Tutorial on UML and XML specifications in DCaseLP

To show how a MAS prototype can be semi-automatically created starting from the specification of its architecture and of the interactions that take place among agents, we will go through, step by step, the creation and implementation of a small MAS prototype whose components are initially analysed and designed using both UML and an XML-based representation of interaction protocols, and are then translated into an executable piece of code.

The MAS we are about to create will represent a simple distributed fruit marketplace where there are agents willing to sell fruit and agents wishing to buy it. Agents that act as mediators between sellers and buyers, namely that are able both to send (to smaller buyers) and to buy (from bigger sellers), are also considered.

1 Step 1: Role modelling

The first step to face, is to define the roles to be played within the MAS and the interactions taking place among roles. We use UML to express the interaction protocols, and FIPA ACL to define messages.

In our MAS, we assume that sellers propose to buy some kind of fruit to buyers (PROPOSE performative). Buyers can either refuse (REJECT_PROPOSAL performative) or accept (ACCEPT_PROPOSAL performative) the proposal, according to some internal condition that is not stated at this specification level.

This protocol has been standardised by FIPA and is known as FIPA Propose Interaction Protocol (see Figure 1). In our case, the roles to be played are Seller instead of Initiator, and Buyer instead of Participant.

Since the UML Modelling Tool that we use into DCaseLP, ArgoUML (`http://www.argouml.org`), does not support the specification of Sequence Diagrams (from which Protocol Diagrams derive), we cannot draw our protocol diagram using

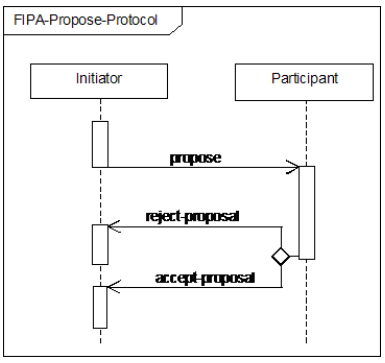

Figure 1: FIPA Propose Interaction Protocol.

ArgoUML[1]. Instead, we must directly express the protocol using the XML-based intermediate format.

The structure of the XML-based intermediate format for protocol diagrams is the following:

```
<protocoldiagram>
   <role>
   ..........
   </role>
   <role>
   ..........
   </role>
</protocoldiagram>
```

For each role that has been identified, the set of sent and received messages must be specified. Since we have two roles, Buyer and Seller, the protocol diagram will look like the following:

```
<protocoldiagram>
   <role>
      <name>Seller</name>
      <msgs>
      ....
      </msgs>
   </role>
   <role>
      <name>Buyer</name>
      <msgs>
      ....
      </msgs>
   </role>
</protocoldiagram>
```

Let us consider the Seller role first. It starts its interaction with the buyer by sending to him a message whose performative is PROPOSE (the content of the `<msg>` tag, characterised by Seller, within the `<sender>` tag, Buyer, within the `<receiver>` tag, and PROPOSE, within the `<act>` tag).

```
<msg>
   <sender>Seller</sender>
   <receiver>Buyer</receiver>
   <act>PROPOSE</act>
</msg>
```

After that, two (mutually exclusive) events may happen: either the Seller receives a REJECT_PROPOSAL, or it receives an ACCEPT_PROPOSAL. The main thread of the Seller agent must then be splitted into two subthreads (each one described by the `<thread>` tag), connected by means of a xor connective (`<xor-thread>` tag).

[1] If we were able to draw the protocol diagram using ArgoUML, we could export it into XMI and obtain, from the XMI code, the specification in our XML-based intermediate format in an automatic way by using the `Specif2Code` program. This is the approach we use to translate Agent and Architecture diagrams, that can be modelled using ArgoUML, and can thus be exported into XMI and automatically translated into our XML-based intermediate format.

```
<xor-thread>
   <thread>
      <msg>
      ....
      </msg>
   </thread>
   <thread>
      <msg>
      ....
      </msg>
   </thread>
</xor-thread>
```

Inside each thread, the reception of one between the REJECT_PROPOSAL and ACCEPT_PROPOSAL messages must be included:

```
<role>
  <name>Seller</name>
  <msgs>
    <msg>
      <sender>Seller</sender>
      <receiver>Buyer</receiver>
      <act>PROPOSE</act>
    </msg>
    <xor-thread>
       <thread>
          <msg>
            <sender>Buyer</sender>
            <receiver>Seller</receiver>
            <act>REJECT_PROPOSAL</act>
          </msg>
      </thread>
      <thread>
          <msg>
            <sender>Buyer</sender>
            <receiver>Seller</receiver>
            <act>ACCEPT_PROPOSAL</act>
          </msg>
      </thread></xor-thread>
    </msgs>
</role>
```

The Buyer role first waits for a message from the Seller role, with PROPOSE performative.

```
<role>
   <name>Buyer</name>
   <msgs>
     <msg>
     <sender>Seller</sender>
     <receiver>Buyer</receiver>
```

```
      <act>PROPOSE</act>
      </msg>
      ....
    </msgs>
</role>
```

After that, it may send one between REJECT_PROPOSAL and ACCEPT_PROPOSAL messages to the Seller (`<xor-send>` tag, with as many `<act>` tags as the mutually exclusive messages to be sent)

```
<xor-send>
   <sender>Buyer</sender>
   <receiver>Seller</receiver>
   <act>REJECT_PROPOSAL</act>
   <act>ACCEPT_PROPOSAL</act>
</xor-send>
```

The final specification of the protocol can be found in the `DCaseLP\UMLInJADE\Tutorial\MarketPlaceXMLIntermedFormat\protocolDiagr.txt` file.

2 Step 2: Agents' class modelling

Once the role model is well understood, the developer needs to address the following:

1. which roles to assign to each agent class (not meant as a Java class!!!) in the architecture diagram;
2. the number of istances of each agent class (in the agent diagram) that is required for the given application.

Architecture and agent diagrams can be defined directly in ArgoUML, and then exported into XMI. An "XMI → XML" translator will generate the XML-based intermediate format for both diagrams, and another "XML → code" translator will use the agent and architecture specifications in intermediate formats, together with the one defining the interaction protocol, to generate Jess and Java code.

The architecture diagram for our example looks like in Figure 2: there are three different agent classes, fruitSeller, fruitBuyer, and fruitExchanger, each one characterised by an attribute (programFruitSeller, programFruitBuyer, and programFruitExchanger, respectively) that, during the translation process, will be used to identify the name of the Jess program for each class. Note that here, class means the type of an agent, and has no relationship with Java classes at all, althought at the end of the translation process, Java classes will be created in order to integrate jess agents into Jade. Each agent class is characterised by the `<<class>>` stereotype, and it is related to a role class (class with stereotype `<<role>>`, in our example Buyer and Seller) by a dependency relation named `plays_role`. A class with stereotype `<<architecture>>` is used to specify which kind of agent architecture (BDI, reactive, etc) characterises the agent class. For the moment, only one architecture (jessArch, meaning that the agents are rule-based declarative agents whose behaviour is described by means of jess rules) is implemented. The dependency relation between the agent class and the architecture class is named `has_architecture`. The ArgoUML architecture diagram can be found in the directory `DCaseLP\UMLInJADE\`

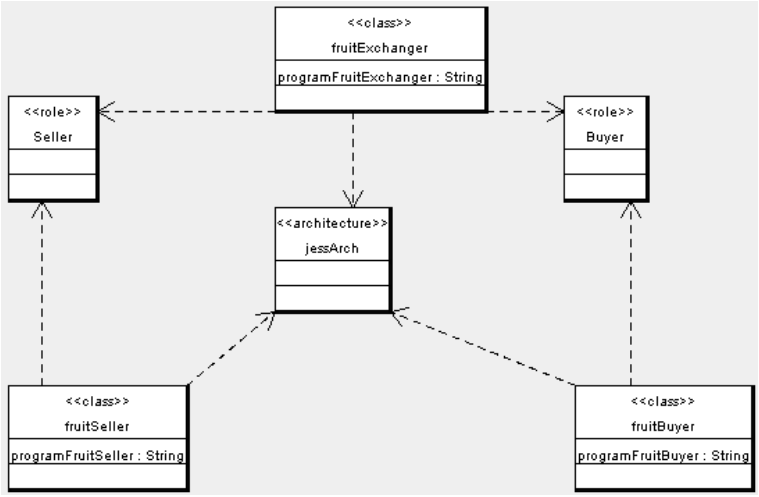

Figure 2: Architecture Diagram.

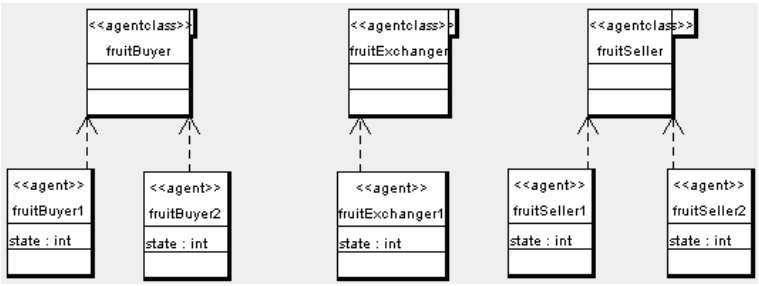

Figure 3: Agent Diagram.

`Tutorial\MarketPlaceArgoUML\ArgoUMLProjects`, under the name of `archDiagr.zargo`.

Finally, the agent diagram for our example looks like in Figure 3: there are the three agent classes already defined in the architecture diagram (here, the stereotype is `<<agentclass>>`) together with their instances (with stereotype `<<agent>>`, and with one attribute named state). The dependency relation between instances of agents, and agent classes, is named `instance_of`. The ArgoUML agent diagram can be found in the directory `DCaseLP\UMLInJADE\Tutorial\MarketPlaceArgoUML\ ArgoUMLProjects`, under the name of `agentDiagr.zargo`.

Once the two diagrams above have been opened using ArgoUML (or, in alternative, they have been defined from scratch, but always using ArgoUML), they can be exported into XMI (select the "File → save project as" option from the menu offered by ArgoUML, and then choose ".xmi" as file type). The two exported diagrams of our example can be found in the `DCaseLP\UMLInJADE\Tutorial\ MarketPlaceArgoUML` directory, with `archDiagr.xmi` and `agentDiagr.xmi` names.

3 Step 3: Generating the code from the UML and XML specifications

Once the agent and architecture diagrams have been modelled as UML class diagrams using ArgoUML, and then have been exported into XMI[2], and the interaction protocol has been defined in our XML-based intermediate format, the generation of the code can start.

1. Make sure that the requirements for running the translation program, described in the UMLInJADE-readme file, in the `DCaseLP\UMLInJADE` directory, are met.

2. Make sure that the directories `DCaseLP\UMLInJADE\jacode`, `DCaseLP\UMLInJADE\jecode`, and `DCaseLP\UMLInJADE\intermed` are empty.

3. From a shell, type in the command `java UMLInJADE.Specif2Code`. The command should work from any directory, since you are supposed to have correctly modified your classpath system variable by adding the path to DCaseLP to it.

4. A window asking to select a file from the XSL directory appears. Select ANY file from the `DCaseLP\UMLInJADE\XSL` directory (note that this is a quick way to select the entire XSL directory, not the specific file, so it does not matter which file you choose, provided that it is in the right directory).

5. A window asking if there are XMI diagrams in the specification of the MAS appears. Since in our example the agent and architecture diagrams are defined in XMI, answer yes.

6. A window asking how many protocol diagrams there are in the system appears; type 1, then press the DONE button, and then select the protocol diagram in XML intermediate format (`DCaseLP\UMLInJADE\Tutorial\MarketPlaceXMLIntermedFormat\protocolDiagr.txt`).

7. A window asking if there is an architecture diagram appears; answer yes, and then select the XMI architecture diagram (`DCaseLP\UMLInJADE\Tutorial\MarketPlaceArgoUML\archDiagr.xmi`).

8. A window asking if there is an agent diagram appears; answer yes, and then select the XMI agent diagram (`DCaseLP\UMLInJADE\Tutorial\MarketPlaceArgoUML\agentDiagr.xmi`).

9. The program should terminate. The directories `DCaseLP\UMLInJADE\jacode`, `DCaseLP\UMLInJADE\jecode`, and `DCaseLP\UMLInJADE\intermed`, that were initially empty, should now contain the generated code.

[2]Note that it is possible to define the agent and architecture diagrams directly using our XML-based intermediate format, whose syntax is described in the UMLInJADE-Manual, in the `DCaseLP\UMLInJADE\Manual` directory.

4 Step 4: Completing the Jess code

The Jess code characterising the behaviour of the instances of the fruitBuyer, fruitSeller and fruitExchanger agent classes (that can be found in the `DCaseLP\UMLInJADE\jecode` directory, within the programFruitBuyer, programFruitSeller and programFruitExchanger files respectively), is the only code that needs to be manually completed in order to run the simulation. The reason of this need depends on the nature of the specifications given in the modelling steps (1 and 2), where some details necessary for running the simulation are missing. In particular, the interaction protocol specifies neither under which conditions one message should be sent, nor which actions (apart from sending and receiving messages) the agent should take after a communicative action. These conditions and these actions must be added to the generated code by the developer. Also the initial state of the agent should be added to the code.

The generated code has a part that defines the functions available to the Jess agent, followed by a part containing the class-specific rules that respect the interaction protocol and architecture diagrams, and vary from agent class to agent class. Finally, the place where putting the initial facts of the agent follows.

4.1 Completing the programFruitBuyer.clp program

Let us consider the fruitBuyer class: its initial knowledge may be defined by the Jess statement

```
; *************** MAIN ********************

(deffacts buyerFacts
   (amount "banana" 20)
   (amount "apple" 0)
)

(reset)
```

meaning that the current amount of apples is 0 Kg and the amount of bananas is 20 Kg. The two facts above, must be written in the main section, and the `(reset)` fact, that is automatically generated by the translation program, must be kept where it is (otherwise, facts and rules cannot be used by the agent).

Let us now consider the rules that govern the behaviour of the buyer agent, and that have been automatically generated from the high level specifications.

The first rule of the fruitBuyer class agent (identified by a `defrule` Jess statement) is the following one:

```
(defrule Buyer_1 ; additional conditions
 => (wait_msg "PROPOSE" "Seller")
 (assert (state Buyer_1 ?cid))
; additional actions
(retract-string "(message \"PROPOSE\" \"Seller\")"))
```

This rule allows the fruitBuyer agent to stay idle waiting for a message coming from an agent playing the Seller role. Jess comments start with a `;`, so the places where putting additional conditions and additional actions can be easily identified by the `; additional conditions` and `; additional actions` comments.

We might add three additional actions that retrieve the content of the message `(bind ?current_content (get_content ?*msg*))`, assert it `(assert (current_content ?current_content))`, and print a message on the standard output `(printout t crlf crlf "The buyer received a PROPOSE " ?current_content " from an agent playing the Seller role" crlf)`. The resulting rule is the following one:

```
(defrule Buyer_1 ; additional conditions
 => (wait_msg "PROPOSE" "Seller")
 (assert (state Buyer_1 (get_cid ?*msg*)))
 (retract-string "(message \"PROPOSE\" \"Seller\")")

 (bind ?current_content (get_content ?*msg*))
 (assert (current_content ?current_content))
 (printout t crlf crlf "The buyer received a PROPOSE "
 ?current_content " from an agent playing the Seller role"
 crlf)
)
```

Note that we prefer to trace the message exchange by printing information about it to the standard output, rather than using the JADE Sniffer agent. The reason for our choice is simple: the Sniffer Agent requires some time to be activated, and when it starts sniffing the messages that the agents exchange, the communication among agents is already over, so no messages can be sniffed by it!

The second rule, instead, looks like the following:

```
(defrule Buyer_2_1
 (state Buyer_1 ?cid)
 ; conditions
 =>
 (bind ?content MSG CONTENT)
 (send (ACLMessage (communicative-act REJECT_PROPOSAL)
 (sender Buyer) (receiver Seller) (conversation-id ?cid)
 (content ?content)))
 (assert (state Buyer_2_1 ?cid))

 (retract-string (str-cat "(state Buyer_1 " ?cid ")"))

 ; actions
)
```

We may modify it so that it is fired only when the content of the last received message, asserted by means of the previous rule, is `?content`, (`(current_content ?content)` in the "conditions" part of the rule), and the buyer possesses an amount of `?content` greater than zero (`(amount ?content ?X&:(> ?X 0))` in the "conditions" part of the rule). In this case, a REJECT_PROPOSAL with content `?content` is sent to the sender agent. Note that we deleted the line `(bind ?content MSG CONTENT)` from the "actions" part of the rule, since the `?content` has been already bound. We also added an action for printing a message to the standard output.

```
(defrule Buyer_2_1
```

```
 (state Buyer_1 ?cid)

 (current_content ?content)
 (amount ?content ?X&:(> ?X 0))
 =>

 (send (ACLMessage (communicative-act REJECT_PROPOSAL)
 (sender Buyer) (receiver Seller) (conversation-id ?cid)
 (content ?content)))(assert (state Buyer_2_1 ?cid))
 (retract-string (str-cat "(state Buyer_1 " ?cid ")"))

 (printout t crlf crlf "The buyer sent a REJECT_PROPOSAL "
 ?content " to an agent playing the Seller role" crlf)
)
```

In a similar way, we can complete the third rule so that it is fired only when the amount of the fruit proposed by the Seller agent, and stored in the `(current_content ?content)` fact, is equal to zero. In this case, the proposal can be accepted.

```
(defrule Buyer_2_2
 (state Buyer_1 ?cid)

 (current_content ?content)
 (amount ?content ?X&:(< ?X 1))
 =>
 (send (ACLMessage (communicative-act ACCEPT_PROPOSAL)
 (sender Buyer) (receiver Seller) (conversation-id ?cid)
 (content ?content)))
 (assert (state Buyer_2_1 ?cid))
 (retract-string (str-cat "(state Buyer_1 " ?cid ")"))
 (printout t crlf crlf "The buyer sent an ACCEPT_PROPOSAL"
 ?content " to an agent playing the Seller role" crlf)
)
```

The last rule is used to join the two threads of the conversation, and does not need to be changed.

```
(defrule Buyer_2
 (state Buyer_2_1 ?cid)
 =>
 (assert (state Buyer_2 ?cid ))
 (retract-string (str-cat "(state Buyer_2_1 " ?cid " )"))
 )
```

The modified code can be found in the directory `DCaseLP\UMLInJADE\Tutorial\CompletedJecode`.

4.2 Completing the programFruitSeller.clp program

The seller agent has no initial knowledge (thus, its knowledge base is empty and does not need to be modified), and its behaviour is characterised by only one rule, used to

propose to buy apples to an agent playing the “Buyer” role. The modified rule is the following one, where the content of the message has been bound to the “apple” atom, the performative has been set to “PROPOSE”, and a “printout” action has been added for monitoring purposes.

```
(defrule Seller_1 ; additional conditions
 =>
 (bind ?cid (newcid)) (bind ?*addr* (read_addr "Buyer"))
 (if (eq nil ?*addr*)
      then (bind ?*addr* (fetch_addr "Buyer")))
 (bind ?content apple)
 (send (ACLMessage (communicative-act PROPOSE)
 (receiver ?*addr*) (protocol "Seller")
 (conversation-id ?cid)
 (content ?content))) (assert (state Seller_1 ?cid))

 (printout t crlf crlf "The seller sent a
 PROPOSE " ?content " to an agent playing the Buyer
 role" crlf)
)
```

The modified code can be found in the directory `DCaseLP\UMLInJADE\Tutorial\CompletedJecode`.

4.3 Completing the programFruitExchanger.clp program

The fruit exchanger agent initially possesses 40 Kg of bananas and 0 Kg of apples. This information is added to the fruit exchanger belief set by means of the following Jess statement

```
(deffacts fruitExchangerFacts
 (amount "banana" 40)
 (amount "apple" 0)
)
(reset)
```

Since the fruit exchanger behaves like both a seller, and a buyer, its behaviour is characterised by the rules that we have already met when completing the Jess code of both the seller and the buyer agents. The only difference is that we must manually include the JADE address of the buyer into one rule, namely the rule where the fruit exchanger behaves like a seller, and proposes to buy something to an agent that plays the “Buyer” role. If we do not modify this rule, when looking for an agent playing the “Buyer” role, the exchanger finds its own address first, and so it sends the PROPOSE message to itself.

So, let us suppose that the fruit exchanger wants to propose to buy bananas to an agent playing the “Buyer” role. The following rule must be used:

```
(defrule Seller_1 ; additional conditions
 =>
 (bind ?cid (newcid))
```

```
 (bind ?my_addr (my_addr))
 (printout t crlf crlf "The fruit exchanger
 has address " ?my_addr crlf)
 (bind ?*addr* (read_addr "Buyer"))
 (if (eq nil ?*addr*)
     then (bind ?*addr* (fetch_addr "Buyer")))
 (printout t crlf crlf "The fruit exchanger fetched
 the address " ?*addr* crlf)

 (bind ?content banana)
 (send (ACLMessage (communicative-act PROPOSE)
 (receiver b@klimt:1099/JADE) (protocol "Seller")
 (conversation-id ?cid)
 (content ?content))) (assert (state Seller_1 ?cid))
 (printout t crlf crlf "The fruit exchanger sent a
 PROPOSE " ?content " to an agent playing the Buyer
 role" crlf)
)
```

In this rule, the lines

```
 (bind ?my_addr (my_addr))
 (printout t crlf crlf "The fruit exchanger has address "
  ?my_addr crlf)
 (bind ?*addr* (read_addr "Buyer"))
 (if (eq nil ?*addr*) then
     (bind ?*addr* (fetch_addr "Buyer")))
 (printout t crlf crlf "The fruit exchanger fetched
 the address " ?*addr* crlf)
```

in the body are used to show that the fruit exchanger fetches its own address from the JADE Directory Facilitator.

For this reason, the address of the other agent in the system that plays the "Buyer" role, namely the buyer agent, must be explicitly put in the code of the exchanger:

```
 (receiver b@klimt:1099/JADE) (protocol "Seller")
 (conversation-id ?cid)
```

The address (in our example, `b@klimt:1099/JADE`) is easily obtained from the agent's name (`b` in our example; the name is decided when the simulation of the MAS is run - you might use `b`, as we do, for making things easier), followed by `@`, followed by the name of the computer where the agent is located (the name of your computer; in our example, the name was `klimt`), followed by `:1099/JADE`.

The other rules of the fruit exchanger are the same as the buyer's ones.

The modified code of the fruit exchanger can be found in the directory `DCaseLP\UMLInJADE\Tutorial\CompletedJecode`, but you cannot use it as it is!!!! You must edit it, and change the buyer's address so that it contains the name of your computer.

5 Running the simulation of the MAS

1. If you copied the Jess programs from the `DCaseLP\UMLInJADE\Tutorial\CompletedJecode` directory to the `DCaseLP\UMLInJADE\jecode` one, remember to edit the `programFruitExchanger.clp` program since you cannot use it as it is!!!! You must edit it, and change the buyer's address so that it contains the name of your computer (see above).

2. Move to the `DCaseLP\UMLInJADE\jacode` directory, and compile the Java sources by typing `javac *.java`:

   ```
   C:\DCaseLP\UMLInJADE\jacode> javac *.java
   ```

3. Remain in the `DCaseLP\UMLInJADE\jacode` directory, and start the execution of a JADE platform containing your agents (named `e`, exchanger, `b`, buyer, and `s`, seller) by typing `java jade.Boot e:fruitExchanger b:fruitBuyer s:fruitSeller`:

   ```
   C:\DCaseLP\UMLInJADE\jacode>java jade.Boot
   e:fruitExchanger b:fruitBuyer s:fruitSeller
   ```

   Note that this command can work only if you have already installed JADE (and, obviously, Java), and if the path to JADE has been added to your classpath. See the "UMLInJADE-readme" file for more details.

- Now the execution of the JADE platform starts. First, some messages from the JADE platform itself are printed to the standard output:

```
2-nov-2005 12.42.58 jade.core.Runtime beginContainer
INFO: ---------------------------------
    This is JADE 3.3 - 2005/03/02 16:11:05
    downloaded in Open Source, under LGPL
    restrictions,
    at http://jade.cselt.it/
-----------------------------------------
2-nov-2005 12.42.59 jade.core.BaseService init
INFO: Service jade.core.management.AgentManagement
initialized
2-nov-2005 12.42.59 jade.core.BaseService init
INFO: Service jade.core.messaging.Messaging
initialized
2-nov-2005 12.42.59
jade.core.messaging.MessagingService boot
INFO: MTP addresses:
http://klimt:7778/acc
2-nov-2005 12.42.59 jade.core.BaseService init
INFO: Service jade.core.mobility.AgentMobility
initialized
2-nov-2005 12.42.59 jade.core.BaseService init
INFO: Service jade.core.event.Notification
initialized
```

```
2-nov-2005 12.42.59 jade.core.AgentContainerImpl
joinPlatform
INFO: ----------------------------------------
Agent container Main-Container@JADE-IMTP://klimt
is ready.
--------------------------------------------
```

Then, the messages from the agents are shown:

```
The fruit exchanger has address e@klimt:1099/JADE

The fruit exchanger fetched the address
e@klimt:1099/JADE

The fruit exchanger sent an PROPOSE banana to
an agent playing the Buyer role

The buyer received a PROPOSE banana from
an agent playing the Seller role

The buyer sent a REJECT_PROPOSAL banana
to an agent playing the Seller role

The seller sent an PROPOSE apple to an
agent playing the Buyer role

The fruit exchanger received a PROPOSE
apple from an agent playing the Seller role

The fruit exchanger sent an ACCEPT_PROPOSAL
apple to an agent playing the Seller role
```

- The messages exchanged by the agents, and traced by means of ad-hoc sentences printed to the standard output might also be sniffed by the Sniffer agent offered by JADE – launching `jade.Boot` followed by the `-gui` option – provided that some instructions for delaying the beginning of the agents' execution was added to the agents' code, since the Sniffer agent requires some time to be activated and ususlly, starts when the conversation is already over.
- The traces of the message exchange demonstrate that the agents behave as expected. In fact, the fruit exchanger proposes to the buyer agent to buy bananas. The buyer agent does not need bananas, since it already possesses 20 Kg of this kind of fruit, and rejects the proposal. After that, the seller agent sends a proposal to the fruit exchanger, to buy apples. The fruit exchanger does not possess any apple, and then it accepts the proposal.
- Note that the order in which agents are activated may change due the order in which they are loaded into the JADE platform, and the agents playing a given role, that are retrieved from the JADE Directory Facilitator, may also change. This means that the messages exchanged within your MAS simulation, may differ from those that we got in this execution run. For example, it may happen that

the seller agent sends the proposal to buy apples to the buyer agent, and not to the fruit exchanger. In this case, the buyer agent would accept the proposal too, since it has no apples.