tr>
</tbody>
</table>

The selection of the agent development platform to be used in the development of the GRACE multi-agent infra-structure has obeyed to a set of criteria, namely:

---

\(^1\) FIPA-OS is provided by Emorphia (http://fipa-os.sourceforge.net).
\(^2\) JADE is provided by CSELT (http://jade.csel.it/).
\(^3\) By using the FIPA JACK extension.
• Be an open source platform and particularly not proprietary.
• Follow the standards in the agents domain, and particularly be FIPA compliant (note that FIPA is currently the standard in the agents domain).
• Provide an ease of use and low programming effort.
• Provide features to support the management of agent-based solutions, such as white pages and yellow pages services.
• Provide an easy integration with other tools or facilities, namely with knowledge management platforms such as Protégé and decision-making systems, such as Java Expert System Shell (JESS) that implement rule-oriented programming.
• Provide good documentation and available support.

In the current work, the selected platform was the JADE platform because it better responds to the mentioned requirements.

2.2. JADE development framework

JADE is a Java-based architecture that uses the Java Remote Method Invocation (RMI) to support the creation of distributed Java technology-based applications. Each agent is implemented with Java “threads” and associated with a container.

JADE aims at simplifying the development of multi-agent systems by providing a set of services and agents in compliance with the FIPA specifications, e.g. naming service and yellow-page service, message transport and parsing service, and a library of FIPA interaction protocols ready to be used [5]. Note that in the essence, the agents developed using the JADE platform are Java Threads, which makes the debugging of multi-threading very difficult; consequently, some tools (implemented as agents) have been developed to simplify the development of agent-based solutions, being every single tool provided by JADE packaged as an agent itself.

JADE provides the mandatory components defined by FIPA to manage the multi-agent platform, which are the [1]:

• Agent Communication Channel (ACC), which is responsible for the conversation channel, supporting the communications among the agents and offering interoperability of the components.
• Agent Management System (AMS), which provides white pages and agent life cycle management services (controlling the access to the platform, authentication, and registration), maintaining a directory of agent identifiers and states.
• Directory Facilitator (DF), which provides a piece of shared memory offering the yellow pages services as defined in the FIPA specifications, and the capability of federation within other DFs. Using this tool, it is possible to register agent’s services, deregister the agents, modify some descriptions, or as the greatest utility of the yellow pages, look for services offered by other agents. Figure 2 illustrates the screenshot of the DF.
Deliverable D1.4
Implementation of multi-agent based infrastructure

Figure 2. Graphical User Interface of the Director Facilitator

In this way, the main container of a JADE multi-agent system application is composed of the ACC, AMS and DF agents, and by a RMI (Remote Method Invocation) registry, that is used by JADE for intra-platform communication, as illustrated in Figure 3.

Figure 3. Containers in the JADE Platform [1]

JADE uses the concept of behaviours to model concurrent tasks in agent programming [6]. A behaviour represents a task that an agent can carry out and is implemented as an object of a class that extends jade.core.behaviours.Behaviour. Several pre-defined behaviours are provided by JADE: One-Shot Behaviour, Cyclic Behaviour (CyclicBehaviour), Temp Behaviour (WakerBehaviour, TickerBehaviour), Compose Behaviour (SequentialBehaviour, ParallelBehaviour, FSMBehaviour). Different behaviours can be used, individually or combining more than one, depending on the expected agent behaviour.

The communication among the agents is performed through message passing, where FIPA-ACL (Agent Communication Language) is the agent communication language to represent messages. JADE provides the FIPA SL (Semantic Language) content language and the agent management ontology, as well as the support for user-defined content languages
and ontologies that can be implemented, registered with agents, and automatically used by the framework.

The Remote Management Agent (RMA) provides a Graphical User Interface (GUI) for the remote management of the platform allowing monitoring and controlling the status of agents, for example to stop and re-start agents (see Figure 4). The RMA allows a fully control of an agent life cycle from a remote host. When the JADE platform is started, a default container is created, which holds the RMA itself, the DF, and AMS.

![Graphical User Interface of the Remote Monitoring Agent](image)

**Figure 4.** Graphical User Interface of the Remote Monitoring Agent

The RMA agent provides a set of graphical tools (packaged as agents) to monitor the state of the agents and to support the debugging phase, usually quite complex in distributed systems. Examples of such tools are the Dummy, Sniffer and Introspector agents.

The *Dummy Agent*, illustrated in Figure 5, is a monitoring and debugging tool that allows to edit, compose and send ACL messages to agents, and to receive and view messages from agents.
When the complexity of the multi-agent system increases, it is very useful to use a tool to check the exchanged messages between the agents. The Sniffer Agent, illustrated in Figure 6, is a debugging tool that allows tracking messages exchanged in a JADE agent platform using a notation similar to the Unified Modelling Language (UML) sequence diagrams. It can be analysed the type of message, the FIPA protocol, the ontology language and its contents.

The Introspector Agent, illustrated in Figure 7, allows monitoring and controlling the lifecycle of a running agent, its exchanged ACL messages (incoming and the outgoing messages) and the behaviours in execution (allowing to execute them step-by-step).
JADE offers an easy and full integration with JESS (Java Expert System Shell), where JADE provides the shell of the agent and guarantees (where possible) the FIPA compliance, while JESS is the engine of the agent that performs all the necessary reasoning [5].

JADE also provides other features such as good documentation and an active mailing list to support technical problems.
3. Implementation of the behaviours of GRACE agents

As previously referred, the implementation of the GRACE agents uses the JADE framework. In this chapter, the generic issues related to the implementation of the structure of behaviours for the GRACE agents will be detailed.

3.1. Package Structure of GRACE Multi-agent System

Aiming to structure the implementation of the source code, a set of packages was created, as illustrated in Figure 8, separating the concepts and functions associated to the multi-agent system application.

![Figure 8. GRACE project packages](image)

Each package contains a set of classes that executes the functionalities associated to the agents. Some packages are related to only one agent, for example the PA or RA packages that are only associated to the PA and RA agents, respectively. As an example, the PA package contains a set of three classes, namely two for the agent itself (PA and WaitingMessage classes) and one related to the GUI (formPA class), as illustrated in Figure 9 (here, it is possible to see the attributes and methods of each one of these classes).
Some packages provide common functions that were used by the GRACE agents, namely the package \textit{BasicServices}, as illustrated in Figure 10, which provides services e.g., related to parsing XML files.
As an example, some functions to parse information are provided to be used by the several GRACE agents.

3.2. Structure of a GRACE Agent

As previously, JADE provides a framework containing several agents (i.e. ACC, AMS and RMA) that will support the management of the developed GRACE agent-based application. The GRACE multi-agent system comprises four types of agents, i.e. Product Type Agents (PTA), Product Agents (PA), Resource Agents (RA) and Independent Meta Agent (IMA), as specified in the GRACE deliverable 1.2 [7]. Each one of these GRACE agent types is a simple Java class that extends the *Agent* class provided by the JADE framework, inheriting basic functionalities, such as registration services, remote management and send/receive ACL messages [5]. These functionalities were extended with features that represent the specific behaviour of the agent, as described in the deliverable 1.2 [7]. The structure, and particularly the execution cycle, of the Agent is illustrated in Figure 11.
Figure 11. Agent execution cycle [5].

The analysis of the agent execution cycle reveals four important methods:

- **setup()**, which represents the start-up of the agent.
- **action()**, which implements the code related to the desired behaviours to be executed.
- **done()**, which tests if the behaviours included in the action method are finished or not.
- **takeDown()**, which performs the last actions of the agent life-cycle.

The GRACE agents, inheriting the JADE Agent class, also inherit this agent execution cycle, being necessary to define the functions and behaviours for each GRACE agent in the proper referred methods. As an example, the structure of the RA agent is illustrated in following extract of code.

```java
package RA;
import jade.*;
...
public class RA extends Agent implements GraceAgent{
private formRA_ myGui = null;
private boolean isMyGuiON = false;
@Override
protected void setup(){
    initialization();
    // start a behaviour to wait for incoming messages
    WaitingMessages waitingMessages = new WaitingMessages(this,
        isMyGuiON, myGui);
```
Deliverable D1.4
Implementation of multi-agent based infrastructure

```java
addBehaviour(waitingMessages);
}
...
}/* end of the RA class

The behaviour of each agent uses multi-threading programming, over the concept of the JADE’s behaviour, allowing executing several actions in parallel. Thus, when the agent is created, the first method to be executed is the `setup()` method that should be fulfilled with the actions to be performed during the start-up of the agent. Analysing the previous piece of code it is possible to verify that in the beginning of the setup method, an initialization function is executed being responsible to register the agents’ skills in the DF, connect to the local database, and create a GUI component, as illustrated in the following piece of code.

```java
@javax.annotation.PostConstruct
public void initialization(){
    loadProfile();
    registerAtDF();
    connectToLocalDB();
    createGui();
}
```

The communication between distributed agents is done over the Ethernet network, using TCP/IP protocol and is asynchronous, i.e. an agent that sends a message continues its execution without the need to wait for the response. The messages specified in the GRACE multi-agent system are encoded using the FIPA-ACL communication language to achieve normalized communication between the agents, being the content of the messages formatted according to the FIPA-SL0 language. The meaning of the message content is standardized according to the GRACEOntology.

For this purpose and since the behaviour of the agent is driven by the messages received from the other agents (i.e. incoming events), a cyclic behaviour called `WaitingMessages` is launched in the `setup()` method. This behaviour is a Java class that is waiting for the arrival of messages, using the `block()` method to block the behaviour until a message arrives and the `receive()` method to extract the incoming message, as illustrated in the next extract of code.

```java
// WaitingMessages waits for the messages from other agents.
class WaitingMessages extends CyclicBehaviour implements IConstants,
    IConstants_GUI {
    // list of attributes
    ...
    WaitingMessages(RA aThis, boolean myGuiON, formRA_ myGui){
        myAgentRA = aThis;
        this.myGui = myGui;
        this.isMyGuiON = myGuiON;
    }
    @Override
```
public void action(){
    ACLMessage msg = myAgent.receive();
    if (msg != null) {
        Integer key = Integer.parseInt(msg.getConversationId());
        switch (key) {
        case RA_STORE_ADVICES:
            myAgent.addBehaviour(new StoreAdvices());
            break;
        case RA_PROVIDES_INFO:
            myAgent.addBehaviour(new ProvidesInfo());
            break;
        case RA_ADAPTS_PARAMETERS:
            adaptsParameters();
            break;
        }
    } else {block();}
}

The arrival of a message triggers a set of actions related to decode the message and select the proper behaviours to be performed. Looking to the code, if the message is not null, it is verified the conversation ID of the message, and depending of the conversation ID of the message a different behaviour or function is called. As an example, if the received message has a RA_PROVIDES_INFO identification, a new behaviour called \textit{ProvidesInfo()} is triggered. Note that after triggering the action related to the received message, the \textit{WaitingMessages} behaviour continues waiting for incoming messages continuously.

All agents in the GRACE system are able to receive messages following this approach.

The behaviours launched in the \textit{setup()} method and those posterior invoked within these behaviours are also provided by the resource agent package in the form of Java classes.

### 3.3. Parameterization of the Agent

The launching of the multi-agent system requires two important actions: launching the JADE platform and launching instantiations of the agent classes developed for the specific agent-based solution (in this case for the GRACE case study).

The JADE platform is started by introducing the following command in the command line:

```
java jade.Boot -gui
```

This command starts the JADE platform, launching the associated agents to manage the agent-based application, namely ACC, AMS, RMA and DF.

After the initialization of the JADE platform, it is necessary to add the agents to the platform. In this way, the structures of the agent classes described in the previous sections for the four types of GRACE agents (i.e. PTA, PA, RA and IMA) are instantiated according to the production line needs. For example, a RA for the bearing insertion station is launched using the following command:
java jade.Boot -container A_Bearing_Insertion:RA.RA(gui);

where A_Bearing_Insertion is the name of the agent and RA.RA is the class for the resource agent (note that the first “RA” is related to the package and the second is the agent class).

Agents may get start-up arguments specified on the command line, supporting the customization of the agent instance. In the previous example, the A_Bearing_Insertion agent is launched with its GUI, which is indicated as argument. Note that the arguments on the command line are specified included in parenthesis and separated by spaces. The arguments are retrieved, as an array of Object, by means of the getArguments() method of the Agent class, as illustrated in the following piece of code.

```java
protected void setup() {
    ...
    // Gets the arguments
    Object[] args = getArguments();
    if (args != null && args.length > 0) {
        argument1 = (String) args[0];
        ...
    }
}
```

The several instances of the agent classes need additional customization and parameterization. For example, each one of the stations disposed along the line will be associated to instances of the RA agent, but each one has its particularities and it is necessary to reflect them in the generic structure of the RA agent. For this purpose, each agent has associated a XML file, describing the particularities and skills/services of the station that it will represent; the file is read when the agent is launched in order to load the agent profile with the correct parameters, as illustrated in Figure 12.

Figure 12 – Each agent consult its own XML file to get the skills information

For example, the following piece of code is used by the PA agent to get the process plan from the XML file.
Deliverable D1.4
Implementation of multi-agent based infrastructure

ParserXml xmlfile = new ParserXml();
xmlfile.getFile("D:/GRACE_MAS_FILES/ProcessPlan.xml");
xmfile.initGraceLineXML();

The information extracted from the XML file is then parsed, as illustrated in the following extract of code.

for (int s = 0; s < Operation.getLength(); s++) {
    Node OperationNode = Operation.item(s);
    if (OperationNode.getNodeType() == Node.ELEMENT_NODE) {
        Element OperationElement = (Element) OperationNode;
        NodeList TagName = OperationElement.getElementsByTagName("id");
        Element operationNameElement = (Element) TagName.item(0);
        NodeList textLNList = operationNameElement.getChildNodes();
        if (((((Node) textLNList.item(0)).getNodeValue().trim() ).toString()).equals(VR_old)) {
            ...
        }
    }
}

The package Basic Services comprises several parsing functions to support the parsing of different situations, such as the ParserXML class that is responsible to handle the parameterization values.

3.4. Registration, Searching and Deregistration in the Director Facilitator

An important mechanism in a multi-agent system is the yellow pages, which provides a service to find agents with specific characteristics or offering particular services/functionalities. In the JADE framework, the yellow pages service defined in the FIPA specification is implemented by the Directory Facilitator (DF).

The registration of an agent in the DF is performed using the following code.

    // Register the PA agent services in the yellow pages (DF)
    @Override
    public boolean registerAtDF()
    {
        boolean res = false;
        Object[] args = getArguments();
        if (args != null) {
            productID = (String) args[1];
Deliverable D1.4
Implementation of multi-agent based infrastructure

```java

DFAgentDescription dfd = new DFAgentDescription();
dfd.setName(getAID());

ServiceDescription service = new ServiceDescription();
service.setType(productID);
service.setName("ProductId");
dfd.addServices(service);

try {
    DFSERVICE.register(this, dfd);
    res = true;
} catch (FIPAException ex) {
    Logger.getLogger(PA.class.getName()).log(Level.SEVERE, null, ex);
    return false;
}
return res;
```

The agents can use the DF to search for other agents providing particular services. For this purpose it is necessary to use the `search()` method when asking the DF, as illustrated in the following piece of code where an agent searches a PA agent by its ID.

```java

DFAgentDescription template = new DFAgentDescription();
ServiceDescription sd = new ServiceDescription();
sd.setType(productId);
template.addServices(sd);
DFAgentDescription[] result = null;

try {
    result = DFSERVICE.search(myAgent, template);
} catch (FIPAException ex) {
    Logger.getLogger(WaitingMessages.class.getName()).log(Level.SEVERE, null, ex);
}

When the agent ends its life, it should deregistrates from the DF, using the following piece of code.

```java

@Override
protected void takeDown() {
    try {
        DFSERVICE.deregister(this);
    }
}
```
catch (Exception e) {} 
}

Note that if the agent doesn’t deregister, the other agents continue thinking that it exists in the system and continue sending requesting messages.

### 3.5. Local Database

In the GRACE multi-agent system, each agent has its local database which contains its knowledge and information. In this work, a SQL database is used.

The use of the Java Database Connectivity (JDBC) API permits the connection to the database to store and retrieve the necessary information by performing the appropriated queries.

```java
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver").newInstance();
String connectionUrl =
    "jdbc:sqlserver://192.168.201.65:1433;databaseName=ImaDB";
Connection conn =
    DriverManager.getConnection(connectionUrl, "sa", "Ima123");
```

The next code construction is used, with the proper SQL statement, wherever is necessary to get or store information from the database:

```java
ResultSet rs = null
try{
    rs = conn.createStatement().executeQuery(query);
}
```

### 3.6. Interaction Protocols

The interaction among the agents uses several interaction patterns according to the specifications elaborated in the deliverable 1.2 and following the FIPA Interaction Protocols Specifications. Figure 13 represents one example of a FIPA conversation, namely a Query performative.
This kind of conversations can became more and more complex depending of the amount of agent’s interaction. The Figure 13 contains a conversation between two agents (a1, a2), where the receiver agent (a1) receives messages which contains several message characteristics, such as the identification of the agent that sends the message, the receiver agent, the ID of the conversation, the identifier of the ontology, the contents of the message, among others characteristics. For example, the following piece of code fulfills the message features.

```java
int performative= ACLMessage.INFORM;
ACLMessages msg = new ACLMessage(performance);
msg.addReceiver(server);
msg.setConversationId(123);
msg.setContent("Content");
send(msg);
```

The performative parameter can be accessed by the class ACLMessage, where each performative corresponds to an integer value (however, these constants can be described by text, e.g. CFP, CONFIRM, CANCEL, FAILURE, INFORM and NOT_UNDERSTOOD). The receiver agent can create a temple of performatives to filter the messages, for example if the agent a2 creates a template for the FAILURE performative, the agent never receive a message because all messages are from different performatives. It can also filter the message from a specific agent. The next piece of code represents a filter of the INFORM performative and referred only to the Machine agent.

```java
MessageTemplate mt =
    MessageTemplate.and(MessageTemplate.MatchPerformatives(ACLMessage.INFORM),
    MessageTemplate.MatchSender(new AID("Machine",
        AID.ISLOCALNAME)));```
In the GRACE project, the exchanged messages have an ID, which is represented in the class interface *IConstants* of the package *Core*. Thus every message sent among the agents has a content, protocol, sender, and identification. The receiver agent knows exactly what to do when receives a message with a specific ID message. These IDs are common among the multi-agent system solutions so every agent can use them.
4. Integration of the ontology within the agents

The ontology, designed in the Deliverable 1.3 [8], plays a crucial role in the GRACE multi-agent system to enable a common understanding among the agents when they are communicating, namely to understand the message at the syntactic level (to extract the content correctly) and at the semantic level (to acquire the exchanged knowledge). This chapter describes the integration of the designed ontology in the developed multi-agent system solution.

4.1. Available Solutions to Translate Ontologies for JADE

Since the GRACE multi-agent system is being developed using the JADE framework, a pertinent question is how to translate the ontology edited and validated in Protégé during the Task 1.3 to be used by the agents developed in JADE.

The exchange of messages with ontologies structure in JADE normally uses an internal container with a restrict format. Due to the fact that the messages are FIPA compliant, JADE agents are able to interact with other agents, not only from the JADE platform but with different systems. For this purpose, JADE created a reference model along with some notions: Concept, Agent Action and Predicate. The new term here is the Agent Action that characterizes an action performed by an agent; any act that the agents can perform should be described as Agent Action. When an Agent Action is performed, it could generate an effect between the agents, e.g., sendMessage or createProduct. During the ontology development, all ontological objects should implement one of the referred interfaces (i.e. Concept, Agent Action or Predicate).

In this way, when an ontology is integrated in the JADE framework, it is necessary to derivate the ontological terms from the interfaces, and generating a set of Java classes. The final result is a model more semantically and expressive for the content language.

The integration of the ontology in the multi-agent system can be performed in two different ways, manually or automatically. The first one is developing the AgentAction, Concept and Predicate classes derived from the ontology schema by hand. This process is very difficult and time consuming; also the time spent to improve any term and refine the process is very high.

The second option is to use some tools that provide some support do develop automatically these classes. Figure 14 summarizes the several alternatives to translate the ontology into Java classes using automatic tools. Any approach represented in the figure follows the same steps: after the design of the ontology schema, it is necessary to produce the knowledge base. At this stage the ontological schema is supported in the OWL language, being the one language purpose to create a common shared content.
The presented alternatives allow the storage procedure in different platforms (memory, text files, databases, etc.). At this stage, the translation to Java classes can be done by translating OWL/RDF structures to Java classes or in alternative to use querying tools, such as SPARQL (Protocol and RDF Query Language), to interrogate the ontological model and create a table result. An alternative approach is to use the OntologyBeanGenerator plug-in that allows to directly generate automatically the Java classes from the Protegé tool.

The several approaches to translate the ontology schema for the Java classes are illustrated in Table 2. The referred alternatives allow reasoning and data importation from several kinds of databases and also reasoning.
Table 2 – Comparison of the possible approaches to translate ontology schema for Java classes

<table>
<thead>
<tr>
<th>Approach</th>
<th>Import from diverse DB</th>
<th>Good user interface</th>
<th>Get Information</th>
<th>Export formats</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protegé (classes)</td>
<td>Yes</td>
<td>Yes</td>
<td>Needs to be exported</td>
<td>XML, RDF, OWL, N-triple</td>
<td>Easy to work</td>
</tr>
<tr>
<td>JENA (SPARQL)</td>
<td>Yes</td>
<td>No (it is a Java API)</td>
<td>Needs to handle the results like SQL result</td>
<td>XML, RDF, OWL, N-triple</td>
<td>Getting data from the knowledge base became difficult</td>
</tr>
<tr>
<td>JENA (classes)</td>
<td>Yes</td>
<td>No (it is a Java API)</td>
<td>Gets the information from the objects</td>
<td>XML, RDF, OWL, N-triple</td>
<td>Changing the schema and knowledge base take some time</td>
</tr>
<tr>
<td>KAON2 (classes or SPARQL)</td>
<td>Yes</td>
<td>No (it is a Java API)</td>
<td>Needs to handle the results like SQL result</td>
<td>XML, RDF, OWL, N-triple</td>
<td>Getting data from the knowledge base became difficult</td>
</tr>
<tr>
<td>Protegé API (classes or SPARQL)</td>
<td>Yes</td>
<td>No (it is a Java API)</td>
<td>Gets the information from the objects</td>
<td>XML, RDF, OWL, N-triple</td>
<td>-</td>
</tr>
<tr>
<td>Protegé + BeanGenerator</td>
<td>Yes</td>
<td>Yes</td>
<td>Exports directly to Java classes</td>
<td>XML, RDF, OWL, N-triple</td>
<td>Easy to create the Java classes (also the modification)</td>
</tr>
</tbody>
</table>

The selection of the alternative that best fits the needs and constraints of this project, must consider issues like the facility to have a fast development and the exporting process.

Jena is a common framework that can be used in several approaches. It can be used individually, but it is explicitly used as the basis of Protegé API. Moreover, it was building a plug-in for JADE to use OWL files. This plugin, entitled AgentOWL [9], is available on the website of JADE, and uses pure OWL files.

Analysing the figure the one that best fits the requirements of this project is the Protegé + Bean Generator. The Protegé offers an export to different formats (RDF, OWL, etc.), by adding the Protegé plugin an exportation to Java classes is offered. However, this approach has some disadvantages, namely the loss of flexibility in multi-agent systems, because it loses the rationalization of new facts and new rules that will have to be in the ontology, with advanced methods, such as Java Reflexion or Jess (rule engine for the Java platform). If an approach that works with OWL is chosen, the tool can run an inference engine with a set of rules, for example expressed in SWRL (Semantic Web Rule Language) and SPARQL (with inserts). But for this work, this kind of transformation “on-the-fly” is not needed.
4.2. Bean Generator Plug-in

The integration of the ontology can be done without any tool as previously referred. However, using the OntologyBeanGenerator plug-in in the Protégé tool, the integration of the ontology in the multi-agent system solution is much faster. This requires the implementation of the following steps:

1. Include/import the JADE abstract ontology into the Protégé.

   The first step is related to add the schema of the Abstract Ontology to the current ontology designed on the Protégé tool, as illustrated in Figure 15.

![Image of Protégé tool showing JADE Abstract Ontology](image)

   **Figure 15.** JADE Abstract Ontology for OntologyBeanGenerator

Looking the Figure 15, some of the classes, namely Concept, Predicate and AgentAction, which have already been defined can be noted. These classes or abstract models may be based on frames or OWL language. Therefore the classes that are defined in the project SimpleJADE-AbstractOntology.pprj must be imported, if based on frames; instead, if they are OWL ontologies, the OWLSimpleJADEAbstractOntology.owl must be imported.

2. Include/import the GRACE ontology into the same Protégé project.

   As second step, it is necessary to import the ontology designed for a specific domain. At this moment it is necessary to create new classes, which will support new concepts, such as AgentAction.

   Initially, an ontology was created without knowing the selected way for the implementation, but at this moment it is necessary to refine the resulted ontology. Figure 16 illustrates a fragment of the resulted matching of the GRACE ontology with the Abstract Ontology from the Ontology-BeanGenerator plug-in.
3. Exportation of the ontology Java classes.

This step can be performed by using several options provided by the OntologyBeanGenerator plug-in, as illustrated in Figure 17.

The available options are:
• J2ME: in the case the multi-agent system will run under an embedded system.
• J2SE (standard edition): in case the multi-agent system will run under a normal computer or smartphones (note that this option is compatible with the lighter version of JADE, named the JADE-LEAP).
• J2SE Java Bean: applied for cases similar to the previous one, but now using JavaBeans.

4. Import the ontology to be ready to be used.

After the exportation of the ontology, it is sometimes necessary to refine the exported Java classes by hand, due to some syntax errors and wrong reference class introduced during the automatic generation process.

Formerly the ontologies were built using the Ontology Frames plug-in, which was a stable version and not introducing errors in the generation process. However, in this work, the ontology was developed in the OWL language and consequently the proper plug-in for this kind of representation is the OWLSimpleJADEAbstractOntology, which does not provide the same correct results as the former one.

4.3. Implementation of the generated classes

In this work, the GRACE ontology was translated using the OntologyBeanGenerator plug-in to several Java classes as illustrated in Figure 18, following the FIPA specifications for the development of ontologies.

The first generated class is the GraceOntology, which represents the vocabulary and main concepts defined in the ontology. The following piece of code represents this class:

```java
public class GraceOntology extends jade.content.onto.Ontology {
```

4 A JavaBean is a Java object that is serializable and allows accessing to properties using getter and setter methods.
// ------------ vocabulary --------------
public static final String _SENDJOURNALDETAILS = "_sendJournalDetails";
public static final String _SENDJOURNAL = "_SendJournal";
public static final String _SENDOPERATION = "_sendOperation";
public static final String _SENDOPERATIONDETAILS = "_SendOperationDetails";
...
private static final long serialVersionUID = 690737088184824236L;
// A symbolic constant, containing the name of this ontology.
public static final String ONTOLOGY_NAME = "GRACE";
// The singleton instance of this ontology.
private static Ontology theInstance = new GraceOntology();
public static Ontology getInstance(){
    return theInstance;
}
private GraceOntology(){
    super(ONTOLOGY_NAME, BasicOntology.getInstance());
    try {
        // ---- adding Concept(s)
        ConceptSchema resourceSchema = new ConceptSchema(RESOURCE);
        add(resourceSchema, DefaultResource.class);
        ConceptSchema qualityControllerSchema = new ConceptSchema(OPERATION);
        add(OPERATION Schema, Default OPERATION.class);
        ...
        // ---- adding AgentAction(s)
        add(_SendOperationFinishedMachineSchema, Default_SendOperationFinishedMachine.class);
        AgentActionSchema _SendOperationSchema = new AgentActionSchema(_SENDOPERATION);
        add(_SendOperationSchema, Default_SendOperation.class);
        AgentActionSchema _SendJournalSchema = new AgentActionSchema(_SENDJOURNAL);
        ...
        // adding fields
        resourceSchema.add(RESOURCE_HASPROPERTY, propertySchema, 0, ObjectSchema.UNLIMITED);
        resourceSchema.add(RESOURCE_LOCATION, (TermSchema)getSchema(BasicOntology.STRING), 0, ObjectSchema.UNLIMITED);
        _SendOperationFinishedMachineSchema.add(_SENDOPERATIONFINISHEDMACHINE_SENDJOURNALDETAILS, journalDetailsSchema, 0, ObjectSchema.UNLIMITED);
        _SendOperationSchema.add(_SENDOPERATION_SENDOPERATION, operationSchema, 0, ObjectSchema.UNLIMITED);
    }
    }
}
Additionally, this class comprises four main parts. The first one is related to the vocabulary defined by the ontology, which comprises the establishment of the constants based on the concepts. The second part is related to adding the list of Concepts, e.g., the resource and Operation concept schemas. The third part is related to adding the AgentAction objects and the last part is related to the restrictions of the previous objects. Any restriction added on the ontology will have a direct translation to one of these several options, i.e., the minimum cardinality, the limitation, the optional/mandatory role, among others. If the validation on the content expression is not the correct one, an exception is thrown.

The second group of generated classes are related to the Java classes that specify the structure and semantics of the ontological objects, namely Concepts and AgentAction classes, and also included in the GRACEOntology package.

The Concept classes, such as Journal, JournalDetails, Resource, Material, ProcessPlan and Operation, are created by extending the class `jade.content.Concept`. The next code represents the interface of the Journal concept.

```java
public interface Journal extends jade.content.Concept {
    public void addJournalID(String elem);
    public boolean removeJournalID(String elem);
    public void clearAllJournalID();
    public Iterator getAllJournalID();
    public List getJournalID();
    public void setJournalID(List l);
    public void addStartDate(String elem);
    public boolean removeStartDate(String elem);
}
```

All the methods included in this kind of classes aiming at manipulating the attributes of the class should be declared. Some considerations in this approach must be taken: it is possible to declare all the attributes and then the setter and getter methods, but making this class as an interface can provide additional benefits.

In fact, previously, the OntologyBeanGenerator plug-in allowed the exportation of the designed ontology to pure Java classes by creating the getter and setter methods. The developers have noticed the advantage of using interfaces in Java. Since the OntologyBeanGenerator plug-in is implemented in Java, it was decided to join these two concepts. Briefly, interfaces serve as a contract, where can be specified which methods and classes are required to be implemented. When two objects are communicating, each one only needs to know the interface of the other. Anything beyond that leads to redundant issues. Therefore, the exported interfaces for the specific and subsequently implemented class, creates a rigid environment, consistent with the best programming practices. The next piece of code illustrates the implementation of the interface for the Journal concept.

```java
public class DefaultJournal implements Journal {
    private String _internalInstanceName = null;
    private List journalID = new ArrayList();
}
```
The AgentAction classes are classes representing the actions performed by the agents, such as _SendJournal, _SendJournalDetails, _SendOperation and _SendOperationFinishedMachine. Each class of the type AgentAction is represented by extending the jade.content.AgentAction class. The next code represents the _SendJournal action.

```java
public interface _SendJournal extends jade.content.AgentAction {
    // Protege name: http://jade.csel.it/beangenerator#_sendJournal
    public void add_sendJournal(Journal elem);
    public boolean remove_sendJournal(Journal elem);
    public void clearAll_sendJournal();
    public Iterator getAll_sendJournal();
    public List get_sendJournal();
    public void set_sendJournal(List l);
}
```

In the same way as occurred for the Journal concept, it is also necessary to declare the implemented Default_SendJournal class for the _SendJournal interface.

At this stage, all classes needed for the ontological model are created. However, as previously referred, some hand-made corrections were required due to the errors introduced by the used plug-in during the automatic translation process. The resulted implemented ontology (i.e. the generated Java classes) is now ready to be used by the GRACE agents.

**4.4. Usage of the GRACE Ontology**

The use of the Java classes (which represents the ontology) by the agents closes the several phases of the development of the GRACE ontology, started with the conceptualization, passing by the specification of the ontology schema and followed by its validation and implementation. Figure 19 illustrates the use of the ontology (generated from the ontology schema edited in Protégé and using the OntologyBeanGenerator plug-in) to support the interaction among distributed agents, where the agents use the same ontology.
(but different fragments of the ontology) to express the shared knowledge that is exchanged.

![Figure 19. Agents using ontologies to exchange knowledge](image)

The use of the ontology expressed in the Java classes by the agents, requires the registration of the ontology according to the selected codec. This process is necessary to be executed only once by each agent. Any codec has as purpose to support the language, in order to maintain the correct semantics and expression of terms. JADE has two basic types of codecs as a content of language, the SL codec and the LEAP codec.

When exporting through the `OntologyBeanGenerator` plug-in, the possibility to choose is given. This decision is made by the developer, choosing one of the two codecs. LEAP is more lightweight, but on the other hand, the content is more readable in SL. It can be said that LEAP is to be read and interpreted by computers, while the content of SL language structure is human readable. In this way, the main key to keep is to only use LEAP when there are strong memory limitations. It would be possible to choose a codec to support an XML-based, or even if it is the developer interests to implement its language, requiring a codec for it.

In this work, it is used SL and the next program illustrates how to register the ontology on the agents’ behaviours.

```java
private Codec codec = new SLCodec();
private Ontology ontology = GraceOntology.getInstance();
WaitingMessages(Agent aThis, boolean myGuiON, formRA_myGui){ ... } @Override
public void action() {
    // Register the language and ontology
    myAgent.getContentManager().registerLanguage(codec);
    myAgent.getContentManager().registerOntology(ontology);
```
Having registered the ontology, the agents can start using the ontology to represent the knowledge and also to send messages containing this knowledge. Considering the example of exchanging a message, several steps should be considered.

First, it is necessary to create the ontological message and send it. The next piece of code illustrates how to create an ontological message structure and send it to other(s) agent(s). In this case, a PA agent sends a message to a RA.

```java
//implementation of the _SendOperation
GRACE_Ontology.Operation operation = new DefaultOperation();
operation.addDuration(1); operation.addName(op.getOperation());
//action
_SendOperation sendOperation = new Default_SendOperation();
sendOperation.add_sendOperation(operation);
sendOntologyMessage(RA_RECEIVES_PROCESS_PLAN, msg.getSender(),
    sendOperation, ACLMessage.INFORM);
...
```

The key issue here is to fulfil the message attribute, namely the `setOntology()` and `setLanguage()` methods, and use the `FillContent()` method to fulfil the content of the message using the desired ontological object representing the knowledge that PA wants to exchange.

The second step is related to receive the ontological message. To properly receive the message, the agent needs to extract the contents with the `extractContent()` method. The extracted content of the message is then parsed by comparing the instance type with the type of value that is expected. In the following piece of code, the result of “if-then” tests determines which type of instance is the content of the message.

```java
ContentElement content = null;
try {
    content = myAgent.getContentManager().extractContent(msg);
    Concept action = ((Action) content).getAction();
}
```
if (action instanceof _SendOperation) {
    _SendOperation sendedOperation = (_SendOperation) action;
    GRACE_Ontology.Operation operation =
        (GRACE_Ontology.Operation)sendedOperation.get_sendOperation().get(0);
    journalDetails.setName(myAgent.getLocalName());
    ...
    }
    ...
}

In the Figure 20 it is possible to see an example of the sequence of messages exchanged between agents, using the Sniffer tool provided by the JADE framework.

![Figure 20. Excerpt from the process of sending the Processplan object from PA to RA](image)

The sequence of exchanged messages has the following meaning:

1) The RA agent named "A_Bearing_Insertion" receives the notification that the product is in front of the machine.

2) The RA agent sends a message to the PA agent named "011210007041 ", notifying that the product is ready to be executed.

3) The PA agent sends to the RA agent a message containing the information regarding the operation.

4) The RA sends a message to the IMA, notifying the start of the operation execution.

5) The RA agent informs the PA that the operation is already finished.

Figure 21 illustrates the content (structured as an ontology and not as string) of the exchanged ACL message between the PA and RA related to the moment 3). Note that if it had been chosen the LEAP codec instead the SL codec, the content field of the Figure 21 it will be empty (remember that the SL codec is human readable).
The content of this ACL message is more detailed as follows.

```plaintext
{action
  (agent-identifier
    :name A_Bearing_Insertion@169.254.114.78:1099/JADE
    :addresses (sequence http://Ricardo-PC:7778/acc))
  (_SendOperation
    :_sendOperation
    (sequence
      (Operation
        :duration (sequence 1)
        :name (sequence Opt_xpto_32_a))))}
```

The response to this ACL message, also captured by the Sniffer tool from the JADE framework, as illustrated as follows.

```plaintext
{action
  (agent-identifier
    :name "011210007041@169.254.114.78:1099/JADE"
    :addresses (sequence http://Ricardo-PC:7778/acc))
  (_SendOperationFinishedMachine
    :_sendJournalDetails
    (sequence
      (JournalDetails
        :startDate (sequence "Mon Mar 12 06:07:59 GMT 2012")
        :startTime (sequence "6:7:59")
        :endTime (sequence "06:08:04")
        :overallResult (sequence "1")
        :productID (sequence "011210007041")
        :endDate (sequence "2012-03-12")
        :name (sequence A_Bearing_Insertion))})
```
This example makes obvious the reason to use ontologies to represent the shared knowledge, because the agent which receives a message from another agent, must possess the schema to perform the parsing. These terms are no longer simple words and start to have semantic meaning.
5. Integration with legacy systems

An important issue in the multi-agent system is their integration in a computational ecosystem, which is already running in the production line at different control levels. In this way, this section describes the integration of the multi-agent solution with the legacy systems, namely the integration with quality control applications developed using LabView™ and the integration with the production database (to support the demonstration of the system using real production data).

5.1. Interface with LabView applications

In the GRACE multi-agent architecture each quality control station (QCS) has an associated QCA that is responsible to introduce intelligence and adaptation in the execution of the inspection operations. The interconnection between the QCA and the QCS, which can be a physical equipment or an operator, is a critical point in the integration of the multi-agent system infrastructure with the legacy systems.

5.1.1. Interface Strategy

The interface between the QCA and QCS is supported by mechanisms that allow the exchange of data using XML files. For this purpose, two XML templates are defined, as illustrated in Figure 22: one to send the data from QCA to the QCS and another one to exchange data from QCS and QCA.

The QCA provides the test plan to the QCS containing mainly the sequence of tests to be performed, the specifications of the components assembled in the product and the parameters for the physical equipment and the inspection algorithms, together with the cycle time that is the maximum allowed time to perform the operations. After the measurement procedure, the QCA receives the results comprising a global score that represents an overall result of all the performed tests and the detailed results of each test that are composed of the quality score, the measurement duration and its uncertainty.

In this work, the QCS is developed in LabVIEW™ [10] imposing specific constraints to be integrated with the Java applications, i.e. the agents. In order to enable the communication,
it is necessary to choose among different technological approaches. The first option is to use the "in process" approach, as shown in Figure 23, that uses JNI (Java Native Interface) to support the calling of LabVIEW™ routines by Java applications (note: it also supports the opposite situation). With the JNI it is possible to invoke Java classes, methods and access to properties, such as strings, from "native code", like C-based dynamic link libraries (.dll). In this way, it is possible to convert the java classes' files, and “Java Archive” files into “.NET Framework”. This is useful since LabVIEW™ supports the use of .NET.

![Figure 23. “in process” approach for the communication between Java and LabVIEW™.](image)

The second approach is the "out process", illustrated in Figure 23, which can be accomplished with Web services. Since Web services add a language-independent layer for distributed applications, the choice of Web services is the most obvious and the most flexible solution to integrate different and heterogeneous programming language applications into a single architecture [11].

![Figure 24. “out process” approach for the communication between Java and LabVIEW™ using web services](image)

In this case, the agent, implemented in Java, and the LabVIEW™ application will communicate by using the service-oriented principles, which are based on providing and using services. The services, which encapsulate the application’s functionalities, can be provided (and announced in a service registry) by the Java application (i.e. agent) or the LabVIEW™ application (i.e. QCS). When an application needs a specific service, it looks for it on the services’ registry by using an UDDI (Universal Description Discovery and Integration) mechanism, and then invoking the needed service.

In this work, and since the integration is focused in two pre-defined applications (which doesn’t require the need to discover the available services in the distributed system), only a
basic layer of the “out process” approach is implemented. This solution involves the use of sockets by TCP/IP protocol and avoids the need to implement a server to host the WSDL (Web Services Description Language) file and the UDDI file, making this negotiation layer lighter. Thus the proposed approach relies on opening two TCP/IP sockets: one for the communication from the QCA to the QCS and the other for the communication from the QCS to the QCA. For the experimental deployment, both the QCA and the quality control software run in the same machine, however this approach allows the two frameworks to run also on different machines.

In this way, the mechanisms to establish the connection and to communicate with the LabView™ applications are included in classes that are aggregated in the LabView package, illustrated in Figure 25.

![Figure 25. Classes used on GRACE to handle the interfaces with LabView applications](image)

The QCA establishes the connection with the LabView application running in the QCS by using the following piece of code.

```java
LabView_Server server = new LabView_Server("5555");
server.start();
LabView client = new LabView();
client.setClientConfigurations("127.0.0.1", "5551"); // i.e. LV Server
client.startClientConnection();
client.killClientConnection();
server.join();
```

In this process, initially a server and a client from the agent side are created, since the idea is to have a conversation in both directions. The connection requires the specification of the IP address and the port of the LabView server, and also the port to the agent server.

After the establishment of the connection, the QCA and QCS exchange information through XML files. The next extract of code is related to the exchange of a XML file, sent by QCA to the QCS through the created socket.

```java
String xmlFile = new ParserXml().loadFileToString("Input.xml");
String message = xmlFile;
```
If (message == null)
    message = "it is null";
}
Cliente = new Socket(this.clientIP, Integer.parseInt(this.clientPort));
saidaOS = cliente.getOutputStream();
saidaOS.write(message.getBytes());

Figure 26 illustrates an example of the XML structure containing the instructions for the inspection test plan sent by the QCA to the QCS, and the information provided by the QCS to the QCA containing the results from the execution of the inspection tests.

```xml
<xml version="1.0" encoding="utf-8">
<Input>
  <Test_Plan>
    <Cycle_Time>20</Cycle_Time>
    <Test1>
      <Part_Description>
        <Part1>
          <Color>Grey</Color>
          <Height>50</Height>
          <Width>10</Width>
        </Part1>
      </Part_Description>
      <Device1>
        <Device>cam0</Device>
        <CamExpTime>200</CamExpTime>
      </Device1>
      <Device2>
        <Device>light0</Device>
        <Color>Red</Color>
      </Device2>
    </Test1>
  </Test_Plan>
  <Additional_info />
</Input>
<xml version="1.0" encoding="utf-8">
<Output>
  <Overall_Diagnosis>60</Overall_Diagnosis>
  <Det_Res>
    <Test1>
      <Inspec_Res>80</Inspec_Res>
      <Meas_Duration>60</Meas_Duration>
      <Meas_Uncert>3.64</Meas_Uncert>
      <Devices>
        <Device1>
          <Opt_Param_A>100</Opt_Param_A>
        </Device1>
        <Device2>
          <Opt_Param_A>Blue</Opt_Param_A>
        </Device2>
      </Devices>
    </Test1>
    <Det_Res>
      <Additional_Info />
    </Det_Res>
  </Output>
</xml>
```

**Figure 26.** Example of the XML (input and output) files

### 5.1.2. Adaptation Rules

Adaptation rules in a QCA can be used to find the optimal parameters of a test plan when a new model of washing machines arrives in the production line, or to adapt the parameters of an already existing test plan in order to deal with drifts and local variations of the optimal solution. Each quality control task implements both kinds of adaptation rules, following the methodology introduced in [12].

In the following, two specific examples, namely the Drum Geometry Control Station (GAP) and the Vision Control Station (WU1) are addressed and the adaptation strategies of some of the test plan parameters shown.
The GAP QCS sends to the GAP QCA the following information (in addition to other):

- **GAP_RES**: result of the test (OK/KO).
- **GAP_MIN_VALUE**: minimum value measured by the QCS.
- **GAP_MAX_VALUE**: maximum value measured by the QCS.
- **GAP_MEAN_VALUE**: mean value measured by the QCS.

The GAP QCS receives from the GAP QCA the following information (in addition to other) in order to adapt the criteria for the selection of good and faulty items:

- **GAP_MIN_THRESH**: minimum value allowed for the GAP measurements.
- **GAP_MAX_THRESH**: maximum value allowed for the GAP measurements.
- **GAP_MAX_DELTA**: maximum value allowed for the difference between the maximum and the minimum value of the GAP measurements.

These parameters are strongly dependent on the model of the items produced and their value must be optimized. The GAP QCA is able to autonomously perform the adaptation of these parameters in order to create a new test plan or optimize an existing one.

Let suppose that a test plan already exists for the inspection task. The GAP QCA uses the following rules in order to adapt test plan parameters to local variations:

- If **GAP_RESULT** is **OK** then **GAP_MIN_VALUE**, **GAP_MAX_VALUE** and **GAP_MEAN_VALUE** are stored in the QCA local database. The database contains a maximum fixed number of results \( M \) for each model, which is managed using a First-In-First-Out (FIFO) logic.

- If the number of results contained in the DB for a specific model is significant \( R \geq P * M \) then the parameters for the selection of good and faulty items are calculated as follows:
  
  o  \[ \text{GAP\_MIN\_THRESH} = \text{mean}(\text{GAP\_MIN\_VALUE}) - k_{\text{min}} * \text{stdev}(\text{GAP\_MIN\_VALUE}) \]
  
  o  \[ \text{GAP\_MAX\_THRESH} = \text{mean}(\text{GAP\_MAX\_VALUE}) + k_{\text{max}} * \text{stdev}(\text{GAP\_MAX\_VALUE}) \]
  
  o  \[ \text{GAP\_MAX\_DELTA} = \text{mean}(\text{GAP\_MAX\_VALUE} - \text{GAP\_MIN\_VALUE}) + k_{\text{delta}} * \text{stdev}(\text{GAP\_MAX\_VALUE} - \text{GAP\_MIN\_VALUE}) \]

  where \( k_{\text{min}}, k_{\text{max}}, k_{\text{delta}} \) are constants defining the amplitude of the acceptance interval, \( R \) represents the results and \( P \) is a constant. The parameters \( M, P, k_{\text{min}}, k_{\text{max}}, k_{\text{delta}} \) can be configured by the Quality Manager.

- If **GAP_RESULT** is **KO** for a fixed number of consecutive items of the same model \( F \) then an alarm is sent to the human operator. If the operator does not confirm the results then the local database containing the previous results referred to the specific model is reset and the parameters are set to the default values.

A second example is related to the Vision Control Station that is located in the first part of the Washing Unit line. The Vision QCS sends to the Vision QCA the following information (in addition to other):
- **WU1_RES**: result of the test (OK/KO).
- **WU1_BELT_SIZE**: value of the belt size measured by the QCS.
- **WU1_BELT_POSITION**: value of the belt position referred to the shaft measured by the QCS.

The Vision QCS receives from the Vision QCA the following information (in addition to other) in order to adapt the criteria for the selection or good and faulty items:

- **WU1_BELT_MIN_THRESH**: minimum value allowed for the belt size measurements.
- **WU1_BELT_MAX_THRESH**: maximum value allowed for the belt size measurements.
- **WU1_BELT_POS_MIN_THRESH**: minimum value allowed for the belt position measurements.
- **WU1_BELT_POS_MAX_THRESH**: maximum value allowed for the belt position measurements.

Like for the aforementioned QCS of the GAP control, these parameters strongly depend on the model of the items produced and their value must be optimized. Let us suppose that a new model of washing machine arrives and the test plan is not available. The Vision QCA uses the following rules in order to find the optimal parameters for the new test plan:

- The belt thickness **WU1_BELT_SIZE** \((Th_i)\) and belt position **WU1_BELT_POSITION** \((BP_i)\) are calculated from the item \(i\) and stored in the QCA local database.

- Then the estimates of the optimal test plan parameters are updated as follows:
  
  a. \(Th_i\) is used to update **WU1_BELT_MIN_THRESH** \((Th_l)\) and **WU1_BELT_MAX_THRESH** \((Th_h)\) with the ±3 standard deviation interval around the mean value \(Th\); values outside the ±4 standard deviation interval are discarded as outliers.

  b. \(BP_i\) is used to update **WU1_BELT_POS_MIN_THRESH** \((BP_l)\) and **WU1_BELT_POS_MAX_THRESH** \((BP_h)\) with the ±3 standard deviation interval around the mean value \(BP\); values outside the ±4 standard deviation interval are discarded as outliers.

- From the \(M\) iteration on, a Confidence Level index \((CI)\) is calculated on the estimates of both \((Th_l, Th_h)\) and \((BP_l, BP_h)\) using the following equation:

\[
CI = (1-\sigma/X)\times100
\]

where \(\sigma\) is the standard deviation and \(X\) the average of the specific parameter both calculated over the last \(M\) iteration values. When the confidence level exceeds a predefined threshold \((CI >= CI_{opt})\), or in the case a skilled operator approves the estimated test plan parameters, the temporary test plan becomes an effective test plan.

The parameter \(M\) is an integer value, \(CI_{opt}\) is a percentage and they can be configured by the Quality Manager.
5.2. Interface with the production database

The interface to the production database, namely the GraDaCo, is necessary to allow the access to the real production data feeding the execution of the demonstration of the GRACE multi-agent system in an off-line manner (i.e. using the real production data and conditions but with historical data).

The first step in the interface between the agent and the production database is to establish the connection to the database when the agent starts its behaviour (included in the initialize() method triggered in the setup() method of the agent). For this purpose, initially, it is necessary to configure the database server in order to be possible to access from the exterior, to import the JDBC Driver 4.0 for SQL Server plug-in. The following piece of code represents the code used by a GRACE agent to connect to the GraDaCo database.

```java
package Interfaces.GraceDataBase;
...

class JDBC {
    private Connection conn = null;
    public Connection getConnection() throws Exception {
        try {
            String connectionUrl =
                    "jdbc:sqlserver://159.155.84.163:1433;databaseName=GraDaCo";
            conn = DriverManager.getConnection(connectionUrl, "user", "password");
        } catch (SQLException ex) {
            Logger.getLogger(JDBC.class.getName()).log(Level.SEVERE,
                    null, ex);
        } return conn;
    }
    public ResultSet query(String query) {
        try {
            Statement st = getConnection().createStatement();
            ResultSet rs = st.executeQuery(query);
            return rs;
        } catch (Exception ex) {
            Logger.getLogger(JDBC.class.getName()).log(Level.SEVERE,
                    null, ex);
            ex.printStackTrace();
            return null;
        }
    }
}
```

During the connection to the database it is important to specify the IP, port and server name of the database. Also, it is necessary to include the user and password fields.

After the establishment of the connection with the production database, the agent can perform queries to it to know production data. For this purpose, the agent uses the
query(String query) method, where query is the desired SQL query. This method returns the data resulted from the query.

The use of this query method to access data in the production database is illustrated considering that a RA agent wants to query the GraDaCo database to retrieve data related to the bearing insertion station. For this purpose, the following piece of code is used.

```java
DataAcessLayer dal_GraDaCo = new DataAcessLayer();
JournalDetails results = new JournalDetails();
results = dal_GraDaCo.operationResultAndTime("A_Bearing", DateUtils.now());
```

It was created a layer to generalize the implementation, where the developer only needs to explain by plain text which is the name of the station. Through this layer some robustness on the system is assured, because if some values of the database that identified the station change is not necessary to change everything, but only the last layer, which is represented in the next piece of code. In this case the “A_Bearing” station is identified by a couple of variables, namely by the IDSystem and IDStation, where system is 2100 and station is equal to 2.

```java
private JDBC dataBase = new JDBC();
journalDetails = this.operationResultOfThisTime(2100, 2, now);
result = dataBase.query("SELECT TOP 1 [DateSrc], [IDResult], [IDProduct]
FROM [GraDaCo].[dbo].[Results]
WHERE IDSystem=2100 and IDStation=2 and 
(([DateSrc] >= CONVERT(DATETIME, " + date + ", 102))");
```

The result of this query returns the data related to a specific product (only 1), namely the execution date and the results of the execution. Note that the retrieved object, i.e. the result of the operation extracted from the database, is an instance of the JournalDetails class.
6. Development of the Graphical User Interfaces

The Graphical User Interfaces (GUIs) are one way to provide an interface for the users to support the administration, management and monitoring of the system. Each type of agent provides different GUIs, since each type handles a particular set of information, and allows different types of interactions with users. In spite of providing different information, the GUIs of the several agents follow a common template of menus customized according to the agent’s particularities.

The use of a Java based framework, like JADE, to develop the system, offers the possibility of using Swing, a well-established toolkit to implement GUIs for desktop applications. Each type of agent in the GRACE system has its GUI implemented as an extension of the `javax.swing.JFrame` component. To perform the adequate interaction between both elements, the agent has a reference to the GUI form and this one have a reference to its owner agent.

The GUIs for all agents uses the information stored in its local database to feed the appropriate graphical components that are relevant to the users (see a previous section where it is detailed how the agent stores/retrieves data from its local database).

The GUI for the PTA agent, presented in Figure 29, allows the visualization of the production orders (managed by PA agents) launched for the execution at the production line, and its current status. For those that already have finished its execution, it is possible to consult the following information: identification, start date and end date, the result of the execution and the current state.

![Figure 27. Screenshot of the Product Type Agent](image)

The GUI for the PA agent, shown in Figure 28, offers the information regarding the moby identifier and the product model, and permits the on-line monitoring of the production of the product instance in a graphical manner.
For each operation belonging to the process plan, it is possible to visualize its current state, the identification of the resource responsible for its execution, the start and end date, and the results from its execution. For immediate identification of the position of the product in the production line, a dynamic graphical representation of a segment of three resources is presented on top of the GUI. In this way it is possible to see the current location of the product and where it goes.

The GUI for the RA agents, shown in Figure 29, enables the visualization of the current state of a resource disposed along the line.

The received notifications send by IMA agent as result of its trend analysis mechanism, are reflected in the RA’s GUI by the presentation of a warning message and the exhibition of the colour of the exhibited card (i.e. yellow, orange or red card) in the line that contains the product data (see Figure 30).
Figure 30. Screenshot of the Resource Agent executing an Operation after Applying Adaptation Procedures

The GUI of the IMA agent, presented in Figure 31, is useful to provide a global perspective of the entire production system (or part of it if considering different IMAs). The GUI for the IMA agent presents a global view of the system taking advantage of the data collection performed over the time for global adaptation/optimization and stored in the IMA’s SQL database.

Figure 31. Screenshot of the Interdependent Meta Agent

The graphical representation of the production line was constructed using `javax.swing.JLabel` elements, permitting the convenient presentation of the detected errors in the processing stations using the yellow, orange and red colours. Two `javax.swing.JTable`
elements were used to offer a table for the presentation of the products in the production line and another one to retrieve data concerning products already completed. The first table reflects immediately the data received by the IMA agent and the second one is fed by the result of a query performed over the IMA’s database.

The refreshing of the information showed in the tables of the IMA’s GUI is achieved by the use of the `refresh` method:

```java
public void refresh(JournalDetails journalDetails) {
    DefaultTableModel dtm = (DefaultTableModel) now_table.getModel();
    ...
    Object[] newline = {
        journalDetails.getProductID(),
        setLocation(journalDetails.getStation_Name(),
        setProductState(journalDetails.getState()),
        getStationName(journalDetails.getStation_Name()),
        journalDetails.getStartTime(),
        setProductState(journalDetails.getState()),
        setResult(journalDetails.getOverallResult()),
    };
    boolean flag = true;
    for(int jdi = 0; jdi < dtm.getRowCount(); jdi++) {
        String key = dtm.getValueAt(jdi, 0).toString();
        if(key.equals(journalDetails.getProductID())) {  
            dtm.setValueAt(journalDetails.getProductID(), jdi, 0);
            ...
            dtm.setValueAt(journalDetails.getStartTime(), jdi, 3);
            ...
            flag = false;
        }
    }
    if(flag) {
        dtm.addRow(newline);
        ...
    }
    ...
}
```

Note that a similar strategy is used to refresh the tables presented in other GUIs belonging to other types of agents.

To achieve the correct graphical representation of the current state of the production line an appropriate data structure to hold the current state information is considered as shown in the following code fragment.

```java
public class formIMA extends javax.swing.JFrame {
    ...
}
private final IMA myAgent;
...
public static boolean errors[]=new boolean[NUMBER_OF_RESOURCES + 1];
private ArrayList<String>[] segmentBuffers =
    (ArrayList<String>[]) new ArrayList[2 * NUMBER_OF_RESOURCES + 2];
    // |0=null|1=c1|2=m1|3=c2|4=m2|...|27=c14|28=m14|29=c15|...
... private javax.swing.JLabel[] resources =
    new javax.swing.JLabel[NUMBER_OF_RESOURCES + 1];
private javax.swing.JLabel[] arrows =
    new javax.swing.JLabel[NUMBER_OF_RESOURCES + 2];
...

The representation of the production line in the GUI considers that the production line is constituted by segments (namely, resources and arrows for the situation where a product is between resources). The methods dealing with the segments set or unset, the first one shown in the below piece of code, corresponds to a set operation corresponding to the arrival of the product represented by the productID to the segment (resource or arrow) represented by the index parameter:

private void setSegment(int index, String productID){
    if(index % 2 == 0){
        if(errors[index / 2] == false){
            resources[index / 2].setBackground(Color.GREEN);
        }
        resources[index/2].setIcon(new javax.swing.ImageIcon(getClass().getResource("/imagens/gears.gif")));
    } else{
        arrows[(index+1)/2].setIcon(new javax.swing.ImageIcon(getClass().getResource("/imagens/arrow1.gif")));
    }
    segmentBuffers[index].add(productID);
    String s = "<html>";
    for(int i = 0; i <= segmentBuffers[index].size() - 1; i++){
        s += segmentBuffers[index].get(i);
        if(i != segmentBuffers[index].size() - 1){
            s += "<br>";
        }
    }
    if(index % 2 == 0){
        resources[index / 2].setToolTipText(s);
    } else{
        arrows[(index + 1) / 2].setToolTipText(s);
    }
}
7. Lessons learned and Conclusions

This deliverable describes the main issues related to the implementation of the GRACE multi-agent system infra-structure specified in the deliverable 1.3. This infra-structure was implemented by using the JADE framework for the development of agent-based solutions, which provides an integrated environment for the development of such system, with less complexity and reduced effort and time-consuming.

The skeleton of the several GRACE agents were implemented, namely the behaviours structure of each agent, the ontology schema for the knowledge representation, the interaction patterns supported by FIPA protocols, the designed ontology, and the integration with legacy systems, such as LabView applications running in quality control stations and the production database. Several GUIs were also implemented to support an easy interaction with the users.

The multi-agent system infrastructure was intensively tested using historical real production data from the GraDaCo database, aiming at testing and correcting mistakes and bugs during the development process.

At this stage, the multi-agent system infra-structure is ready to accommodate the adaptation mechanisms, developed in tasks 2.2, 2.3 and 3.3 aiming at providing adaptation and optimization based on the integration of the quality and process control, and to be integrated with process and quality control devices. These issues will be performed in the task 5.1 of the GRACE project that is related to the integration and customization towards the demonstration.
References