1. JADE implementation short guide

Building an agent:

1) Extend your class from `jade.core.Agent`,
2) In optional method `getArguments()` process startup parameters
3) In mandatory method `setup()`
   a) Register content languages
   b) Register ontologies
   c) Start behaviors

Sending a message:

1) Create classes, which represents your message
2) Create ontology that describes your message
3) Create instance of your message and put it to list (`java.util.List`)
4) Create instance of class `ACLMessage`
5) Fill set of receivers
6) Fill name of ontology and language
7) Use method `fillMsgContent(ACLMessage m, List list)` to format content
8) Use method `send(ACLMessage m)`

Receiving a message:

1) Create behaviour for responding messages
2) Receive a message from the message buffer (if any) in the mandatory method `action()` of the behaviour
   a) Build the instance of the content message by the method `extractContent(ACLMessage m)`
   b) Process the message
   c) If necessary, send back some response

Starting the community

Run `jade.Boot` with the startup parameters that describes your agents

2. JADE implementation example

In this chapter, we will design, step-by-step, simple multi-agent system with two agent classes. **Source codes** for all examples are included in the **file tutorial_src.zip**.

2.1. Hello world

In this part you will create the agent that will write message “Hello world” and agent’s name on the system console.

This code solves the problem:

```
package tutorial;
import jade.core.Agent;
public class HelloWorldAgent extends Agent {
  protected void setup() {
    System.out.println("Hello World. My name is "+this.getLocalName());
  }
}
```

The first step is to extend your agent class from `jade.core.Agent`. The second is to override the `setup()` method and simply print "Hello World" message. The method `getLocalName()` returns the name of the agent.

You can run this example from command line as

```
> java jade.Boot john:tutorial.HelloWorldAgent
```

Your agent is started indirectly by class jade.Boot that processes startup parameters. The description of one agent is represented by two attributes. The first one is the name of the agent and the second one, after colon, is the name of the class that represents your agent.

You can see following message in system console

```
Hello World. My name is john
```

After that you can press Ctrl-C for stop processing.

2.2. Hello world with startup parameters

If your agent needs to get some information from the command line, you can extend the `HelloWorldAgent` class with the method as follows:

```
package tutorial;
import jade.core.Agent;
public class HelloWorldAgentWithParameters extends Agent {

  private String service;

  protected void setup() {
    Object[] args = getArguments();
    service = String.valueOf(args[0]);

    System.out.println("Hello World. My name is "+this.getLocalName()+
        " and I provide "+service+" service.");
  }
}
```

This example shows how to process startup parameters. The class `HelloWorldAgentWithParameters` has one private attribute `service` that is indicated as parameter in the command line and is received by the method `getArguments()`. In the next step `args` is transformed into its string representation.

You can test this example from command line as

```
> java jade.Boot john:tutorial.HelloWorldAgentWithParameters(printing)
```

The parameters are placed in brackets after the name of the class – without the spaces.
You can see on the system console:

```
Hello World. My name is john and I provide printing service.
```

You can also run two agents with same class and different parameters, e.g.

```
> java jade.Boot john:tutorial.HelloWorldAgentWithParameters(printing)
       jack:tutorial.HelloWorldAgentWithParameters(laminating)
```

This test gets these results:

```
Hello World. My name is john and I provide printing service.
Hello World. My name is jack and I provide laminating service.
```

2.3. Sending simple messages

In this part, we will develop two agents. The first one sends to the second one simple message. The agents use Agent Communication Language (ACL) for the communication, and each message is represented by the class `jade.lang.acl.ACLMessage`.

At the beginning, the agent `SenderAgent` sends the message to the agent with the name `jack`. The `ReceiverAgent` is waiting for the messages and, if some message arrives, it prints the message on the console.

The class `SenderAgent` is very simple. It creates new instance of the `ACLMessage` and fills the `receiver` and the `content` attribute. At the end it uses the method `send(ACLMessage m)`.

```
package tutorial.simplemessage;

import jade.core.AID;
import jade.core.Agent;
import jade.lang.acl.ACLMessage;

public class SenderAgent extends Agent {
  protected void setup() {
    System.out.println("Hello. My name is "+this.getLocalName());
    sendMessage();
  }

  private void sendMessage() {
    AID r = new AID ("jack@"+getHap(), AID.ISGUID);
    ACLMessage aclMessage = new ACLMessage(ACLMessage.REQUEST);
    aclMessage.addReceiver(r);
    aclMessage.setContent("Hello! How are you?");
    this.send(aclMessage);
  }
}
```

Instead of `AID r = new AID ("jack@"+getHap(), AID.ISGUID);` you can also use the statement `AID r = new AID ("jack", AID.ISLOCALNAME);` for the same result.

Receiving messages is little more complicated than sending messages. You must write some behaviour that is responsible for processing the incoming messages. You have to add the behaviour in the `setup()` method to the list of behaviours.

```
package tutorial.simplemessage;

import jade.core.Agent;

public class ReceiverAgent extends Agent {

  protected void setup() {
    System.out.println("Hello. My name is "+this.getLocalName());
    addBehaviour(new ResponderBehaviour(this));
  }
}
```

The class `ResponderBehaviour` is extended from the `SimpleBehaviour` class, that is an abstract class. Therefore you must override its methods `action()` and `done()`. In the method `action()` you process incoming messages. The method `done()` returns true only if the behaviour is finished. In our case this method returns always false. You must also specify, which messages may be processed by this behaviour. In our case, this behaviour processes only messages with performative REQUEST. This constraint is specified in the class `MessageTemplate`.

```
package tutorial.simplemessage;

import jade.core.Agent;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
import jade.core.behaviours.SimpleBehaviour;

public class ResponderBehaviour extends SimpleBehaviour {

private static final MessageTemplate mt =
        MessageTemplate.MatchPerformative(ACLMessage.REQUEST);

  public ResponderBehaviour(Agent agent) {
    super(agent);
  }

  public void action() {
    ACLMessage aclMessage = myAgent.receive(mt);

    if (aclMessage!=null) {
        System.out.println(myAgent.getLocalName()+": I receive message.\n"+aclMessage);
    } else {
        this.block();
      }
  }

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

You can use alternatively

```
ACLMessage aclMessage = myAgent.blockingReceive(mt, 100);
System.out.println(myAgent.getLocalName()+": I receive message.\n"+aclMessage);
```

instead of the if-block, but then the method `done()` has to return `true`.

You can run this example as:

```
>java jade.Boot john:tutorial.simplemessage.SenderAgent
      jack:tutorial.simplemessage.ReceiverAgent
```

As a result you can see something like:

```
Hello. My name is john
Hello. My name is jack
jack: I receive message.
(REQUEST
 :sender  ( agent-identifier  :name john@bubik-small:1099/JADE)
 :receiver  (set ( agent-identifier  :name jack@bubik-small:1099/JADE) )
 :content  "Hello! How are you?"
 )
```

2.4. Sending data messages using ontology

In previous step, we described how to send and receive a simple message. In most cases we need to send an object to another agent. Hence, we need some string representation of the object and mechanism for encoding and decoding the object into string and vice versa. This problem is already solved in JADE. You must follow these three steps:

1. Design the class that represents data which you want to send
2. Design the ontology that describes binding between java object and its string representation
3. Register the codec for the content language and register the ontology

In this case we will create the class `Person` with two attributes – `name` and `age`. As a content language we will use SL0. Because SL0 does not support named slots in the top level, we have

to build one more class `PersonMessage` that contains only one instance of the class `Person`. Finally, we will develop the `PersonOntology`.

The class `Person` must contain set and get methods for all attributes. In our case it looks like:

```
package tutorial.datamessage;

public class Person {
  private String name;
  private Integer age;

  public Person() {
  }

  public void setName(String name) {
    this.name = name;
  }

  public void setAge(Integer age) {
    this.age = age;
  }

  public String getName() {
    return name;
  }


  public Integer getAge() {
    return age;
  }

  public String toString() {
    return "(person\n\t:name " + getName() + "\n\t:age " + getAge() + ")";
  }
}
```

```
package tutorial.datamessage;

public class PersonMessage {

  private Person person = new Person();

  public PersonMessage() {
  }

  public void set_0(Person a) {
    person = a;
  }

  public void setPerson(Person person) {
    set_0(person);
  }

  public Person get_0() {
    return person;
  }

  public Person getPerson() {
    return get_0();
  }

}
```

The class `PersonOntology` describes binding between java objects and theirs string representations. Both classes have theirs own roles.

```
package tutorial.datamessage;

import jade.onto.Frame;
import jade.onto.Ontology;
import jade.onto.DefaultOntology;
import jade.onto.SlotDescriptor;
import jade.onto.OntologyException;

import jade.onto.basic.*;

import java.util.*;

public class PersonOntology {

  public final static String NAME = "person-ontology";

  private static Ontology theInstance = new DefaultOntology();

  public static Ontology instance() {
    return theInstance;
  }

  private static void initInstance() {
    try {
      theInstance.joinOntology(BasicOntology.instance());

      theInstance.addRole(
          "person-message",
          new SlotDescriptor[]{
          new SlotDescriptor(Ontology.FRAME_SLOT, "person", Ontology.M),
          }, PersonMessage.class);

      theInstance.addRole(
          "person",
          new SlotDescriptor[]{
          new SlotDescriptor("name", Ontology.PRIMITIVE_SLOT,
                Ontology.STRING_TYPE, Ontology.M),
          new SlotDescriptor("age", Ontology.PRIMITIVE_SLOT,
                Ontology.INTEGER_TYPE, Ontology.M),
          }, Person.class);

    }
    catch (OntologyException oe) {
      oe.printStackTrace();
    }
  }

  static {
    initInstance();
  }
}
```

The sender agent has similar skeleton like in the previous case – methods `setup()` and `sendMessage()`. In the method `setup()` you must register codec for the SL0 content language and the ontology. You can send some message after registration only. In the method `sendMessage()` we create some instance of the class `Person` that we want to send to the another agent. We put this instance into the `PersonMessage`, which is put into the list. Next, we create the instance of the `ACLMessage` and set the receiver agent address. You must also fill the language and the ontology slot. The method `fillMsgContent` takes your data, codec and ontology and transforms this data into the string representation. After that you can send the prepared message.

```
package tutorial.datamessage;

import java.util.ArrayList;
import jade.core.AID;
import jade.core.Agent;
import jade.lang.acl.ACLMessage;
import jade.lang.sl.SL0Codec;
import jade.onto.basic.Action;

public class SenderAgent extends Agent {

  protected void setup() {
    System.out.println("Hello. My name is " + this.getLocalName());

    this.registerLanguage(SL0Codec.NAME, new SL0Codec());
    this.registerOntology(PersonOntology.NAME, PersonOntology.instance());

    sendMessage();
  }

  private void sendMessage() {
    try {
      ArrayList list = new ArrayList();

      Person person = new Person();
      person.setName("Marry");
      person.setAge(new Integer(22));
      PersonMessage personMessage = new PersonMessage();
      personMessage.setPerson(person);

      list.add(personMessage);

      AID r = new AID("jack@"+getHap(), AID.ISGUID);
      ACLMessage aclMessage = new ACLMessage(ACLMessage.REQUEST);
      aclMessage.addReceiver(r);
      aclMessage.setLanguage(SL0Codec.NAME);
      aclMessage.setOntology(PersonOntology.NAME);

      this.fillMsgContent(aclMessage, list);

      this.send(aclMessage);
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

The `ReceiverAgent` must also register same language and ontology like the `SenderAgent`.

```
package tutorial.datamessage;

import jade.core.Agent;
import jade.lang.sl.SL0Codec;

public class ReceiverAgent extends Agent {

  protected void setup() {
    System.out.println("Hello. My name is "+this.getLocalName());

    this.registerLanguage(SL0Codec.NAME, new SL0Codec());
    this.registerOntology(PersonOntology.NAME, PersonOntology.instance());

    addBehaviour(new ResponderBehaviour(this));
  }
}
```

The `ResponderBehaviour` has the same skeleton like in the previous case. The method `extractMsgContent` reconstructs object from string representation. Finally, your object is printed on the console.

```
package tutorial.datamessage;

import jade.util.leap.ArrayList;
import jade.core.Agent;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
import jade.core.behaviours.SimpleBehaviour;
import jade.onto.basic.Action;

public class ResponderBehaviour extends SimpleBehaviour {

  private final static MessageTemplate mt =
       MessageTemplate.MatchPerformative(ACLMessage.REQUEST);

  public ResponderBehaviour(Agent agent) {
    super(agent);
  }

  public void action() {
      ACLMessage aclMessage = myAgent.receive(mt);
      if (aclMessage != null) {
        try {
          ArrayList list = (ArrayList) myAgent.extractMsgContent(aclMessage);
          PersonMessage personMessage = (PersonMessage) list.get(0);
          Person person = personMessage.getPerson();
          System.out.println(myAgent.getLocalName() + ": I receive message\n" +
              aclMessage + "\nwith content\n" + person);
        }
        catch (Exception ex) {
          ex.printStackTrace();
        }
      }
      else {
        this.block();
      }
  }

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

You can start this example from command line as:

```
> java jade.Boot john:tutorial.datamessage.SenderAgent
  jack:tutorial.datamessage.ReceiverAgent
```

You can see the results on the console:

```
Hello. My name is john
Hello. My name is jack
jack: I receive message
(REQUEST
 :sender  ( agent-identifier  :name john@bubik-small:1099/JADE)
 :receiver  (set ( agent-identifier  :name jack@bubik-small:1099/JADE) )
 :content  "((person-message (person :name Marry :age 22 ) ) ) "
 :language  FIPA-SL0
 :ontology  person-ontology
)
with content
(person
        :name Marry
        :age 22)
```

This example is fully functional two-agent system. The agents contain full negotiation support. Each agent can register several services and process appropriate requests in its behaviors. Every

other agent can search these services and use them for achieving its goal (e.g. task planning). By the way you can also create ontologies automatically if you use Protégé 2000 with Java Bean Generator.

The subsequent sections 2.5 Registering services and 2.6 Searching services are additional and not subject of the course.

2.5. Registering services

If the agent provides some services, he should register them at DF agent. Another agents can search for this service and use your agent for solving its problem.

Services registeration is usualy placed at the startup of the agent. Hence, the `setup()` method is the best place for registering services. You can simply extend previous example with the new method `registerService()` that is invoked in the method `setup()`.

```
package tutorial.df;

import jade.core.Agent;
import jade.domain.FIPAException;
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.FIPAAgentManagement.DFAgentDescription;

public class ProviderAgent extends Agent {

  private String service;

  protected void setup() {
    Object[] args = getArguments();
    service = String.valueOf(args[0]);

    System.out.println("Hello. My name is "+this.getLocalName()+
      " and I provide "+service+" service.");
    registerService();
  }

  private void registerService() {
    DFAgentDescription dfd = new DFAgentDescription();
    dfd.setName(this.getAID());

    ServiceDescription sd = new ServiceDescription();
    sd.setType(service);
    sd.setName(service);

    dfd.addServices(sd);
    try {
      DFService.register(this, dfd);
    }
    catch (FIPAException e) {
      System.err.println(getLocalName() +
        " registration with DF unsucceeded. Reason: " + e.getMessage());
      doDelete();
    }
  }
}
```

The method `registerService()` creates the instance of the class `DFAgentDescription`, fills it with proper data and sends it to the DF agent using method `DFService.register(Agent a, DFAgentDescription d)`. If some problem occurs, the registration throws exception and the agent prints error message on console and kills itself -`doDelete()`. The class `DfAgentDescription` contains list of `ServiceDescription` classes. Hence, we create new instance of the `ServiceDescription` and fill it with proper data, that we got from the startup

parameters (attribute `service`). After that, we add it to the list of the services by the method `dfd.addServices(ServiceDescription sd)`.
If you run this example with the parameter `-gui`, JADE platform opens the frame with some interesting information. In this case, only one item the menu `Tools->Show the DF GUI` is important. It opens new frame with the DF information. You can see all registered agents and its services.

Run this example as:

```
> java jade.Boot -gui  john:tutorial.df.ProviderAgent(printing)
                       jack:tutorial.df.ProviderAgent(laminating)
```

2.6. Searching services

If some agent has registered his services, another agent can search them. In this part, we will develop the `SearchAgent`. The structure of the code is the similar as previous examples.

```
package tutorial.df;

import jade.util.leap.List;
import jade.util.leap.Iterator;

import jade.core.Agent;
import jade.domain.FIPAException;
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.FIPAAgentManagement.SearchConstraints;

public class SearchAgent extends Agent {

  protected void setup() {
    System.out.println("Hello. I am "+this.getLocalName()+".");
    this.searchAgents();
  }

  private void searchAgents() {
    DFAgentDescription dfd = new DFAgentDescription();
    SearchConstraints c = new SearchConstraints();
    doWait(2000);

    try {
      DFAgentDescription[] result = DFService.search( this, dfd);
      for( int i=0; i<result.length; i++) {
          String out = result[i].getName().getLocalName()+"@"+getHap()+" provide";
          Iterator iter2 = result[i].getAllServices();
           while (iter2.hasNext()) {
                ServiceDescription sd = (ServiceDescription)iter2.next();
                out += " "+sd.getName();
           }
           System.out.println( this.getLocalName()+": "+out);
      }
    }
    catch (Exception fe) {
      System.err.println(getLocalName() +
        " search with DF is not succeeded because of " + fe.getMessage());
      doDelete();
    }
  }
}
```

The method `searchAgents()` sends search request to the DF. If you don't specify the searching parameters and constraints, the DF will return all registered agents. The result is printed on the console.

You can use agents from the previous paragraph and start the community as

```
> java jade.Boot -gui  john:tutorial.df.ProviderAgent(printing)
  jack:tutorial.df.ProviderAgent(laminating)
  boss:tutorial.df.SearchAgent
```

If the agents are started in the right sequence, you should see the result like:

```
Hello. My name is john and I provide printing service.
Hello. My name is jack and I provide laminating service.
Hello. I am boss.
boss: john@bubik-small:1099/JADE provide printing
boss: jack@bubik-small:1099/JADE provide laminating
```