JADE

Java **A**gent **D**evelopment **E**nvironment

Ömer Faruk ALACA

Introduction to Jade

- JADE (Java Agent DEvelopment Framework) is a software framework **fully implemented in Java language**.

- Easy implementation of multi-agent systems using **a middle-ware**
  - Compatible with **FIPA specifications***
  - a set of tools that supports the **debugging and deployment phase**.

*For more information : https://jade.tilab.com/papers/JADETutorialIEEE/JADETutorial_FIPA.pdf

Introduction to Jade

- The agent platform can be **distributed across machines**
  - not even need to share the same OS
- The configuration can be controlled via a **remote GUI.**
- The configuration can be even changed at run-time by **creating new agents** and **moving agents from one machine to another one**, as and when required.

Introduction to Jade

- The only system requirement is the Java Run Time version 5 or later.
- JADE is distributed in **Open Source** under the **LGPL License**.
  - Further details and documentation can be found at http://jade.tilab.com/
- BT, Telefonica, CNET, NHK, Imperial College, IRST, KPN, University of Helsinky, INRIA, ATOS, and many others.

Jade Overview

- JADE is a middleware that facilitates the development of multi-agent systems. It includes
  - A **runtime environment**
    - where JADE agents can “live” and that must be active on a given host before one or more agents can be executed on that host.
  - A **library** of classes
    - that programmers have to/can use (directly or by specializing them) to develop their agents.
  - A suite of **graphical tools**
    - that allows administrating and monitoring the activity of running agents.

Download JADE

https://jade.tilab.com/dl.php?file=JADE-all-4.5.0.zip

Containers and Platforms

- Each running instance of the JADE runtime environment is called a **Container**
  - it can contain several agents.
- The set of active containers is called a **Platform**.
- A single special **Main container** must always be active in a platform
  - all other containers register with it as soon as they start.

Containers and Platforms

- The first container to start in a platform must be a **main container** while all other containers must be **“normal”** (i.e. non-main) containers
- Normal containers must **“be told”** where to find (host and port) their main container (i.e. the main container to register with).
- If another main container is started somewhere in the network it constitutes a different platform to which new normal containers can possibly register.

Containers and Platforms

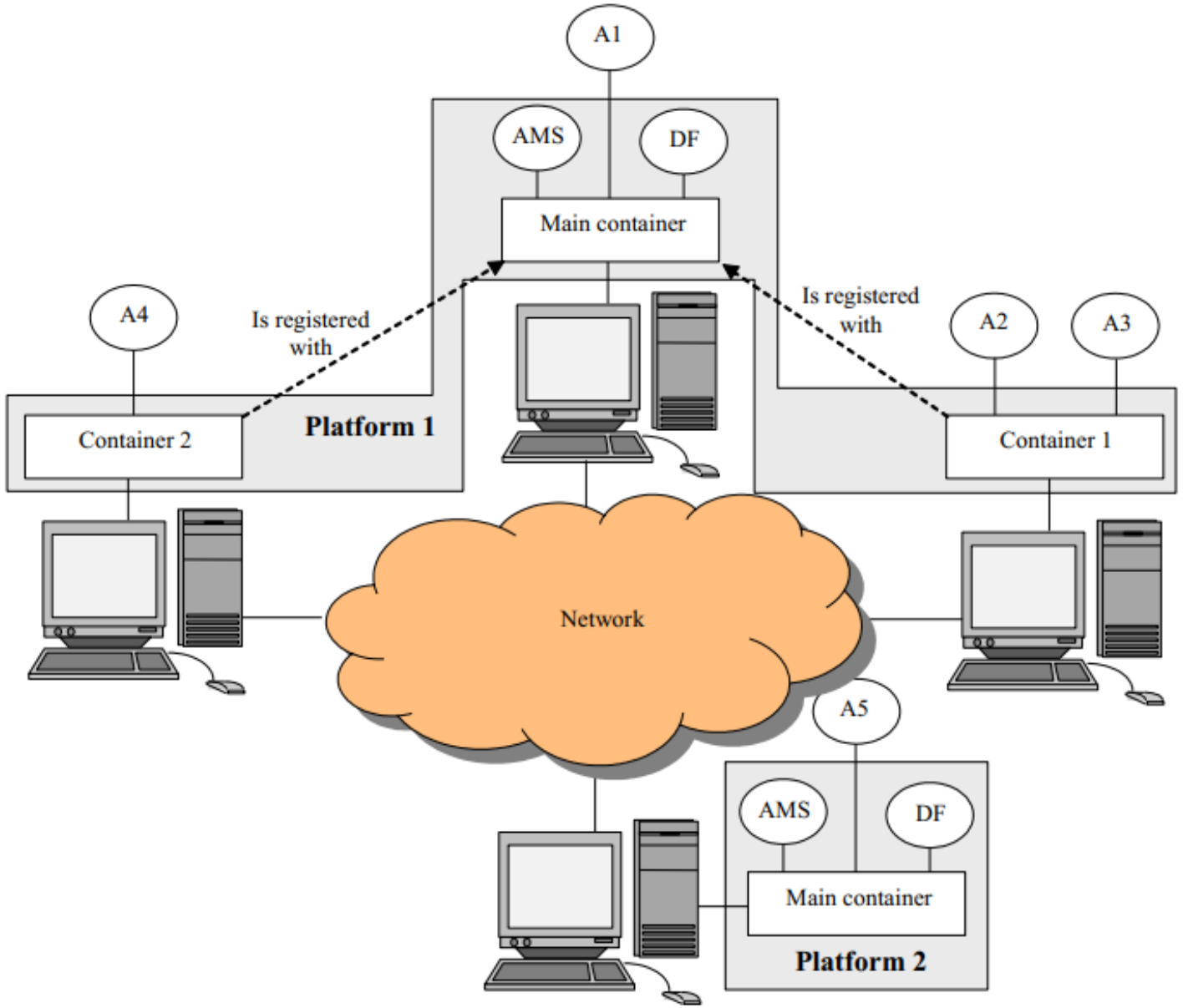


Figure 1Containers and Platforms

Containers and Platforms

- Main Container activating the JADE management GUI (-gui) option.
  - <classpath> must include all jade classes plus all required application-specific classes.

```
java -cp <classpath> jade.Boot -gui
```

- peripheral container (***-container*** option) that registers to a main container running on host **avalon.tilab.com** (**-host** option) and activates an agent called john of class **myPackage.MyClass** (**-agents**) option

```
java -cp <classpath> jade.Boot -container -host avalon.tilab.com -agents john:myPackage.myClass
```

Main Container

- A main container differs from normal containers as it holds two special agents
  - Agent Management System (AMS)
  - Directory Facilitator (DF)

AMS (Agent Management System)

- provides the naming service
  - i.e. ensures that each agent in the platform has a unique name
- represents the authority in the platform
  - for instance it is possible to create/kill agents on remote containers by requesting that to the AMS

- This tutorial does not illustrate how to interact with the AMS as this is part of the **advanced JADE programming**

DF (Directory Facilitator)

- provides a Yellow Pages service
  - means of which an agent can find other agents providing the services he requires in order to achieve his goals.

The “Book Trading” Example

1. The scenario considered in this example includes **some agents selling books** and **other agents buying books** on behalf of their users.
2. 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.
3. 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).**
4. Having bought the target book the buyer agent terminates.

The “Book Trading” Example

5. Each seller agent has **a minimal GUI** by means of which the user can insert new titles (and the associated price) in the local catalogue of books for sale.
6. Seller agents continuously wait for requests from buyer agents.
7. 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.
8. When they receive a purchase order they serve it and remove the requested book from their catalogue.

Creating Jade Agents – The Agent Class

```
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.");
  }
}
```

Creating a JADE agent is as simple as defining a class extending the **jade.core.Agent** class

The **setup()** method is intended to include agent initializations.

The Agent Class - Agent Identifiers

- 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.

The Agent Class - Agent Identifiers

- An `AID` object includes a ***globally unique name*** plus ***a number of addresses***.
- `<nickname>@<platform-name>` ? Globally Unique Name
  - An agent called *Peter* living on a platform called *P1* = ***Peter@P1***
- 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.

The Agent Class - Agent Identifiers

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

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

- The `ISLOCALNAME` constant indicates that the first parameter represents the nickname (local to the platform) and **not the globally unique name** of the agent.

The Agent Class - Running agents

- The created agent can be compiled as follows:

```
javac -classpath <JADE-classes> BookBuyerAgent.java
```

- In order to execute the compiled agent the JADE runtime must be started and a nickname for the agent to run must be chosen:

```
java -classpath <JADE-classes>;. jade.Boot buyer:BookBuyerAgent
```

The Agent Class - Running

```
C:\jade>java -classpath <JADE-classes>;. jade.Boot buyer:BookBuyerAgent
5-mag-2008 11.06.45 jade.core.Runtime beginContainer
INFO: ----------------------------------
    This is JADE snapshot - revision 5995 of 2007/09/03 09:45:22
    downloaded in Open Source, under LGPL restrictions,
    at http://jade.tilab.com/
----------------------------------------
5-mag-2008 11.06.51 jade.core.BaseService init
INFO: Service jade.core.management.AgentManagement initialized
5-mag-2008 11.06.51 jade.core.BaseService init
INFO: Service jade.core.messaging.Messaging initialized
5-mag-2008 11.06.52 jade.core.BaseService init
INFO: Service jade.core.mobility.AgentMobility initialized
5-mag-2008 11.06.52 jade.core.BaseService init
INFO: Service jade.core.event.Notification initialized
5-mag-2008 11.06.52 jade.core.messaging.MessagingService clearCachedSlice
INFO: Clearing cache
5-mag-2008 11.06.53 jade.mtp.http.HTTPServer <init>
INFO: HTTP-MTP Using XML parser com.sun.org.apache.xerces.internal.parsers.SAXParser
5-mag-2008 11.06.54 jade.core.messaging.MessagingService boot
INFO: MTP addresses:
http://NBNT2004130496.telecomitalia.local:7778/acc
5-mag-2008 11.06.54 jade.core.AgentContainerImpl joinPlatform
INFO: ----------------------------------------
Agent container Main-Container@NBNT2004130496 is ready.
--------------------------------------------
Hello! Buyer-agent buyer@NBNT2004130496:1099/JADE is ready.
```

automatically assigned on the basis of the host and port you are running JADE on

The Agent Class - Agent termination

- Even if it does not have anything else to do after printing the welcome message, our agent is still running.
- In order to make it terminate its `doDelete()` method must be called.
- Similarly to the `setup()` method, the `takeDown()` method is invoked just before an agent terminates and is intended to include agent clean-up operations.

The Agent Class - Passing arguments to an agent

- Agents may get start-up arguments specified on the command line.
- These arguments can be retrieved, as an array of `Object`, by means of the `getArguments()` method of the `Agent` class.

```
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)};
```

we will describe how to dynamically discover seller agents, next slides!

The Agent Class - Passing arguments to an agent

```
  // Put agent initializations here
  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);
    }
    else {
      // Make the agent terminate immediately
      System.out.println("No book title specified");
      doDelete();
    }
  }

  // Put agent clean-up operations here
  protected void takeDown() {
    // Printout a dismissal message
    System.out.println("Buyer-agent "+getAID().getName()+" terminating.");
  }
}
```

Get command line arguments here!

Agent terminate
and
Clean-up Operations

The Agent Class - Passing arguments to an agent

```
C:\jade>java jade.Boot buyer:BookBuyerAgent(The-Lord-of-the-rings)
...
...
5-mag-2008 11.11.00 jade.core.AgentContainerImpl joinPlatform
INFO: ----------------------------------
Agent container Main-Container@NBNT2004130496 is ready.
----------------------------------
Hello! Buyer-agent buyer@NBNT2004130496:1099/JADE is ready.
Trying to buy The-Lord-of-the-Rings
```

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

Agent Tasks – The Behaviour Class

- The actual job an agent has to do is typically carried out within **“behaviours”.**
- A behaviour
  - represents a ***task*** that an agent can carry out.
  - is implemented as an object of a class that extends `jade.core.behaviours.Behaviour`.

Agent Tasks – The Behaviour Class

- 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 means of the `addBehaviour()` method of the `Agent` class.

- Behaviours can be added at any time:
  - when an agent starts (in the `setup()` method)
  - from within other behaviours.

Agent Tasks -The Behaviour Class

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

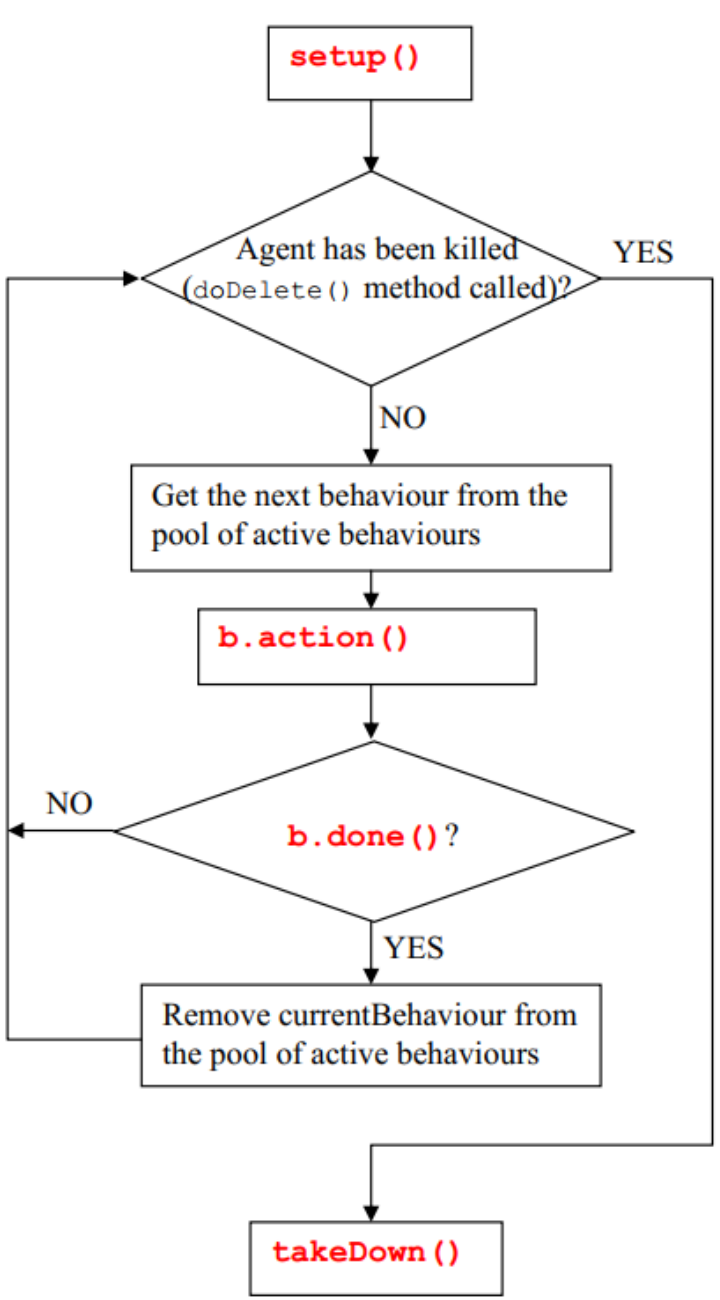
Behaviours scheduling and execution

- 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.

Behaviours scheduling and execution

- Though requiring a small additional effort to programmers, this approach has several advantages.
  - Allows having a single Java thread per agent
  - Provides better performances since behaviour switch is extremely faster than Java thread switch.
  - Eliminates all synchronization issues between concurrent behaviours accessing the same resources (this speed-up performances too) since all behaviours are executed by the same Java thread.

Agent Thread path of execution



- Initializations
- Addition of initial behaviours

Highlighted in red the methods that programmers have to implement

- Agent "life" (execution of behaviours)

- Clean-up operations

Figure 2. Agent Thread path of execution

Behaviours scheduling and execution

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

  public boolean done() {
    return true;
  }
}
```

prevents any other behaviour to be executed since its **action()** method never returns.

Behaviours scheduling and execution

- When there are **no behaviours available for execution** the agent's thread goes to sleep in order not to consume CPU time.
- It is waken up as soon as there is again a behaviour available for execution.

Types of Behaviours

- We can distinguish among three types of behavior

1. One-shot behaviours
2. Cyclic behaviours
3. Generic behaviours

One-shot Behaviours

- "One-shot" behaviours that 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

- "Cyclic" behaviours that 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

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

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

```
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;
  }
}
```

Complex Behaviours

- JADE provides the possibility of combining simple behaviours together to create complex behaviours.
  - `SequentialBehaviour`
  - `ParallelBehaviour`
  - `FSMBehaviour`

- Refer to the **Javadoc** of the `SequentialBehaviour`, `ParallelBehaviour` and `FSMBehaviour` for the details.

Scheduling operations at given points in time

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

1. WakerBehaviour
2. TickerBehaviour

WakerBehaviour

- The `WakerBehaviour` whose `action()` and `done()` methods are already implemented in such a way to execute the `handleElapsedTimeout()` abstract method 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.

TickerBehaviour

- The `TickerBehaviour` whose `action()` and `done()` methods are already implemented in such a way to execute the `onTick()` abstract method repetitively waiting a given period (specified in the constructor) after each execution.

```
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 in the Book Trading example

AGENT COMMUNICATION – The ACLMessage Class

- 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.

AGENT COMMUNICATION – The ACLMessage Class

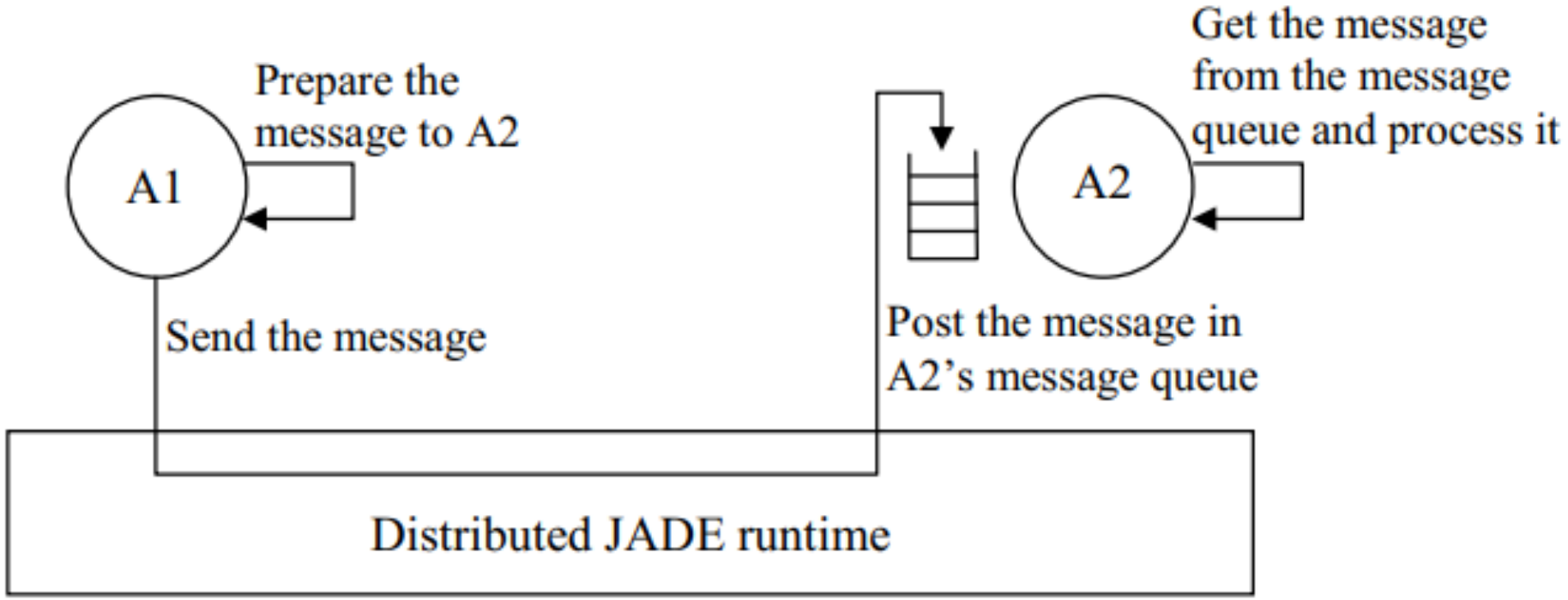


Figure 3. The JADE asynchronous message passing paradigm

THE ACLMESSAGE CLASS

–The ACL language

- Messages exchanged by JADE agents have a format specified by the ACL language defined by the FIPA* international standard for agent interoperability.
- `jade.lang.acl.ACLMessage` class that provides `get` and `set` methods for handling all fields of a message.

Parameter	Category of Parameters
performative	Type of communicative acts
sender	Participant in communication
receiver	Participant in communication
reply-to	Participant in communication
content	Content of message
language	Description of Content
encoding	Description of Content
ontology	Description of Content
protocol	Control of conversation
conversation-id	Control of conversation
reply-with	Control of conversation
in-reply-to	Control of conversation
reply-by	Control of conversation

Table 1: FIPA ACL Message Parameters

* http://www.fipa.org

THE ACLMESSAGE CLASS

–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.

```
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);
```

informs an agent whose nickname is *Peter* that *today it's raining*.

The Book Trading example messages

THE ACLMESSAGE CLASS

–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 means of 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
}
```

THE ACLMESSAGE CLASS

–Blocking a behaviour waiting for a message

- the agent's thread starts a continuous loop that is extremely CPU consuming.
- In order to avoid that we would like to execute the `action()` method the cyclic behaviour only when a new message is received.
- In order to do that we can use the `block()` method of the Behaviour class.

```
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**.

THE ACLMESSAGE CLASS

–Selecting messages with given characteristics from the message queue

- When a template is specified the `receive()` method returns the first message (if any) matching it, while ignores all non-matching messages.
- 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.

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

THE ACLMESSAGE CLASS

–Complex conversations

- A conversation is a sequence of messages exchanged by two or more agents with well defined causal and temporal relations.
- The `RequestPerformer` behaviour mentioned in Book Trading represents an example of a behaviour carrying out a “complex” conversation.
  - ✔send a CFP message to several agents (the known seller agents),
  - ✔get back all the replies
  - ✔in case at least a `PROPOSE` reply is received,
  - ✔a further `ACCEPT_PROPOSAL` message (to the seller agent that made the proposal) and get back the response.

THE ACLMESSAGE CLASS

–Receiving messages in blocking mode

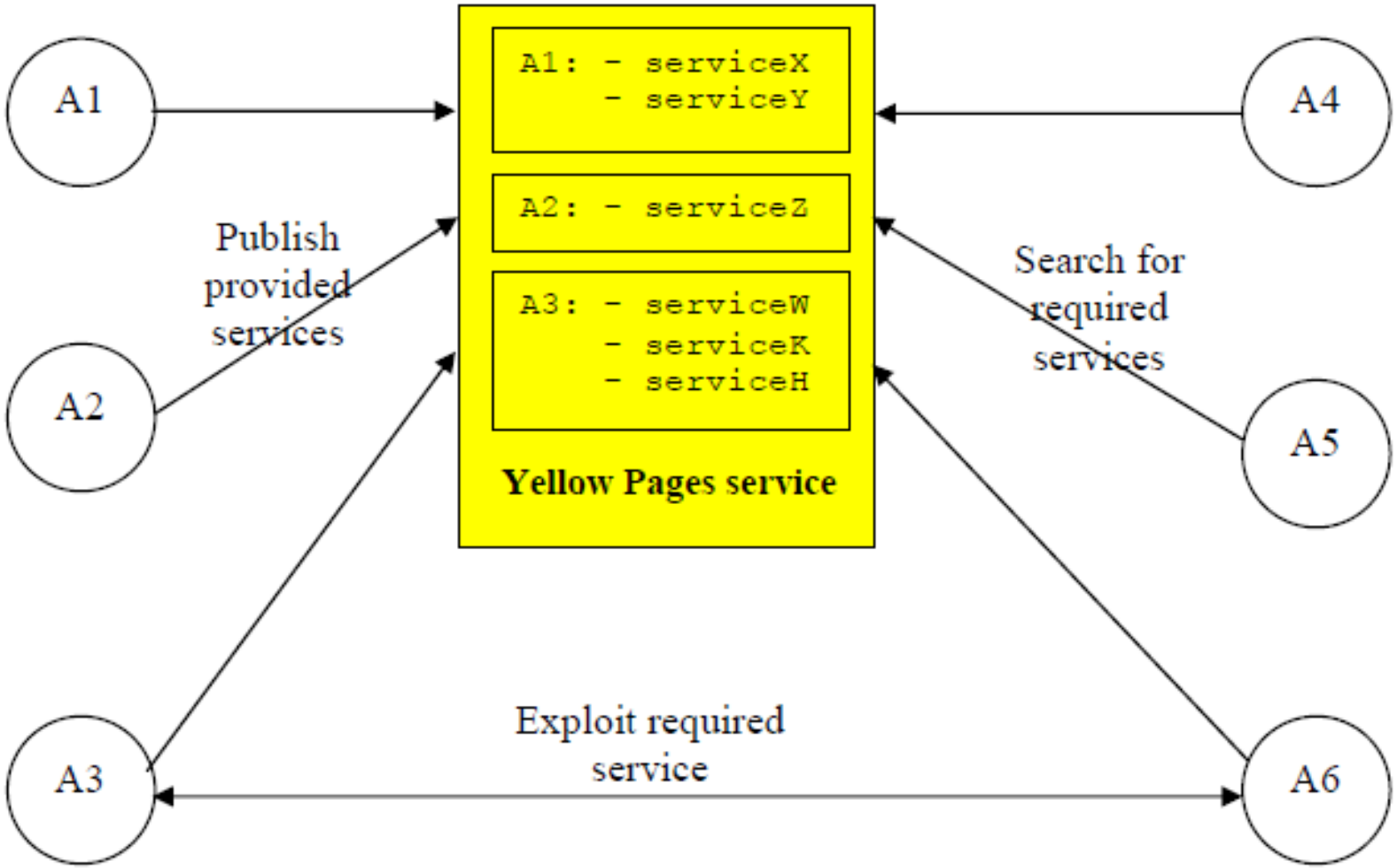
- if you call `blockingReceive()` from within a behaviour, this prevents all other behaviours to run until the call to `blockingReceive()` returns.
- a good programming practice
  - ✔to receive messages is use `blockingReceive()` in the `setup()` and `takeDown()` methods; use `receive()` in combination with `Behaviour.block()` within behaviours.

The ACLMessage Class
Book Trading example

THE YELLOW PAGES SERVICE –The DFSERVICE Class

- A “yellow pages” service allows agents to publish one or more services they provide so that other agents can find and successively exploit them.
- The yellow pages service in JADE (according to the FIPA specification) is provided by an agent called DF (Directory Facilitator).
- Each FIPA compliant platform hosts a default DF agent (whose local name is “df”)

THE YELLOW PAGES SERVICE –The DFSERVICE Class



THE YELLOW PAGES SERVICE –Interacting with the DF

- To interact with DF by exchanging ACL messages using a
  - ✔proper content language (the SL0 language)
  - ✔a proper ontology (the FIPA-agent-management ontology) according to the FIPA specification.
- In order to simplify these interactions,
  - ✔JADE provides the `jade.domain.DFService` class by means of which it is possible to publish and search for services through method calls.

THE YELLOW PAGES SERVICE –Publishing services

- In order to publish a service an agent must create a proper description and call the `register()` static method of the `DFService` class.

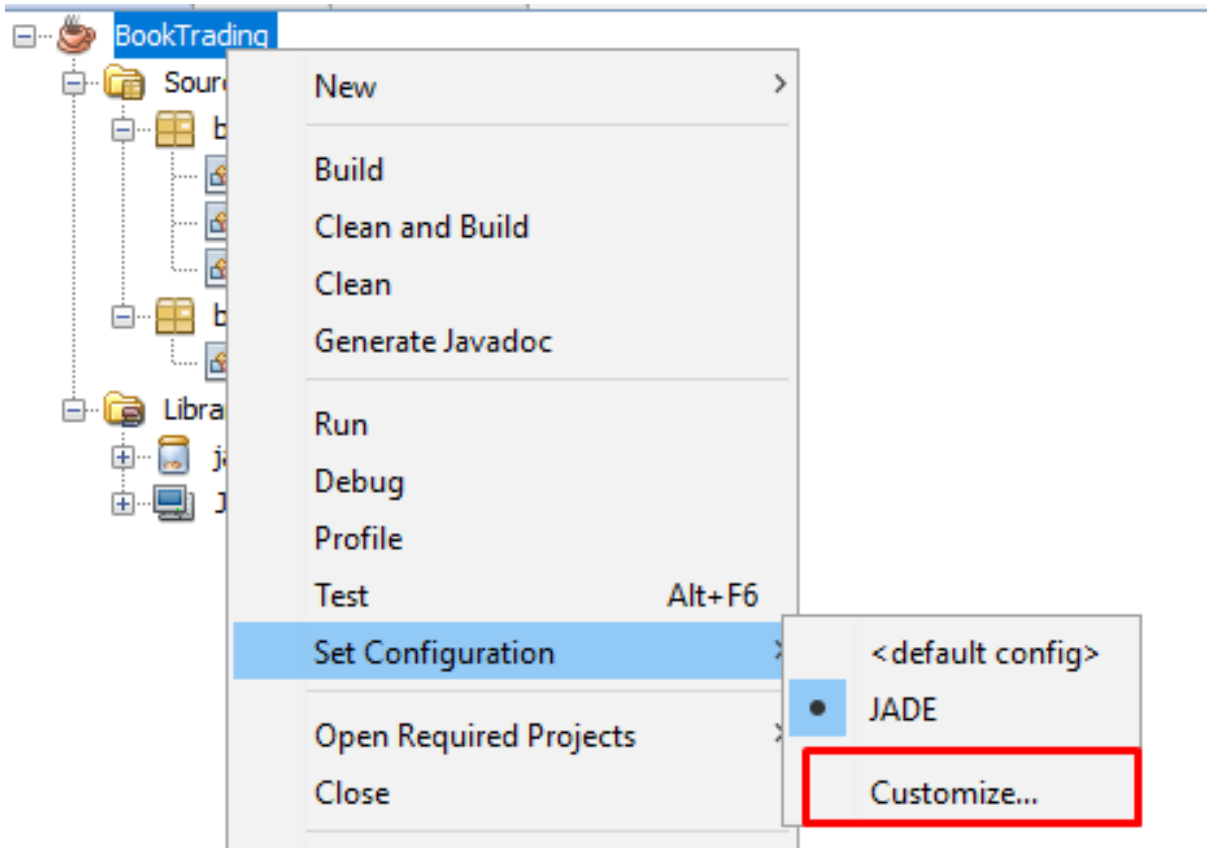
```
protected void setup() {
  ...
  // Register the book-selling service in the yellow pages
  DFAgentDescription dfd = new DFAgentDescription();
  dfd.setName(getAID());
  ServiceDescription sd = new ServiceDescription();
  sd.setType("book-selling");
  sd.setName("JADE-book-trading");
  dfd.addServices(sd);
  try {
    DFService.register(this, dfd);
  }
  catch (FIPAException fe) {
    fe.printStackTrace();
  }
  ...
}
```

THE YELLOW PAGES SERVICE –Searching for services

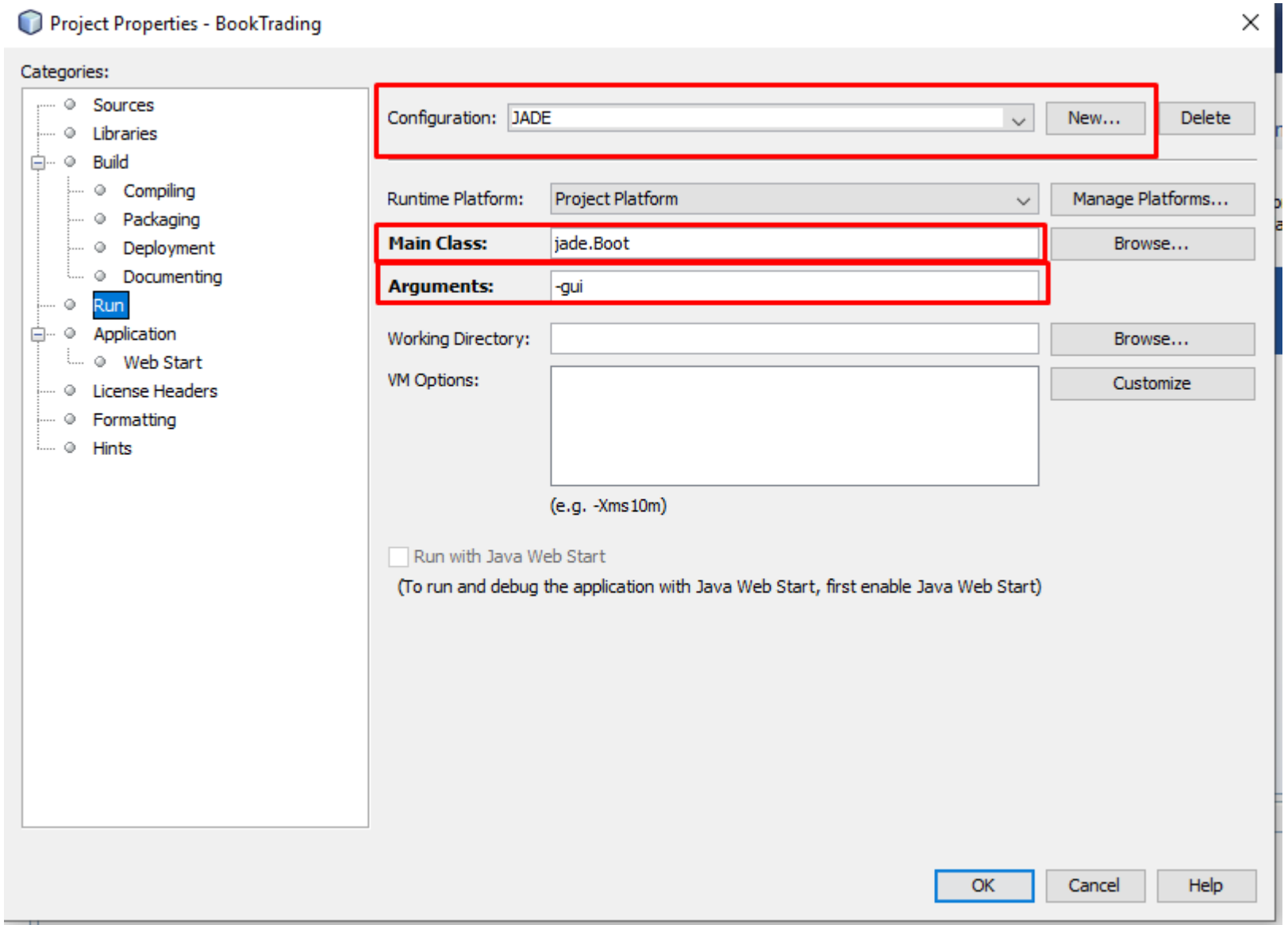
- The `search()` static method of the `DFService` class can be used as exemplified in the code used by the Book buyer agent to dynamically find all agents that provide a service of type “book-selling”.

```
addBehaviour(new TickerBehaviour(this, 60000) {
  protected void onTick() {
    // Update the list of seller agents
    DFAgentDescription template = new DFAgentDescription();
    ServiceDescription sd = new ServiceDescription();
    sd.setType("book-selling");
    template.addServices(sd);
    try {
      DFAgentDescription[] result = DFService.search(myAgent, template);
      sellerAgents = new AID[result.length];
      for (int i = 0; i < result.length; ++i) {
        sellerAgents[i] = result[i].getName();
      }
    }
    catch (FIPAException fe) {
      fe.printStackTrace();
    }
```

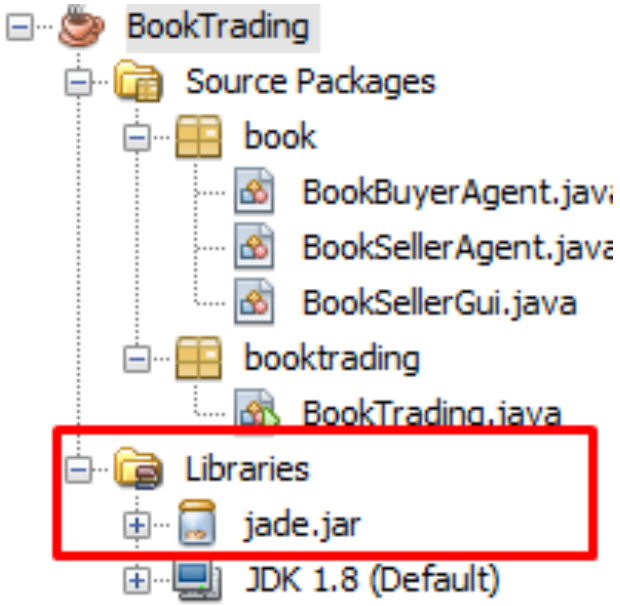
JADE – Netbeans Integration



JADE – Netbeans Integration



JADE – Netbeans Integration



REFERENCES

- G. Caire , «JADE Tutorial - JADE Programming For Beginners» 2009
- F. Bellifemine, G. Caire, T. Trucco,G. Rimassa, «JADE PROGRAMMER'S GUIDE», 2010
- Foundation for Intelligent Pyhsical Agents: http://www.fipa.org
- JADE : http://jade.tilab.com