Artificial Intelligence
Agent Oriented Software Engineering

Maurizio Martelli, Viviana Mascardi
{martelli, mascardi}@disi.unige.it

University of Genoa – Department of Computer and Information Science
Disclaimer

- This presentation may contain material protected by copyright laws.
- In particular, many lessons are based on the material that Russel and Norvig made available on the Web, as part of their book “Artificial Intelligence: A Modern Approach”
- We made this material available on the Web only to ensure timely dissemination among the students of the SIIT Master of the University of Genova, and is meant only for students’ personal use.
- Any use different from the students’ personal use is prohibited.
Outline

1. Introduction
2. AOSE methodologies
3. Notation techniques
4. MAS infrastructures

The features of agent-based systems are well suited to tackle the complexity of developing software in modern scenarios:

1. the autonomy of application components reflects the intrinsically decentralised nature of modern distributed systems and can be considered as the natural extension to the notions of modularity and encapsulation for systems that are owned by different stakeholders;
2 the flexible way in which agents operate and interact (both with each other and with the environment) is suited to the dynamic and unpredictable scenarios where software is expected to operate;

3 the concept of agency provides for a unified view of AI results and achievements, by making agents and MASs act as sound and manageable repositories of intelligent behaviours.
In the last few years, together with the increasing acceptance of agent-based computing as a novel software engineering paradigm, there has been a great deal of research related to the identification and definition of suitable models and techniques to support the development of complex software systems in terms of MASs.

This research, can be roughly grouped under the term Agent-Oriented Software Engineering (AOSE).
The Promise of AOSE for distributed systems engineering

Today’s software engineering approaches are increasingly adopting abstractions approaching that of agent-based computing.

This trend can be better understood by recognising that the vast majority of modern distributed systems scenarios are intrinsically prone to be developed in terms of MASs, and that modern distributed systems are already de facto MASs, i.e., they are indeed composed of autonomous, situated, and social components.
As far as autonomy is concerned, almost all of today’s software systems already integrate autonomous components. At its weakest, autonomy reduces to the ability of a component to react to and handle events, as in the case of graphical interfaces or simple embedded sensors. However, in many cases, autonomy implies that a component integrates an autonomous thread of execution, and can execute in a proactive way.
Today’s computing systems are also typically situated.

For example, control systems for physical domains and sensor networks, are built to explicitly manage data from the surrounding physical environment, and take into account the unpredictable dynamics of the environment.

Mobile and pervasive computing applications recognise (under the general term of context-awareness) the need for applications to model explicitly environmental characteristics and data.

Internet applications are also conceived for being aware of the features and the dynamics of the environment where they work, namely, the Web.
Sociality in modern distributed systems comes in different flavors:

1. the capability of components of supporting dynamic interactions, i.e., interaction established at run-time with previously unknown components;
2. the somewhat higher interaction level, overcoming the traditional client-server scheme;
3. the enforcement of some sorts of societal rules governing the interactions.
Artificial Intelligence is mainly concerned with building intelligent systems: the very name of “Artificial Intelligence” literally suggests the notion of artifacts exhibiting intelligent behaviour.

Therefore, AI can be considered as an engineering field (dealing with constructive concerns), rather than simply a scientific one (dealing with understanding and predicting intelligent systems behaviour).
The Promise of AOSE for intelligent systems engineering

- As already seen in the previous lessons, nearly 30 years of AI research were conducted by having research groups concentrate on single, isolated aspects of AI (like, say, artificial vision, knowledge representation, planning).
- There were few attempts of producing a reasonable set of conceptual and practical tools, which could promote the integration of such a vast amount of research findings into the mainstream practice of software development.
- This is where agents are actually becoming a key abstraction in today’s AI.
The very notion of agents provides a uniform conceptual space where all the findings of the AI field can be easily framed and related, and eventually find mainstream acceptance.
The three current mainstream research directions in AOSE, besides those concerned with agent architectures and programming languages, already discussed in the previous lessons, are:

1. AOSE methodologies
2. Notation techniques
3. MAS infrastructures

We will discuss each of them in the next sections of this lesson.
Traditional methodologies of software development, driving engineers from analysis to design and development, must be tuned to match the abstractions of agent-oriented computing.

To this end, a variety of novel methodologies to discipline and support the development process of a MAS have been defined in the past few years, clarifying the various sets of abstractions that must come into play during MAS development and the duties and responsibilities of software engineers.
AOSE methodologies

- AOSE methodologies, like any other ones, have two components: one that describes the process elements of the approach, and a second that focuses on the work products and their documentation.

- AOSE methodologies also need to contain sufficient abstractions to fully model and support agents and MASs. They need to focus on an organized society of agents playing roles within an environment. Within such an MAS, agents interact according to protocols determined by the agents’ roles.

- Note that when we speak of an AOSE methodology, we generally do not mean a methodology that is itself constructed on agent-oriented principles, but one that is oriented at the creation of agent-based software.
Genealogy of some AOSE methodologies

- MAS-CommonKADS (+AI/KE)
- INGENIAS
- MESSAGE
- Adelfe
- RAP
- RUP
- AOR
- OMT
- Fusion
- OPEN
- OPEN
- Prometheus Roadmap
- OO
- Gaia
- Agent OPEN
- Tropos
- AAII
- MaSE
- OMT
- PASSI


The Prometheus Design Tool:
Prometheus

The Prometheus methodology consists of three phases.

1. The *system specification phase* focuses on identifying the basic functionalities of the system, along with inputs (percepts), outputs (actions) and any important shared data sources.

2. The *architectural design phase* uses the outputs from the previous phase to determine which agents the system will contain and how they will interact.

3. The *detailed design phase* looks at the internals of each agent and how it will accomplish its tasks within the overall system.
Prometheus

```bash
java -jar /home/mascardi/Programs/PDT_3.3/PDT_3.3.jar
```

```
Open /home/mascardi/Programs/PDT_3.3/BookStore_3.0.pd
```
This phase consists of the following steps, which are interleaved and iterated until the specification is considered sufficiently complete:

1. Identification of actors and their interactions with the system, in the form of percepts and actions;
2. Developing scenarios (by means of use cases) illustrating the system’s operation;
3. Identification of the system goals and sub-goals;
4. Identifying any external data;
5. Grouping goals and other items into the basic roles of the system.
This phase uses artefacts produced in the System Specification Phase to determine what agent types will be included in the system and the interaction between these agents. The steps in this phase are:

1. Determine the agent types
2. Develop the interaction protocols
3. Develop the system overview diagram
This phase uses artefacts produced in the Architectural Design Phase to define the internals of every agent in the system and to specify how agents accomplish their overall tasks.

- Each agent is refined in terms of its capabilities, internal events, plans, and data structures.
- Each capability has a capability overview diagram that captures the structure of the plans within this capability and the events that are associated with these plans.
- The dynamic behaviour is described by process diagrams based on the interaction protocols identified in the previous phase.
We will go through the design and implementation of a small library management system. This system should be able to do the following things:

- Allow for checkout of books, providing a return date to the customer
- Allow for return of books
- Allow for reservation of unavailable books
- Allow for notification of overdue books
- Allow for notification of arrival of reserved books
Checkout books
  – Record book code to the user id checked out list.
  – Provide return date.

Return books
  – Remove book code from the user id

Reserve unavailable books
  – Record book code as reserved for user id
  – Show the current due date for the book
System specification: Goals and subgoals

- Give notification for overdue books
  - Access book record at the start of the day
  - Send email for overdue books
- Give notification of arrival of reserved books
  - Access the reserved list for user
  - Send email notification
System specification: Goals and subgoals

- **Checkout**
  - Record book code as checked out
  - Provide return date

- **Reserve**
  - Record book code as reserved
  - Show current due date
System specification: Goals and subgoals

- Return
  - Remove book code from checkout list
  - Find overdue books at the start of the day
  - Notify overdue
    - Send email for overdue books
System specification: Goals and subgoals

- Notify arrival
  - Find the reserved user
  - Send email mentioning arrival
Depending on the system we can identify 5 different scenarios:

1. When the user comes to checkout the book.
2. When the user returns the book.
4. When the user asks to reserve a book.
5. When the reserved book arrives.
System specification: Scenarios

- Checkout scenario
- Reserve scenario
- Return scenario
- Overdue scenario
- Arrival scenario
System specification: Checkout scenario

1. Request for checkout
2. Provide return date
3. Record book code as checked out
4. Provide book
System specification: Return scenario

1. Book returned
2. Remove book code from checkout list
System specification: Reserve scenario

1. Request for reservation
2. Record book code as reserved
3. Show current due date
4. Provide current due date
System specification: Arrival scenario

1. Reserved book arrives
2. Find the reserved user
3. Send arrival email
System specification: Overdue scenario

1. Start of the day
2. Find overdue books at the start of the day
3. Send overdue email
System specification: Role grouping

1. Checkout books
2. Return books
3. Overdue books
4. Reserve books
5. Send Arrival Notification
System specification: Checkout books role

- Request for checkout
- Checkout books
  - Provide book
  - Checkout
    - Record book code as checked out
    - Provide return date
System specification: Return and overdue books roles

- **Book returned**
  - Return books
  - Return
  - Remove book code from checkout list

- **Start of day**
  - Overdue books
  - Notify overdue
  - Find overdue books at the start of the day
  - Send email for overdue books
  - Send overdue email
System specification:
Reserve books and send arrival notification roles
In this phase we need to identify what type of data need to be stored in the system as beliefs. In our running example, we need to keep the information of all books that have been checked out and also the books that have been reserved.

- The checkout books role and return books role update the checkout belief and the overdue books role uses this to send overdue email.
- The reserve books role updates the reserve belief and the arrival notification uses it.
Architectural design: Data coupling for checkout role
Architectural design:
Data coupling for reserve role
Now we group the roles and identify three agents to carry out these roles in the system. The agents identified are:

- Checkout Agent
- Reservation Agent
- Overdue Agent
Architectural design: Checkout agent-role coupling

- **CheckoutAgent**
  - Checkout books
  - Return books
Architectural design: Reservation agent-role coupling
Architectural design: Overdue agent-role coupling
Architectural design: Agent acquaintance diagram
Architectural design: System overview diagram
Detailed design: Capabilities of the checkout agent

- Checkout Capability
- Return Capability
- Get Return Date Capability
Detailed design: Capabilities of the checkout agent
Detailed design: Checkout plan

- Request for checkout
- Checkout book
- Checkout database
- Provide book
Detailed design: Return plan

- Book returned
  - Checkin book
    - Checkout database
Detailed design: Get return date plan

Diagram:
- Get return date
- Get return date plan
- Return date
- Checkout database
Detailed design: Capabilities of the reservation agent

- Reservation Capability
- Arrival Notification Capability
Detailed design: Capabilities of the reservation agent

- Request for reservation
- Reserved book arrives
- ReservationCapability
- Reserve database
- Get return date
- Return date
- Provide current due date
- ArrivalNotificationCapability
- Send arrival email
Detailed design: Reservation plan
Detailed design: Arrival notification plan

- Reserved book arrives
  - Send email plan
    - Reserve database
      - Send arrival email
Detailed design: Capabilities of the overdue agent

- Find overdue books Capability
Detailed design: Find overdue books plan

1. Start of day
2. Find overdue books
   - Get return date
   - Return date
   - Send overdue email
PDT demo

- Let us start a demo session...
Specific notation techniques to express the outcome of the various phases of a MAS development process are needed, because traditional object- and component-oriented notation techniques cannot easily apply.

In this context, the AUML proposal, extending standard UML toward agent-oriented systems, is the subject of a great deal of research and it is rapidly becoming a de facto standard.

FIPA ACL represents an effort to standardize agent communication.
FIPA specifications: an overview

http://www.fipa.org/repository/bysubject.html
FIPA specifications: communication

http://www.fipa.org/repository/aclspecs.html
### FIPA Message Structure

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Category of Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>performative</td>
<td>Type of communicative acts</td>
</tr>
<tr>
<td>sender</td>
<td>Participant in communication</td>
</tr>
<tr>
<td>receiver</td>
<td>Participant in communication</td>
</tr>
<tr>
<td>reply-to</td>
<td>Participant in communication</td>
</tr>
<tr>
<td>content</td>
<td>Content of message</td>
</tr>
<tr>
<td>language</td>
<td>Description of Content</td>
</tr>
<tr>
<td>encoding</td>
<td>Description of Content</td>
</tr>
<tr>
<td>ontology</td>
<td>Description of Content</td>
</tr>
<tr>
<td>protocol</td>
<td>Control of conversation</td>
</tr>
<tr>
<td>conversation-id</td>
<td>Control of conversation</td>
</tr>
<tr>
<td>reply-with</td>
<td>Control of conversation</td>
</tr>
<tr>
<td>in-reply-to</td>
<td>Control of conversation</td>
</tr>
<tr>
<td>reply-by</td>
<td>Control of conversation</td>
</tr>
</tbody>
</table>

FIPA Communicative Acts

- Accept Proposal
- Agree
- Cancel
- Call for Proposal
- Confirm
- Disconfirm
- Failure
- Inform
- Inform If
- Inform Ref
- Not Understood
- Propagate
- Propose
- Proxy
Notation techniques

FIPA Communicative Acts

- Query If
- Query Ref
- Refuse
- Reject Proposal
- Request
- Request When
- Request Whenever
- Subscribe

Accept proposal: The action of accepting a previously submitted proposal to perform an action.
A tuple consisting of an action expression denoting the action to be done, and a proposition giving the conditions of the agreement.
accept-proposal is a general-purpose acceptance of a proposal that was previously submitted (typically through a propose act). The agent sending the acceptance informs the receiver that it intends that (at some point in the future) the receiving agent will perform the action, once the given precondition is, or becomes, true.

The proposition given as part of the acceptance indicates the preconditions that the agent is attaching to the acceptance. A typical use of this is to finalize the details of a deal in some protocol. For example, a previous offer to “hold a meeting anytime on Tuesday” might be accepted with an additional condition that the time of the meeting is 11.00.

Note for future extension: an agent may intend that an action become done without necessarily intending the precondition. For example, during negotiation about a given task, the negotiating parties may not unequivocally intend their opening bids: agent a may bid a price p as a precondition, but be prepared to accept price p'.
$<i, \text{accept-proposal} (j, <j, \text{act}>, \phi)> \equiv$

$<i, \text{inform} (j, I_i \text{ Done} (<j, \text{act}>, \phi)>$

FP: $B_i \alpha \land \neg B_i (B_f j \alpha \lor U_f j \alpha)$

RE: $B_j \alpha$

Where:

$\alpha = I_i \text{ Done} (<j, \text{act}>, \phi)$
Agent $i$ informs $j$ that it accepts an offer from $j$ to stream a given multimedia title to channel 19 when the customer is ready. Agent $i$ will inform $j$ of this fact when appropriate.

(accept-proposal
 :sender (agent-identifier :name i)
 :receiver (set (agent-identifier :name j))
 :in-reply-to bid089
 :content
   "((action (agent-identifier :name j)
       (stream-content movie1234 19))
  (B (agent-identifier :name j)
     (ready customer78)))"
 :language fipa-sl)


When it makes sense to reuse portions of UML, then do it; when it doesn’t make sense to use UML, use something else or create something new.
AIPs in AUML

- AUML is very suitable to represent Agent Interaction Protocols (AIPs).
- We consider the AUML interaction diagrams specification proposed by the FIPA Technical Modeling Committee, 2003, (FIPA modeling: Interaction diagrams, first proposal, july 02 2003).
AIPs in AUML

- An AIP is usually defined in relation to a particular agent activity.
- AIPs in AUML extend UML 2.0 sequence diagrams.
- The protocol name is contained in a box on the upper-left corner of the diagram and is preceded by the keyword `sd` (sequence diagram).
AIPs in AUML: Actors, roles, and lifelines

- An AIP consists of some lifelines; for each lifeline, a rectangular box contains information about the agent to which that lifeline belongs, the role it plays in the MAS, and its class.

- The syntax used is `agent-name/ agent-role: agent-class`. 
AIPs in AUML: Message exchange

- Labelled arrows between two lifelines represent messages.
- Messages must satisfy standardised communication languages (e.g. FIPA-ACL or KQML) which allow to specify the “speech act” (or “performative”) of the message and the fields that it is expected to have.
AIPs in AUML: Message exchange

- AUML supports different interaction operators:
  - A box, with an interaction operator given in its top-left corner, can surround a part of the sequence diagram.
  - Boxes can recursively contain messages and other boxes, and can be divided into a number of regions separated from each other by heavy horizontal dashed lines.
  - Each region can include a condition depicted as text in square brackets.
  - If the condition is satisfied, then that region is selected and the activities inside it are performed.
The weak sequencing operator as defined in UML 2.0 means a weak sequencing in a sequence of messages, that is to say, the messages within this box on the same lifeline are ordered but it is not possible to make any assumption for message ordering coming from different lifelines in the same box.
Alternative means that there are several paths to follow, and that agents have to choose at most one to continue.

Guards are associated with alternatives: when a guard is evaluated to true, that alternative path is chosen.

An else alternative may be optionally present: it is entered if no guard evaluates to true.

If no guard is evaluated to true, and no else alternative is provided, the alternative box of the diagram is not executed.
The option operator only considers one path in the region.

If the condition associated with this path is evaluated to true, then the path is executed, else nothing happens and the interaction follows after this portion of the diagram.
The loop operator allows designers to represent that an ordered set of activities has to be applied several times.

Designers can use either lower and upper bounds or a boolean expression.

As long as conditions are satisfied, the loop is executed and the messages are sent and received.
AIPs in AUML: Parameters

- Parameters are written separately outside the diagram in a comment, stereotyped as <<parameters>>.
- AIPs can have many parameters such as content language for messages, agent communication language (ACL), ontology, etc.
Visual representation of AIPs in AUML
Visual representation of AIPs in AUML
Textual representation of AIPs in AUML (after Winikoff)

start FruitMarket
agent r reader
agent p publisher
message r p request availability_and_price(fruit(F))
box alternative
  box seq
  message p r inform available(fruit(F))
  message p r propose buy(fruit(F),price(EuroForKg))
  box alternative
    message r p request delivery_modes
    message p r inform delivery_mode(ListOfModes)
    message r p request accepted_payment_methods
    message p r inform accepted_payment_methods(ListOfMethods)
...
Open `C:\Programmi\PDT\Examples\protocol.txt` to be cut and pasted into Entities ⇒ Edit Protocols
To support the development and execution of MASs, novel tools and novel software infrastructures are needed.

In this context, various tools are being proposed to transform standard MAS specifications (i.e., AUML specifications) into actual agent code, and a variety of middleware infrastructures have been deployed to provide proper services supporting the execution of distributed MASs.
FIPA abstract architecture:
Infrastructures ad-hoc for MASs

- There are many free and/or open source implementations of the latest FIPA standards, including
  - FIPA-OS (http://sourceforge.net/projects/fipa-os/)
  - JADE (http://jade.tilab.com/)
  - ZEUS (http://sourceforge.net/projects/zeusagent)

- There are also several high profile commercial toolkits such as Agent Oriented Software’s JACK, Whitestein’s Living Systems Technology Suite and Lost Wax’s agent framework.
Other infrastructures suitable for MAS deployment

- The Extensible Markup Language (XML) is the universal format for structured documents and data on the Web.
- The Resource Description Format (RDF) is a framework for describing and interchanging metadata.
- Web services are software systems designed to support interoperable machine-to-machine interaction over a network (W3C definition). Because this definition encompasses many different systems, in common usage the term usually refers to those services that use SOAP-formatted XML envelopes and have their interfaces described by WSDL.
Grid computing provides the ability to perform higher throughput computing by taking advantage of many networked computers to model a virtual computer architecture able to distribute process execution across a parallel infrastructure. Grid computing is supported by both commercial and open-source toolkits.
A peer-to-peer (or P2P) computer network relies primarily on the computing power and bandwidth of the participants in the network rather than concentrating it in a relatively low number of servers. P2P networks are typically used for connecting nodes via largely ad hoc connections. They are used for sharing content files containing audio, video, data or anything in digital format, as well as realtime data, such as telephony traffic. Examples of P2P systems are Napster, SETI@HOME (Search for Extraterrestrial Intelligence), Gnutella.
**JADE**

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

It simplifies the implementation of MASs through a middle-ware that complies with the FIPA specifications and through a set of tools that supports the debugging and deployment phases.

The agent platform can be distributed across machines (which not even need to share the same OS) and the configuration can be controlled via a remote GUI.

The configuration can be even changed at run-time by moving agents from one machine to another one, as and when required.

JADE is completely implemented in Java language and the minimal system requirement is the version 1.4 of JAVA (the runtime environment or the JDK).
On-line tutorials

- **Tutorial 3** [http://www.iro.umontreal.ca/~%7EvaucherAgents/Jade/JadePrimer.html](http://www.iro.umontreal.ca/~%7EvaucherAgents/Jade/JadePrimer.html)
From command line: `java jade.Boot -gui`

![JADE demo GUI](image)
A type of agent is created by extending the `jade.core.Agent` class and redefining the `setup()` method.

Each Agent instance is identified by an AID (`jade.core.AID`).
- An AID is composed of a unique name plus some addresses.
- An agent can retrieve its AID through the `getAID()` method of the `Agent` class.

```java
import jade.core.Agent;

public class HalloWorldAgent extends Agent {

    protected void setup() {
        System.out.println("Hallo World! my name is "+getAID().getName());
    }
}
```
Local names, GUID and addresses

- Agent names are of the form `<local-name>@<platform-name>`
- The complete name of an agent must be globally unique.
- The default platform name is `<main-host>:`<main-port>/JADE
- The platform name can be set using the `-name` option
- Within a single JADE platform agents are referred through their names only.
- Given the name of an agent its AID can be created as
  - `AID id = new AID(localname, AID.ISLOCALNAME);`
  - `AID id = new AID(name, AID.ISGUID);`
- The addresses included in an AID are those of the platform MTPs and are ONLY used in communication between agents living on different FIPA platforms
Passing arguments to an agent

- It is possible to pass arguments to an agent
  - java jade.Boot .... a:myPackage.MyAgent(arg1 arg2)
  - The agent can retrieve its arguments through the `getArguments()` method of the `Agent` class

```java
protected void setup() {
    System.out.println("Hallo World! my name is "+getAID().getName());
    Object[] args = getArguments();
    if (args != null) {
        System.out.println("My arguments are:");
        for (int i = 0; i < args.length; ++i) {
            System.out.println("- "+args[i]);
        }
    }
}
```
Agent termination

- An agent terminates when its `doDelete()` method is called.
- On termination the agent’s `takeDown()` method is invoked (intended to include clean-up operations).

```java
protected void setup() {
    System.out.println("Hallo World! my name is "+getAID().getName());
    Object[] args = getArguments();
    if (args != null) {
        System.out.println("My arguments are: ");
        for (int i = 0; i < args.length; ++i) {
            System.out.println("- "+args[i]);
        }
    }
    doDelete();
}

protected void takeDown() {
    System.out.println("Bye...");
}
```
The Behaviour class

- The actual job that an agent does is typically carried out within “behaviours”
- Behaviours are created by extending the jade.core.behaviours.Behaviour class
- To make an agent execute a task it is sufficient to create an instance of the corresponding Behaviour subclass and call the addBehaviour() method of the Agent class.
- Each Behaviour subclass must implement
  - public void action(): what the behaviour actually does
  - public boolean done(): Whether the behaviour is finished
An agent can execute several behaviours in parallel, however, behaviour scheduling is not preemptive, but cooperative and everything occurs within a single Java Thread.

Behaviour switch occurs only when the action() method of the currently scheduled behaviour returns.
The agent execution model

1. **setup()**
   - Initializations
   - Addition of initial behaviours

2. **Agent has been killed (dodelete() method called)?**
   - Yes: **Get the next behaviour from the pool of active behaviours**
   - No: **b.action()**

3. **b.done()?**
   - Yes: **Remove current behaviour from the pool of active behaviours**
   - No: **takeDown()**

Highlighted in red the methods that programmers have to/can implement.
Behaviour types

• “One shot” behaviours.
  – Complete immediately and their `action()` method is executed only once.
  – Their `done()` method simply returns `true`.
  – `jade.core.behaviours.OneShotBehaviour` class

• “Cyclic” behaviours.
  – Never complete and their `action()` method executes the same operation each time it is invoked
  – Their `done()` method simply returns `false`.
  – `jade.core.behaviours.CyclicBehaviour` class

• “Complex” behaviours.
  – Embed a state and execute in their `action()` method different operation depending on their state.
  – Complete when a given condition is met.
Scheduling operations at given points in time

• JADE provides two ready-made classes by means of which it is possible to easily implement behaviours that execute certain operations at given points in time
  • WakerBehaviour
    – The `action()` and `done()` method are already implemented so that the `onWake()` method (to be implemented by subclasses) is executed after a given timeout
    – After that execution the behaviour completes.
  • TickerBehaviour
    – The `action()` and `done()` method are already implemented so that the `onTick()` (to be implemented by subclasses) method is executed periodically with a given period
    – The behaviour runs forever unless its `stop()` method is executed.
More about behaviours

- **The `onStart()` method of the Behaviour class** is invoked only once before the first execution of the `action()` method. Suited for operations that must occur at the beginning of the behaviour.

- **The `onEnd()` method of the Behaviour class** is invoked only once after the `done()` method returns `true`. Suited for operations that must occur at the end of the behaviour.

- Each behaviour has a pointer to the agent executing it: the protected member variable `myAgent`.

- **The `removeBehaviour()` method of the Agent class** can be used to remove a behaviour from the agent pool of behaviours. The `onEnd()` method is not called.

- When the pool of active behaviours of an agent is empty the agent enters the **IDLE state** and its thread goes to sleep.
The communication model

- Based on asynchronous message passing
- Message format defined by the ACL language (FIPA)

```
A1
<table>
<thead>
<tr>
<th>Prepare the message to A2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send the message</td>
</tr>
</tbody>
</table>

A2
<table>
<thead>
<tr>
<th>Get the message from the message queue and process it</th>
</tr>
</thead>
<tbody>
<tr>
<td>Post the message in A2’s message queue</td>
</tr>
</tbody>
</table>

Distributed JADE runtime
```
The ACLMessage class

- Messages exchanged by agents are instances of the `jade.lang.acl.ACLMessage` class.
- Provide accessor methods to get and set all the fields defined by the ACL language
  - `get/setPerformative();`
  - `get/setSender();`
  - `add/getAllReceiver();`
  - `get/setLanguage();`
  - `get/setOntology();`
  - `get/setContent();`
  - `....`
Sending and receiving messages

- Sending a message is as simple as creating an `ACLMessage` object and calling the `send()` method of the `Agent` class.

```java
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);
```

- Reading messages from the private message queue is accomplished through the `receive()` method of the `Agent` class.

```java
ACLMessage msg = receive();
if (msg != null) {
    // Process the message
}
```
A behaviour that processes incoming messages does not know exactly when a message will arrive → It should poll the message queue by continuously calling `myAgent.receive()`.

This of course would completely waste the CPU time.

The `block()` method of the `Behaviour` class removes a behaviour from the agent pool and puts it in a blocked state (it's not a blocking call!!).

Each time a message is received all blocked behaviours are inserted back in the agent pool and have a chance to read and process the message.

```java
public void action() {
    ACLMessage msg = myAgent.receive();
    if (msg != null) {
        // Process the message
    } else {
        block();
    }
}
```

This is the strongly recommended pattern to receive messages within a behaviour.
Selective reading from the message queue

- The `receive()` method returns the first message in the message queue and removes it.
- If there are two (or more) behaviours receiving messages, one may "steal" a message that the other one was interested in.
- To avoid this it is possible to read only messages with certain characteristics (e.g. whose sender is agent "Peter") specifying a `jade.lang.acl.MessageTemplate` parameter in the `receive()` method.

```java
MessageTemplate tpl = MessageTemplate.MatchOntology("Test-Ontology");

public void action() {
    ACLMessage msg = myAgent.receive(tpl);
    if (msg != null) {
        // Process the message
    } else {
        block();
    }
}
```
Receiving messages in blocking mode

- The `Agent` class also provides the `blockingReceive()` method that returns only when there is a message in the message queue.
- There are overloaded versions that accept a `MessageTemplate` (the method returns only when there is a message matching the template) and or a timeout (if it expires the method returns null).
- Since it is a blocking call it is “dangerous” to use `blockingReceive()` within a behaviour. In fact no other behaviour can run until `blockingReceive()` returns.

- Use `receive()` + `Behaviour.block()` to receive messages within behaviours.
- Use `blockingReceive()` to receive messages within the agent `setup()` and `takeDown()` methods.
The yellow pages service

A1: - serviceX
    - serviceY

A2: - serviceZ

A3: - serviceW
    - serviceK
    - serviceH

Yellow Pages service

DF

A4

A5

A6

A3

Publish provided services

Search for agents providing the required services

Exploit required service
The DF is an agent and as such it communicates using ACL.

The ontology and language that the DF “understands” are specified by FIPA. It is possible to search/register to a DF agent of a remote platform.

The `jade.domain.DFService` class provides static utility methods that facilitate the interactions with the DF:
- `register();`
- `modify();`
- `deregister();`
- `search();`

The JADE DF also supports a subscription mechanism.
DF Description format

- When an agent registers with the DF it must provide a description (implemented by the jade.domain.FIPAAgentManagement.DFAgentDescription class) basically composed of
  - The agent AID
  - A collection of service descriptions (implemented by the class ServiceDescription). This, on its turn, includes:
    - The service type (e.g. “Weather forecast”);
    - The service name (e.g. “Meteo-1”);
    - The languages, ontologies and interaction protocols that must be known to exploit the service
    - A collection of service-specific properties in the form key-value pair
- When an agent searches/subscribes to the DF it must specify another DFAgentDescription that is used as a template
JADE demo