COS30018 – Intelligent Systems

Week 3 tutorial

This week we will cover the following items:

- Administration control and runtime management with JADE
- Implement the "Book Trading" platform example with JADE
    - Create agents
    - Implement agent behaviours
    - Implement agent communication
- Other JADE practices

1. Administration control and runtime management

Launching JADE runtime means creating a Container. This container can contain agents that can be started directly at container startup time or later (e.g., through the JADE management GUI). Moreover, the first container launched in the platform becomes the **Main Container** of that platform.

In order to have extra control of the JADE environment (such as the host and the port being used when a Main Container is generated), please refer to a tutorial in the file **"Administration control"** on Canvas. This is very helpful when you want to manipulate with distributed/remote platforms in JADE because you need to understand the host/port so that you can connect multi platforms together.

In that tutorial, you will know how to specify the port, the host name/address, the container name, and the platform name.

2. "Book Trading" example

This part introduces a simple example of "Book Trading" platform to illustrate the steps required to develop agent-based applications with JADE. The scenario considered in this example includes some agents selling books and other agents buying books on behalf of their users.

Platform description: Each buyer agent receives the title of the book to buy (the "target book") as a command line argument and periodically requests all known seller agents to provide an offer. As soon as an offer is received the buyer agent accepts it and issues a purchase order. If more than one seller agent provides an offer the buyer agent accepts the best one (lowest price). Having bought the target book the buyer agent terminates.

Each seller agent has a minimal GUI by which the user can insert new titles (and the associated price) in the local catalogue of books for sale. Seller agents continuously wait for requests from buyer agents. When asked to provide an offer for a book they check if the requested book is in their catalogue and in this case reply with the price. Otherwise they refuse. When they receive a purchase order they serve it and remove the requested book from their catalogue.

The complete source code of this example is provided on Canvas.

a. Creating JADE agents:

JADE Agent

The main method to initialize a JADE agent is the `setup()` method. Therefore, any agent class that extends `jade.core.Agent` should implement the `setup()` method. Within the `setup()` method we can add "behaviours" within which the agent tasks are implemented. The `setup()` method is invoked by the JADE runtime as soon as an agent starts.

```
import jade.core.Agent;

public class BookBuyerAgent extends Agent {
  protected void setup() {
    // Printout a welcome message
    System.out.println("Hello! Buyer-agent "+getAID().getName()+" is ready.");
  }
}
```

Agent Identifier

Each agent is identified by an "agent identifier" represented as an instance of the `jade.core.AID` class. The `getAID()` method of the Agent class allows retrieving the agent identifier. An AID object includes a globally unique name plus a number of addresses. The agent name in JADE has the form `<nickname>@<platform-name>` (as explained in section 1 of this tutorial) so that an agent called `Peter` living on a platform called `P1` will have `Peter@P1` as globally unique name. The addresses included in the AID are the addresses of the platform the agent lives in. These addresses are only used when an agent needs to communicate with another agent living on a different platform.

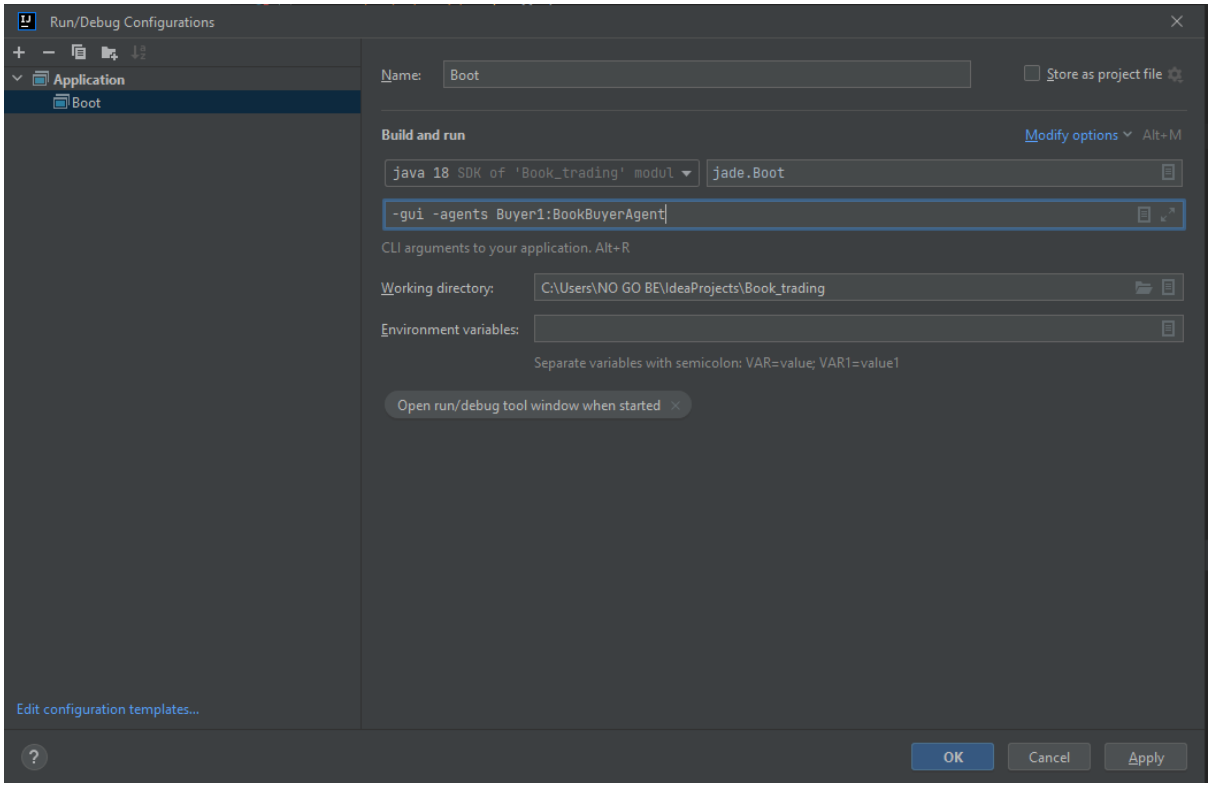
Knowing the nickname of an agent, its AID can be obtained as follows:

```
String nickname = "Peter";
AID id = new AID(nickname, AID.ISLOCALNAME);
```

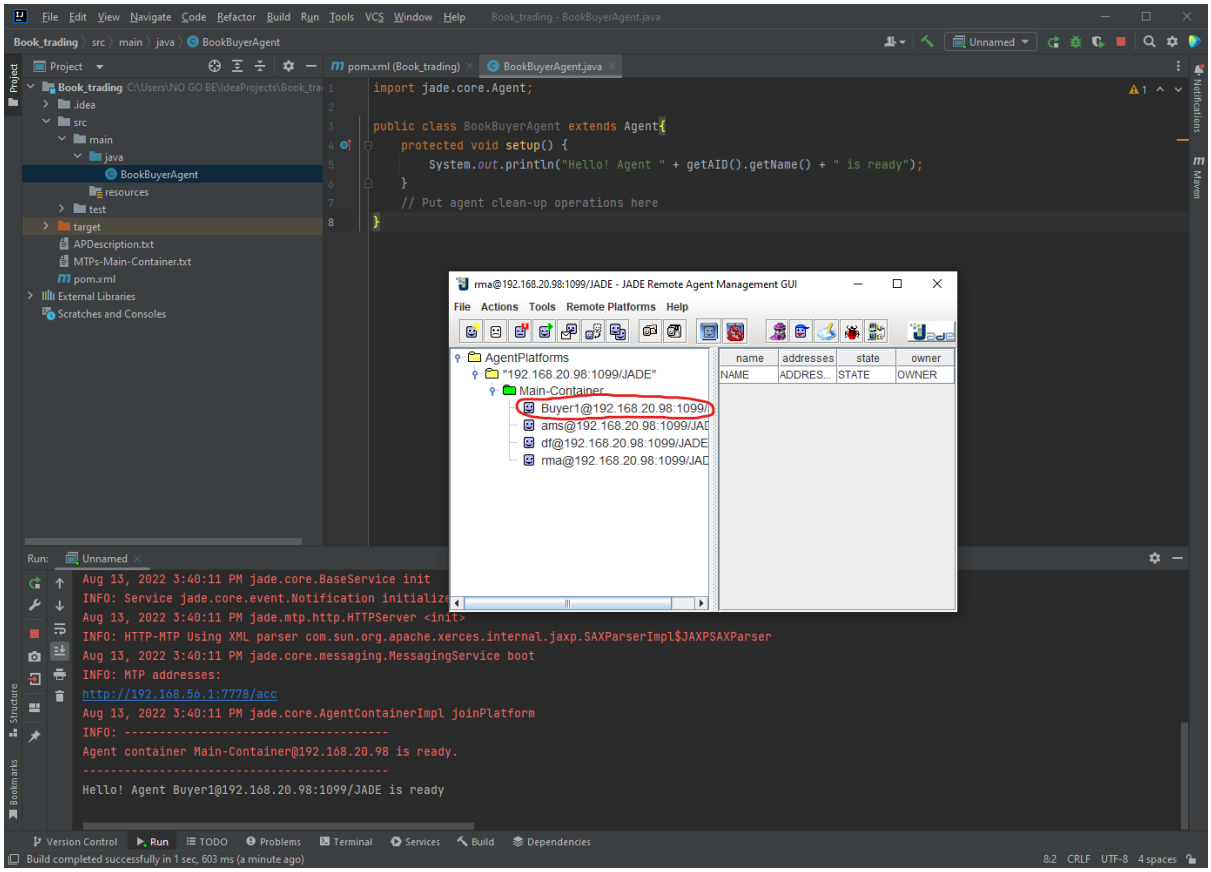
Running agents

Go to Run -> Edit Configurations and add the following to the Program Arguments:

```
-gui -agents Buyer1:BookBuyerAgent
```

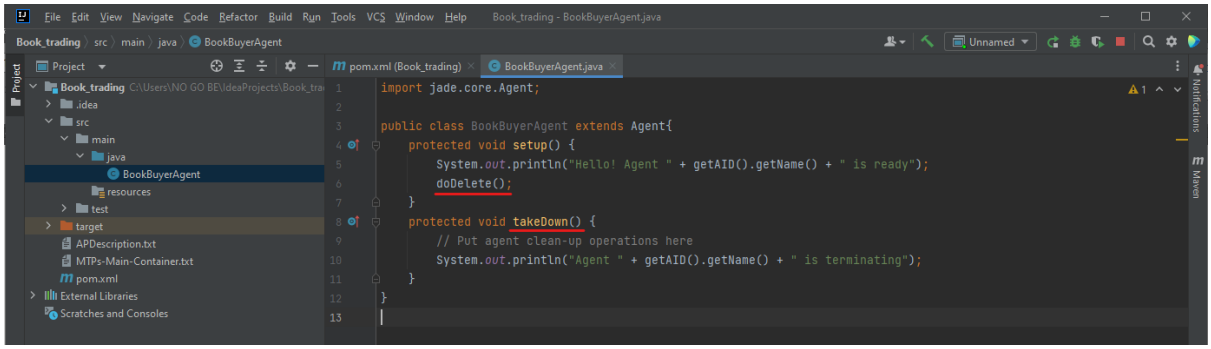


Run the program and we will have a GUI opened and there will be a “Buyer1” agent lives under the Main Container as below:



Agent termination

In order to terminate the agent, we can use the `doDelete()` method. Similar to the `setup()` method that is invoked by the JADE runtime as soon as an agent starts, the `takedown()` method is invoked just before an agent terminates and is intended to include agent clean-up operations.



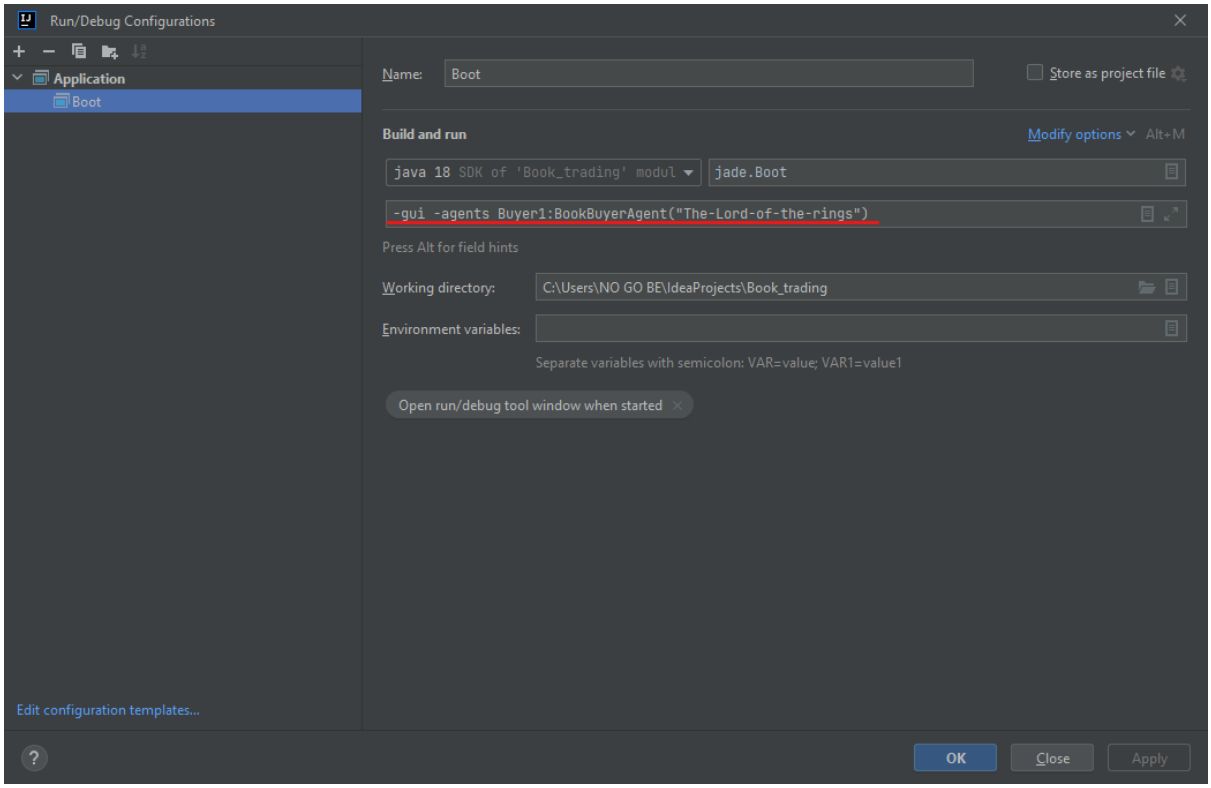
Run the program again, and you will see the "Buyer1" agent terminates when it finishes its job.

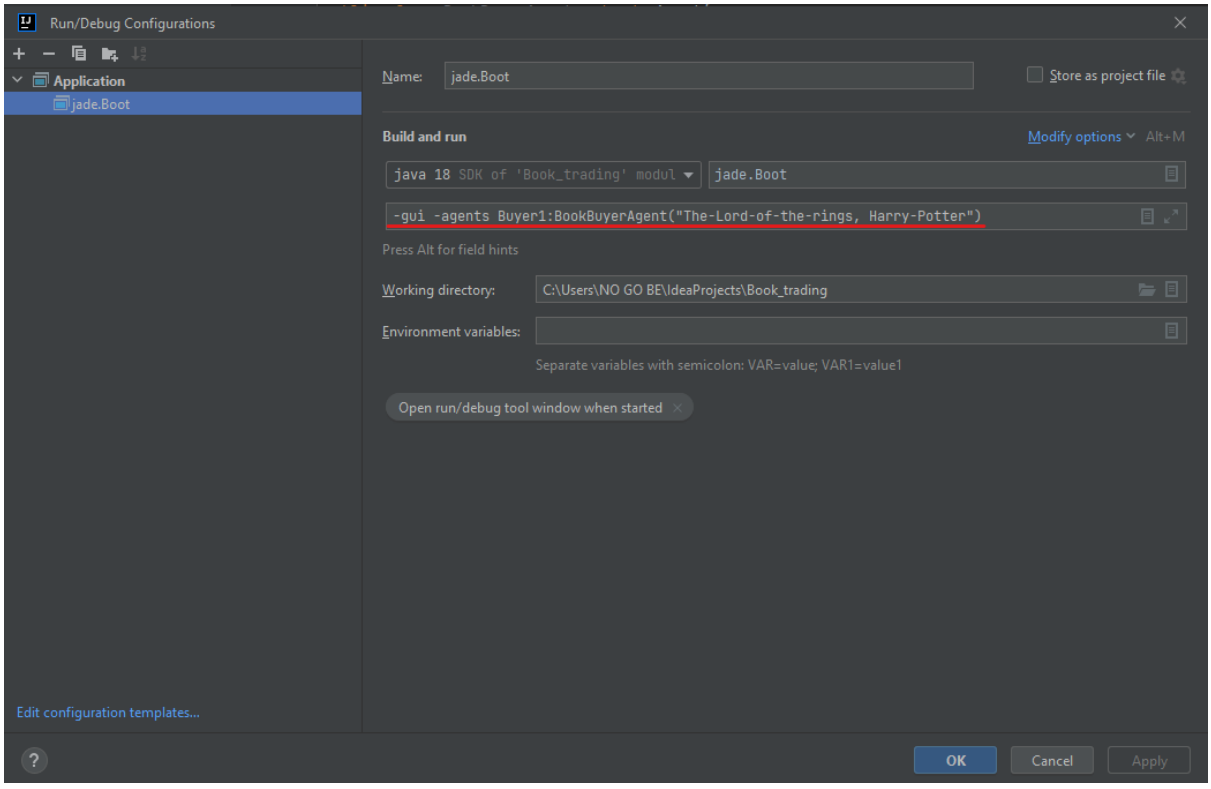
Passing arguments to agent

Agents may get start-up arguments specified on the command line. These arguments can be retrieved, as an array of Object, by using the `getArguments()` method of the Agent class. As we want our `BookBuyerAgent` to get the title of the book to buy as a command line argument. To achieve that we modify it as follows:

```
import jade.core.Agent;
import jade.core.AID;

public class BookBuyerAgent extends Agent{
    // The title of the book to buy
    private String targetBookTitle;
    // The list of known seller agents
    private AID[] sellerAgents = {new AID("seller1", AID.ISLOCALNAME),
                                  new AID("seller2", AID.ISLOCALNAME)};

    // Put agent initializations here
    protected void setup() {
        System.out.println("Hello! Agent " + getAID().getName() + " is ready");
        // Get the title of the book to buy as a start-up argument
        Object[] args = getArguments();
        if (args != null && args.length > 0) {
            targetBookTitle = (String) args[0];
            System.out.println("Trying to buy " + targetBookTitle);
        }
        else {
            // Make the agent terminate immediately
            System.out.println("No book title specified");
            doDelete();
        }
        doDelete();
    }

    protected void takeDown() {
        // Put agent clean-up operations here
        System.out.println("Agent " + getAID().getName() + " is terminating");
    }
}
```

Arguments on the command line are specified included in parenthesis and separated by commas.





b. Agent behaviours

Agent tasks – The behaviour class

The actual job an agent has to do is typically carried out within “behaviours”. A behaviour is implemented as an object of a class that extends `jade.core.behaviours.Behaviour`. In order to make an agent execute the task implemented by a behaviour object it is sufficient to add the

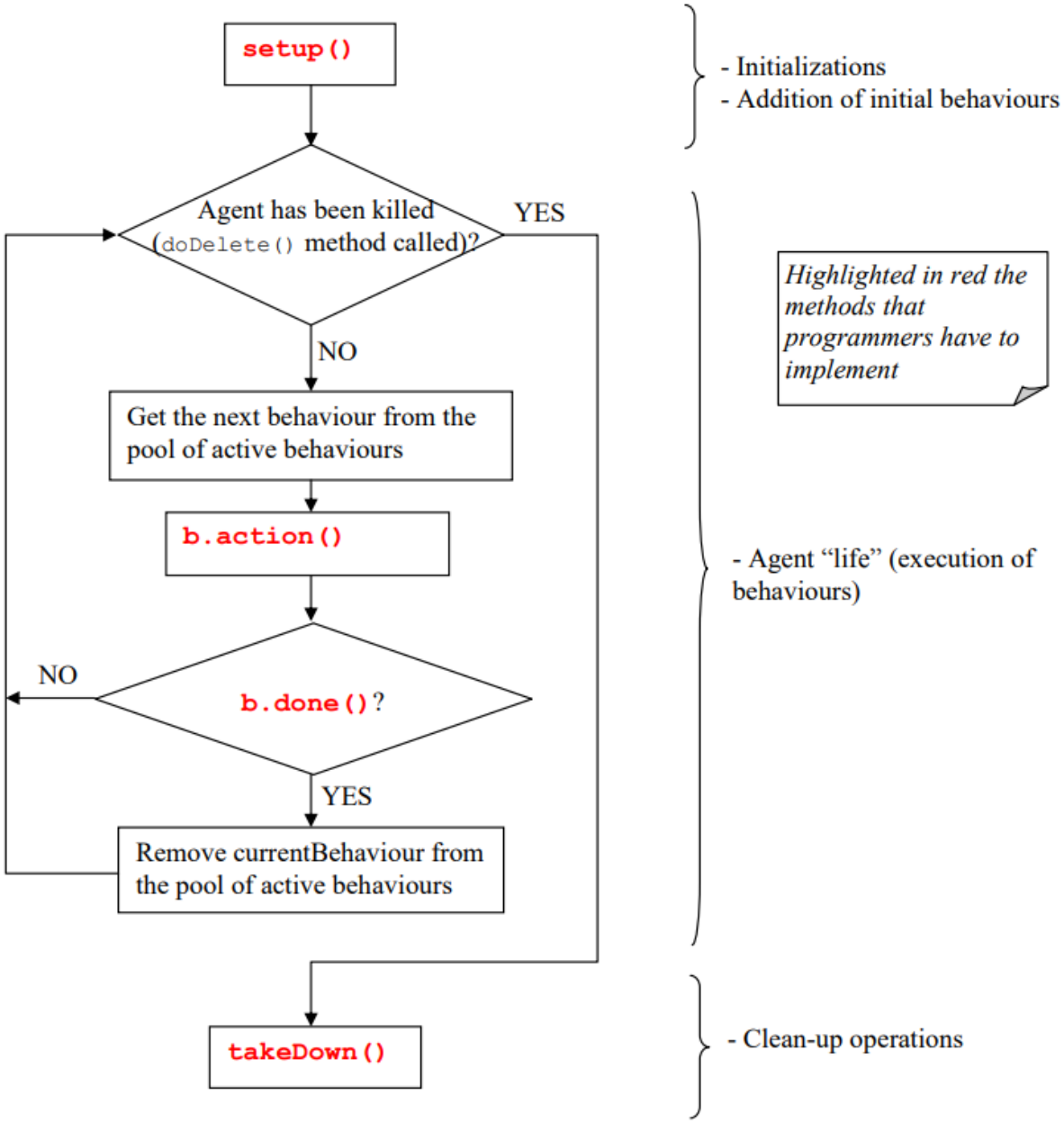
behaviour to the agent by the `addBehaviour()` method of the Agent class. Behaviours can be added at any time: when an agent starts (in the `setup()` method) or from within other behaviours.

Each class extending Behaviour must implement the `action()` method, that actually defines the operations to be performed when the behaviour is in execution and the `done()` method (returns a Boolean value), that specifies whether or not a behaviour has completed and have to be removed from the pool of behaviours an agent is carrying out.

Behaviours scheduling and execution

An agent can execute several behaviours concurrently. However, it is important to notice that scheduling of behaviours in an agent is not pre-emptive (as for Java threads) but cooperative. This means that when a behaviour is scheduled for execution its `action()` method is called and runs until it returns. Therefore, it is the programmer who defines when an agent switches from the execution of a behaviour to the execution of the next one.

The path of execution of the agent thread is depicted below:



Taking into account the described scheduling mechanism it is important to stress that a behaviour like that reported below prevents any other behaviour to be executed since its `action()` method never return. This behaviour is called **"Overbearing Behaviour"**:

```
public class OverbearingBehaviour extends Behaviour {
    public void action() {
        while (true) {
            // do something
        }
    }
}
```

One-shot behaviours, cyclic behaviours and generic behaviours

One-shot behaviours - complete immediately and whose `action()` method is executed only once. The `jade.core.behaviours.OneShotBehaviour` already implements the `done()` method by returning `true` and can be conveniently extended to implement one-shot behaviours.

```
public class MyOneShotBehaviour extends OneShotBehaviour {
  public void action() {
    // perform operation X
  }
}
```

Operation X is performed only once.

Cyclic behaviours - never complete and whose `action()` method executes the same operations each time it is called. The `jade.core.behaviours.CyclicBehaviour` already implements the `done()` method by returning `false` and can be conveniently extended to implement cyclic behaviours.

```
public class MyCyclicBehaviour extends CyclicBehaviour {
  public void action() {
    // perform operation Y
  }
}
```

Operation Y is performed repetitively forever (until the agent carrying out the above behaviour terminates).

Generic behaviours - embeds a status and execute different operations depending on that status. They complete when a given condition is met.

```
public class MyThreeStepBehaviour extends Behaviour {
  private int step = 0;
  public void action() {
    switch (step) {
    case 0:
      // perform operation X
      step++;
      break;
    case 1:
      // perform operation Y
      step++;
      break;
    case 2:
      // perform operation Z
      step++;
      break;
    }
  }

  public boolean done() {
    return step == 3;
  }
}
```

Operations X, Y and Z are performed one after the other and then the behaviour completes.

Behaviours scheduling operations

JADE provides two ready-made classes (in the `jade.core.behaviours` package), which it is possible to easily implement behaviours that execute certain operations at given points in time.

Waker behaviour - This behaviour's `action()` and `done()` method are implemented in such a way that that the behaviour completes after the execution of the `handleElapsedTimeout()` method. This method is called after a given timeout (specified in the constructor) expires.

```
public class MyAgent extends Agent {
  protected void setup() {
    System.out.println("Adding waker behaviour");
    addBehaviour(new WakerBehaviour(this, 10000) {
      protected void handleElapsedTimeout() {
        // perform operation X
      }
    } );
  }
}
```

Operation X is performed 10 seconds after the "Adding waker behaviour" printout appears.

Ticker behaviour - This behaviour's `action()` and `done()` method are implemented in such a way that that the `onTick()` method is executed repetitively after waiting for a given period (specified in the constructor) after each execution. A `TickerBehaviour` never completes.

```
public class MyAgent extends Agent {
  protected void setup() {
    addBehaviour(new TickerBehaviour(this, 10000) {
      protected void onTick() {
        // perform operation Y
      }
    } );
  }
}
```

Operation Y is performed periodically every 10 seconds.

Behaviours required for the book trading example

Book-buyer agent behaviours:

Book-buyer agent periodically requests seller agents the book it was instructed to buy. We can easily achieve that by using a `TickerBehaviour` that, on each tick, adds another behaviour that actually deals with the request to seller agents. Here is how the `setup()` method of our `BookBuyerAgent` class can be modified.

```
protected void setup() {
    // Printout a welcome message
    System.out.println("Hello! Buyer-agent "+getAID().getName()+" is ready.");

    // Get the title of the book to buy as a start-up argument
    Object[] args = getArguments();
    if (args != null && args.length > 0) {
        targetBookTitle = (String) args[0];
        System.out.println("Trying to buy " + targetBookTitle);
        // Add a TickerBehaviour that schedules a request to seller agents every 60
seconds
        addBehaviour(new TickerBehaviour(this, 60000) {
            @Override
            protected void onTick() {
                myAgent.addBehaviour(new RequestPerformer());
            }
        });
    }
    else {
        // Make the agent terminate
        System.out.println("No target book title specified");
        doDelete();
    }
}
```

Note the use of the `myAgent` protected variable: each behaviour has a pointer to the agent that is executing it.

The `RequestPerformer` behaviour actually dealing with the request to seller agents will be implemented later in this tutorial where we will discuss agent communication.

Book-seller agent behaviours:

Each Book-seller agent waits for requests from buyer agents and serves them. These requests can be requests to provide an offer for a book or purchase orders. A possible design to achieve that is to make a Book-seller agent execute two cyclic behaviours: one dedicated to serve requests for offer and the other dedicated to serve purchase orders. How actually incoming requests from buyer agents are detected and served is described later where we will discuss agent communication. Moreover, we need to make the Book-seller agents execute a one-shot behaviour updating the catalogue of books available for sale whenever the user adds a new book from the GUI. Here is how the `BookSellerAgent` class can be implemented (the `OfferRequestsServer` and `PurchaseOrdersServer` classes will be presented later).

```java
import jade.core.Agent;
import jade.core.behaviours.*;
import java.util.*;

public class BookSellerAgent extends Agent {
    // The catalogue of books for sale (maps the title of a book to its price)
    private Hashtable catalogue;
    // The GUI for the user can add books in the catalogue
    private BookSellerGui myGui;
    // Put agent initializations here
    protected void setup() {
        // Create the catalogue
        catalogue = new Hashtable();
        // Create and show the GUI
        myGui = new BookSellerGui(this);
        myGui.show();
        // Add the behaviour serving requests for offer from buyer agents
        addBehaviour(new OfferRequestsServer());
        // Add the behaviour serving purchase orders from buyer agents
        addBehaviour(new PurchaseOrdersServer());
    }
    // Put agent clean-up operations here
    protected void takeDown() {
        // Close the GUI
        myGui.dispose();
        // Printout a dismissal message
        System.out.println("Seller-agent "+getAID().getName()+" terminating.");
    }
    /**
     This is invoked by the GUI when the user adds a new book for sale
     */
    public void updateCatalogue(final String title, final int price) {
        addBehaviour(new OneShotBehaviour() {
            @Override
            public void action() {
                catalogue.put(title, new Integer(price));
            }
        });
    }
}
```

The `BookSellerGui` class is a simple Swing GUI and is not presented here since it is outside the scope of this tutorial. Its code is given in the sources packaged with this tutorial on Canvas.

c. Agent communication

One of the most important features that JADE agents provide is the ability to communicate. The communication paradigm adopted is the asynchronous message passing. Each agent has a sort of mailbox (the agent message queue) where the JADE runtime posts messages sent by other agents. Whenever a message is posted in the message queue the receiving agent is notified. If and when the agent actually picks up the message from the message queue to process it is completely up to the programmer.

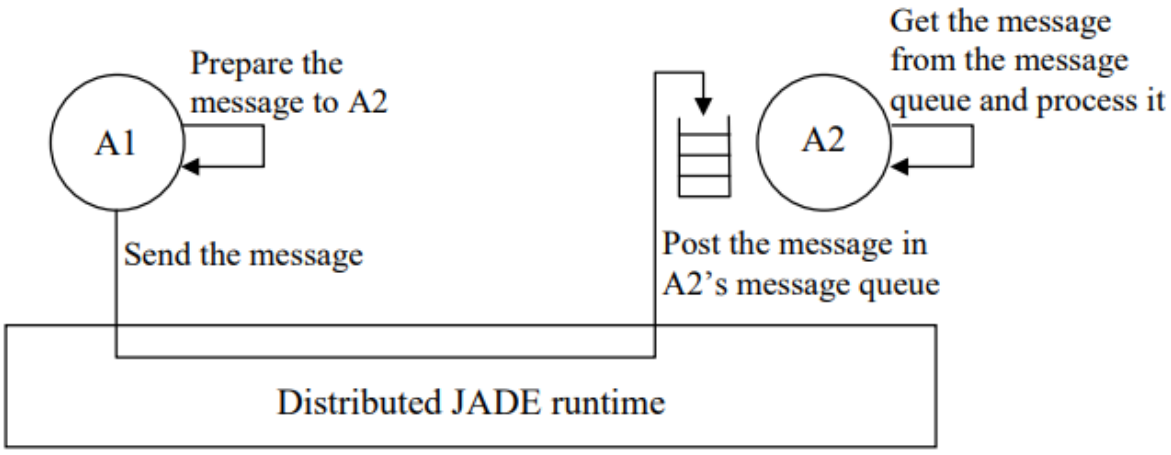


Figure 1. JADE asynchronous message passing paradigm

ACL language - Messages exchanged by JADE agents have a format specified by the ACL language defined by the FIPA (http://www.fipa.org) international standard for agent interoperability. This format comprises a number of fields and in particular.

- Sender
- Receiver
- Communicative act (performative) (e.g., REQUEST, INFORM, …)
- Content - the actual information included in the message
- Content language - the syntax used to express the content
- The ontology - the vocabulary of the symbols used in the content and their meaning
- Some fields used to control several concurrent conversations and to specify timeouts for receiving a reply such as **conversation-id**, **reply-with**, **in-reply-to**, **reply-by**.

A message in JADE is implemented as an object of the `jade.lang.acl.ACLMessage` class that provides `get` and `set` methods for handling all fields of a message.

Sending messages - Sending a message to another agent is as simple as filling the fields of an `ACLMessage` object and then call the `send()` method of the Agent class. The code below informs an agent whose nickname is Peter that today it's raining:

```
ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
msg.addReceiver(new AID("Peter", AID.ISLOCALNAME));
msg.setLanguage("English");
msg.setOntology("Weather-forecast-ontology");
msg.setContent("Today it's raining");
send(msg);
```

The book trading example messages

Considering our book trading example we can conveniently use the `CFP` (call for proposal) performative for messages that Buyer-agents send to Seller-agents to request an offer for a book. The `PROPOSE` performative can be used for messages carrying seller offers, and the `ACCEPT_PROPOSAL` performative for messages carrying offer acceptance, i.e. purchase orders. Finally the `REFUSE` performative will be used for messages sent by seller agents when the requested book is not in their catalogue. In both types of messages sent by buyer agents we assume

that the message content is the title of the book. The content of `PROPOSE` messages will be the price of the book. As an example, here is how a `CFP` message can be created and sent:

```
// Message carrying a request for offer
ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
for (int i = 0; i < sellerAgents.lenght; ++i) {
  cfp.addReceiver(sellerAgents[i]);
}
cfp.setContent(targetBookTitle);
myAgent.send(cfp);
```

Receiving messages

As mentioned above the JADE runtime automatically posts messages in the receiver's private message queue as soon as they arrive. An agent can pick up messages from its message queue by the `receive()` method. This method returns the first message in the message queue (removing it) or null if the message queue is empty and immediately returns.

```
ACLMessage msg = receive();
if (msg != null) {
  // Process the message
}
```

Blocking a behaviour waiting for a message

Very often programmers need to implement behaviours that process messages received by other agents. This is the case for the `OfferRequestsServer` and `PurchaseOrdersServer` behaviours in the book-seller agent implementation above, where we need to serve messages from buyer agents carrying requests for offer and purchase orders. Such behaviour must be continuously running (cyclic behaviours) and, at each execution of their `action()` method, must check if a message has been received and process it. The two behaviours are very similar. Here we present the `OfferRequestsServer` behaviour. Look at the sources given on Canvas for the code of the `PurchaseOrdersServer`.

```
/**
   Inner class OfferRequestsServer.
   This is the behaviour used by Book-seller agents to serve incoming requests
   for offer from buyer agents.
   If the requested book is in the local catalogue the seller agent replies
   with a PROPOSE message specifying the price. Otherwise a REFUSE message is
   sent back.
 */
private class OfferRequestsServer extends CyclicBehaviour {
  public void action() {
    ACLMessage msg = myAgent.receive();
    if (msg != null) {
      // Message received. Process it
      String title = msg.getContent();
      ACLMessage reply = msg.createReply();

      Integer price = (Integer) catalogue.get(title);
      if (price != null) {
        // The requested book is available for sale. Reply with the price
        reply.setPerformative(ACLMessage.PROPOSE);
        reply.setContent(String.valueOf(price.intValue()));
      }
      else {
        // The requested book is NOT available for sale.
        reply.setPerformative(ACLMessage.REFUSE);
        reply.setContent("not-available");
      }
      myAgent.send(reply);
    }
  }
}  // End of inner class OfferRequestsServer
```

We decided to implement the `OfferRequestsServer` behaviour as an inner class of the `BookSellerAgent` class. This simplifies things as we can directly access the catalogue of books for sale; it is not mandatory however.

The `createReply()` method of the `ACLMessage` class automatically creates a new `ACLMessage` properly setting the receivers and all the fields used to control the conversation (conversation-id, reply-with, in-reply-to) if any.

When we add the above behaviour, the agent's thread starts a continuous loop that is extremely CPU consuming. To avoid that we would like to execute the `action()` method only when a new message is received. In order to do that we can use the `block()` method of the `Behaviour` class. This method marks the behaviour as "blocked" so that the agent does not schedule it for execution anymore. When a new message is inserted in the agent's message queue, all blocked behaviours becomes available for execution again.

block

```
public void block(long millis)
```

Blocks this behaviour for a specified amount of time. The behaviour will be restarted when among the three following events happens.

- *A time of `millis` milliseconds has passed since the call to `block()`.*
- *An ACL message is received by the agent this behaviour belongs to.*
- *Method `restart()` is called explicitly on this behaviour object.*

Parameters:

`millis` - The amount of time to block, in milliseconds. ***Notice:** a value of 0 for `millis` is equivalent to a call to `block()` without arguments.*

The `action()` method must therefore be modified as follows:

```
public void action() {
  ACLMessage msg = myAgent.receive();
  if (msg != null) {
    // Message received. Process it
    ...
  }
  else {
    block();
  }
}
```

The above code is the typical (and strongly suggested) pattern for receiving messages inside a behaviour.

Selecting messages with given characteristics from the message queue

Considering that both the `OfferRequestsServer` and `PurchaseOrdersServer` behaviours are cyclic behaviour whose `action()` method starts with a call to `myAgent.receive()`, you may have noticed a problem: how can we be sure that the `OfferRequestsServer` behaviour picks up from the agent's message queue only messages carrying requests for offer and the `PurchaseOrdersServer` behaviour only messages carrying purchase orders?

We solve the problem by specifying proper "templates" when we call the `receive()` method. Such templates are implemented as instances of the `jade.lang.acl.MessageTemplate` class that provides a number of factory methods to create templates in a very simple and flexible way.

As mentioned earlier, we use the `CFP` performative for messages carrying requests for offer and the `ACCEPT_PROPOSAL` performative for messages carrying proposal acceptances, i.e. purchase orders. Therefore we modify the `action()` method of the `OfferRequestsServer` so that the call to `myAgent.receive()` ignores all messages except those whose performative is `CFP`.

```
public void action() {
  MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.CFP);
  ACLMessage msg = myAgent.receive(mt);
  if (msg != null) {
    // CFP Message received. Process it
    ...
  }
  else {
    block();
  }
}
```

Complex conversations

The `RequestPerformer` behaviour mentioned earlier in the book-buyer agent implementation represents an example of a behaviour carrying out a "complex" conversation. A conversation is a sequence of messages exchanged by two or more agents with well-defined causal and temporal relations. The `RequestPerformer` behaviour has to send a `CFP` message to several seller agents, get back all the replies and, in case at least a `PROPOSE` reply is received, send a further `ACCEPT_PROPOSAL` message (to the seller agent that made the proposal) and get back the response. Whenever a conversation has to be carried out it is a good practice to specify the

conversation control fields in the messages exchanged within the conversation. This allows to easily create templates matching the possible replies.

```
private class RequestPerformer extends Behaviour {
    private AID bestSeller; // The agent who provides the best offer
    private int bestPrice;  // The best offered price
    private int repliesCnt = 0; // The counter of replies from seller
agents
    private MessageTemplate mt; // The template to receive replies
    private int step = 0;

    public void action() {
        switch (step) {
            case 0:
                // Send the cfp to all sellers
                ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
                for (int i = 0; i < sellerAgents.length; ++i) {
                    cfp.addReceiver(sellerAgents[i]);
                }
                cfp.setContent(targetBookTitle);
                cfp.setConversationId("book-trade");
                cfp.setReplyWith("cfp"+System.currentTimeMillis()); //
Unique value
                myAgent.send(cfp);
                // Prepare the template to get proposals
                mt =
MessageTemplate.and(MessageTemplate.MatchConversationId("book-trade"),

MessageTemplate.MatchInReplyTo(cfp.getReplyWith()));
                step = 1;
                break;
            case 1:
                // Receive all proposals/refusals from seller agents
                ACLMessage reply = myAgent.receive(mt);
                if (reply != null) {
                    // Reply received
                    if (reply.getPerformative() == ACLMessage.PROPOSE) {
                        // This is an offer
                        int price = Integer.parseInt(reply.getContent());
                        if (bestSeller == null || price < bestPrice) {
                            // This is the best offer at present
                            bestPrice = price;
                            bestSeller = reply.getSender();
                        }
                    }
                    repliesCnt++;
                    if (repliesCnt >= sellerAgents.length) {
                        // We received all replies
                        step = 2;
                    }
                } else {
                    block();
                }
                break;
            case 2:
                // Send the purchase order to the seller that provided the
best offer
                ACLMessage order = new
ACLMessage(ACLMessage.ACCEPT_PROPOSAL);
                order.addReceiver(bestSeller);
                order.setContent(targetBookTitle);
```

```
                order.setConversationId("book-trade");
                order.setReplyWith("order"+System.currentTimeMillis());
                myAgent.send(order);
                // Prepare the template to get the purchase order reply
                mt =
MessageTemplate.and(MessageTemplate.MatchConversationId("book-trade"),

MessageTemplate.MatchInReplyTo(order.getReplyWith()));
                step = 3;
                break;
            case 3:
                // Receive the purchase order reply
                reply = myAgent.receive(mt);
                if (reply != null) {
                    // Purchase order reply received
                    if (reply.getPerformative() == ACLMessage.INFORM) {
                        // Purchase successful. We can terminate
                        System.out.println(targetBookTitle+" successfully
purchased from agent "+reply.getSender().getName());
                        System.out.println("Price = "+bestPrice);
                        myAgent.doDelete();
                    } else {
                        System.out.println("Attempt failed: requested book
already sold.");
                    }

                    step = 4;
                } else {
                    block();
                }
                break;
        }
    }

    public boolean done() {
        if (step == 2 && bestSeller == null) {
            System.out.println("Attempt failed: "+targetBookTitle+" not
available for sale");
        }
        return ((step == 2 && bestSeller == null) || step == 4);
    }
}  // End of inner class RequestPerformer
```

3. Other JADE practices

So far we have assumed that there is a fixed set of seller agents (seller1 and seller2) and that each buyer agent already knows them (see the `AID[] sellerAgents` member variable of the `BookBuyerAgent` class). There is a way to get rid of this assumption and exploit the **yellow pages** service provided by the JADE platform to make buyer agents dynamically discover seller agents available at a given point in time.

For the above improvement and other advanced practices such as:

- Implementing sequential/parallel tasks
- Implementing tasks with finite state machine based scheduling
- How to use `AMService` to get details about other agents in the JADE platform
- Simple agent communication using messages
- Interacting with the JADE platform from a non-JADE program

- Using simple message templates
- Using FIPA Request Interaction Protocol
- How agents can register services with the JADE Directory Facilitator (DF) service
- How agents can dynamically lookup for agents offering “specific” services
- How agents can subscribe for specific types of services so that they get notified when an agent registers the service with the DF Agent
- How the Sniffer agent can be used to monitor/visualize message exchanges between agents
- How agents move between containers

Please look at the detailed instructions given in the “Other JADE practices (1)”, “Other JADE practices (2)”, “Other JADE practices (3)” on Canvas. Depending on your chosen project, only a few of those practices are useful to you.