# WADE USER GUIDE

## USAGE RESTRICTED ACCORDING TO LICENSE AGREEMENT.

Last update: 02-July-2010 - WADE 2.6

Authors: Giovanni Caire (Telecom Italia S.p.A)

WADE (Workflows and Agents Development framework) is a software platform that facilitates the development of distributed multi agent applications where agent tasks can be defined according to the workflow metaphor.
This document gives an overview of the WADE platform, presents its architecture and main functionalities, and outlines the conceptual model underlying it. Two major aspects of the platform are described: the execution model based on the workflow metaphor and the configuration and control features designed to hide the complexity of the distribution. WADE is based on JADE, a popular open source agent-oriented middleware and inherits from it a distributed topology with peer-to-peer networking and a software component architecture based on the agent paradigm.

# TABLE OF CONTENTS

# 1 Introduction

WADE (Workflows and Agents Development Environment) is a domain independent software platform built on top of ***JADE*** (http://jade.tilab.com), a popular open source middleware conceived to facilitate the development of distributed applications based on the ***agent-oriented paradigm***. Readers that are not familiar with JADE, are encouraged to look at the JADE White paper or (if they have programming skills) at the JADE Programming Tutorial for Beginners first.
As depicted in Figure 1, JADE provides a distributed runtime environment, the agent and behaviour (a task performed by an agent) abstractions, peer to peer communication between agents and basic agent lifecycle management and discovery mechanisms. WADE adds to JADE the support for the execution of tasks defined according to the ***workflow*** metaphor and a number of mechanisms that help ***managing the complexity of the distribution*** both in terms of administration and fault management.
Workflows and managed distribution are actually the two key elements of the platform and readers should primarily consider them when deciding if and how WADE can fulfil their needs.
In principle WADE supports "notepad-programming" in the sense that there is no hidden stuff that developers can't control. However, especially considering that one of the main advantages of the workflow approach is the possibility of representing processes in a friendly graphical form, WADE comes with a development environment called ***WOLF*** that facilitates the creation of WADE-based application. WOLF is an Eclipse (www.eclipse.org) plug-in and as a consequence allows WADE developers to exploit the full power of the Eclipse IDE plus additional WADE-specific features.
This document does not discuss WOLF. Interested readers should refer to the WADE Tutorial (http://jade.tilab.com/wade/doc/tutorial/WADE-Tutorial.pdf) for details about how to install and use it.

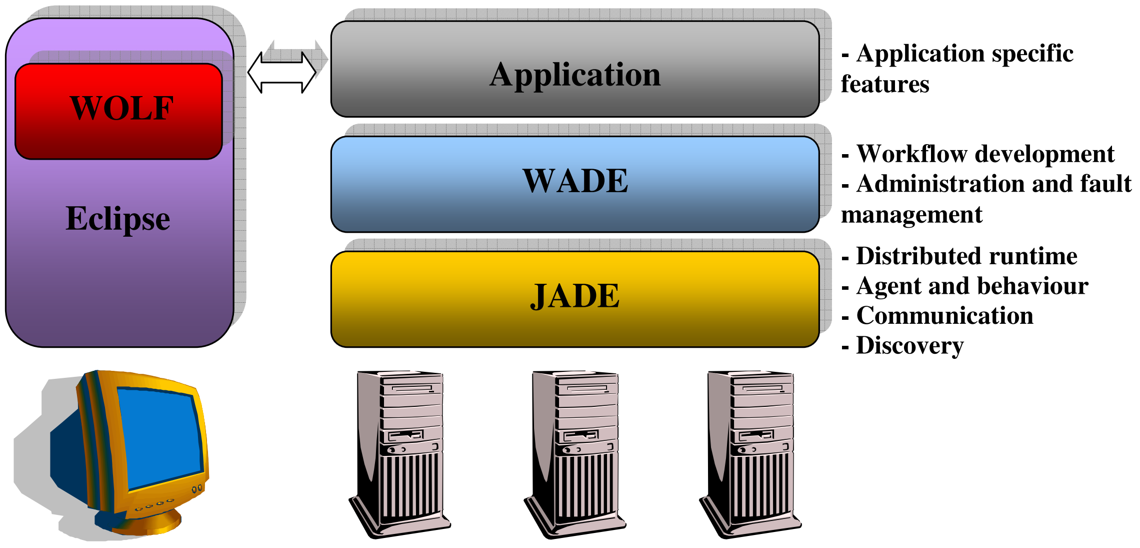


**Figure 1. The WADE platform**

## 1.1 Workflow based development

A workflow is a formal definition of a process in terms of activities to be executed, relations between them, criteria that specify the activation and termination and additional information such as the participants, the software tools to be invoked, required inputs and expected outputs and internal data manipulated during the execution.

### 1.1.1 Advantages

The key aspect of the workflow metaphor is the fact that the ***execution steps as well as their sequencing are made explicit***. This makes it possible to give a ***graphical representation*** of a process defined as a workflow. Such representation is clearly extremely more intuitive with respect to a piece of software code and in general is understandable by domain experts as well as by programmers. Domain experts can therefore validate system logics directly and not only on documents that most of the time are not perfectly up to date. In some cases they could even contribute to the actual development of the system without the need for any programming skill.
Another important characteristic is that, being the execution steps explicitly identified, the workflow engine (i.e. a system able to automatically execute a process defined as a workflow) can trace them. This makes it possible to create ***automatic mechanisms*** to facilitate system monitoring and problem investigation. Additionally, when processes have to be executed within the scope of a transaction, ***semi-automatic rollback procedures*** can be activated in case of unexpected fault.
Finally, since ***workflows are fully self-documented***, workflow-based development releases the development team of the burden of keeping documentation aligned each time design choices must be revisited to face implementation details or evolving requirements.

### 1.1.2 Scope

If on the one hand the workflow metaphor has a number of advantages as discussed in previous section, it is in general not suitable to deal with low level operations such as data management and transformation, mathematical computations and so on. Software code is typically more powerful and efficient to implement them. As a consequence, nowadays the workflow metaphor is mostly used in BPM environments where a workflow represents a business process and orchestrates a number of existing systems typically (but not necessarily) accessible by means of Web Services-based interfaces.
The main challenge in WADE is to ***bring the workflow approach from the business process level to the level of system internal logics***. That is, even if it can be used for that purpose too, WADE does not primarily target high level orchestration of services provided by different systems, but the implementation of the internal behaviour of each single system.
A direct consequence of the described approach is that WADE is expected to be particularly suitable for applications that imply the execution of ***possibly long and fairly complex tasks***. On the other hand, when dealing with systems that mainly retrieve/present/transform/store data from/to a data repository following user requests/inputs (as for instance all classical Web applications) usage of state of the art middleware technologies such as J2EE application servers is likely more indicated.

### 1.1.3 Approach

First of all it should be noticed that WADE does not include a single powerful workflow engine as the majority of BPM oriented tools. On the contrary WADE provides an extension of the basic `Agent` class of the JADE library called `WorkflowEngineAgent` that embeds a small and lightweight workflow engine (we talk about "***micro-workflow engine***"). As a consequence, besides normal JADE behaviours, all Workflow-Engine agents active in a WADE-based multi-agent applications are able to execute workflows represented according to a WADE specific formalism.
The second important point to highlight is that, in order to allow developers to exploit the workflow metaphor to define system internal logics and, at the same time, to give them the same power of a software programming language and a comparable execution efficiency, ***the WADE workflow representation formalism is based on the Java language***. That is, a workflow that can be executed by WADE Workflow-Engine agents is expressed as a Java class with a well defined structure (detailed in chapter 3). As such WADE workflows can be edited, refactored, debugged and in general managed as all Java classes and can include all pieces of code (methods, fields of whatever types, inner classes, references to external classes and so on) needed to implement the process

details. In addition, of course, the execution flow they specify can be presented and modified in a friendly, graphical way. More in details WOLF (the development environment for WADE based applications) is an Eclipse plugin and allows developers to work with a graphical view (suitable to manage the process flow) and a code view (the usual Eclipse Java editor suitable to define execution details) that are kept in synch.
Therefore the WADE workflow engine embedded into Workflow-Engine agents is not an interpreter of a workflow description language, but just executes compiled Java code. This on the one hand makes it extremely performant, but on the other hand requires the necessary workflow classes to be available when an agent is requested to execute a workflow. For this reason WADE uses ad hoc Java class loaders to allow deploying new/modified workflows that become immediately executable without the need to turn the system down.
Another important characteristic of WADE workflows is that they ***can be extended***. That is, it is possible to create new workflows by extending existing ones and just defining the differences. This feature will be detailed in section 3.2.5.
Finally it must be noticed that WADE does not impose that all system logics are defined as workflows. Developers are free to exploit the workflow metaphor to describe those tasks for which they think it is appropriate and use normal JADE behaviours (or other purely Java patterns) elsewhere. In certain cases one could even decide to create a WADE based application that does not use workflows at all.
Ho to define and execute workflows is described in details in chapters 3 and 4.

## 1.2 Managing the complexity of the distribution

WADE inherits from JADE a distributed runtime composted of several "***Containers***" that can run on different hosts and can contain each one a number of ***agents***. Even if this is not strictly mandatory, most of the time a container corresponds to a JVM. The set of active containers is called a ***Platform***. As depicted in Figure 2 a special container exists in the platform called "***Main Container***". The Main Container must be the first one to start and all other containers (typically called peripheral containers) register to it at bootstrap time. Furthermore the Main Container holds two special agents.
The ***AMS*** (Agent Management System) that represents the ***authority in the platform***, i.e. it is the only agent that can activate platform management actions such as creating/killing other agents, killing containers and shutting down the platform. Normal agents wishing to perform such actions must request them to the AMS.
The ***DF*** (Directory Facilitator) that implements the ***yellow pages service*** by means of which agents can advertise their services and find other agents offering services they need.
Further details on main JADE architectural components can be found in the JADE administration Tutorial (http://jade.tilab.com/doc/tutorials/JADEAdmin/index.html).

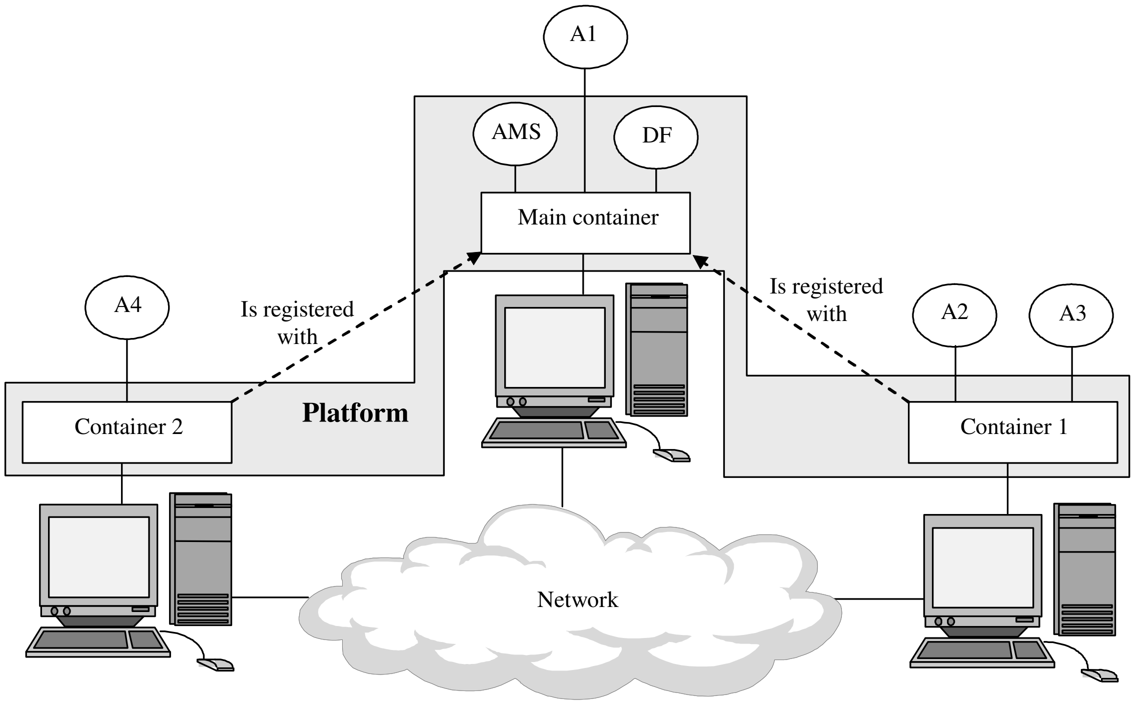


**Figure 2. JADE distributed architecture**

Distribution is often considered a useful characteristic, especially for applications that are expected to support heavy loads, since a distributed application can be deployed on highly scalable hardware architectures such as blades. It is clear however that administering a distributed application is more complex than administering a monolithic application unless proper tools are made available. Furthermore the probability that a host crashes increases proportionally with the number of hosts the application is distributed on and proper recovery mechanisms are necessary to ensure the application can survive. Clustering systems are often used for those purposes, but they are typically quite expensive and difficult to configure.
WADE faces these problems by providing a number of mechanisms that help the administrator in

- Configuring (tuning parameters) the application.
- Activating/deactivating the application spreading components (i.e. containers and agents) across available hosts according to specific needs.
- Monitoring runtime events and critical conditions such as disk and memory consumption.
- Deploying new/modified system logics at runtime without system down.
- Automatically recovering from host, container and agent faults.

The main mechanisms that WADE provides to hide the complexity of the distribution are described in chapters 2 and 5 that focus on the administration and fault tolerance aspects.

# 2 WADE-based applications

## 2.1 Architecture

This section presents the components that (together with the basic components provided by JADE and summarized in 1.2) make up the WADE architecture. As depicted in Figure 3, from the architectural point of view, a WADE-based application appears to be quite similar to a purely JADE based application. It runs on a set of hosts each one holding one or more JADE containers and, as usual, the Main Container (hosting the JADE AMS and DF) must be activated first with other containers registering to it at bootstrap time. Application specific agents are distributed across the containers according to some criteria that depend on the application requirements. WADE specific components are the ***Boot Daemon*** processes (one per host) each one responsible for activating containers in the local host, the ***Configuration Agent*** (CFA) always running in the Main Container and responsible for interacting with the boot daemons and controlling the application life cycle and the ***Controller Agents*** (one per container) responsible for supervising activities in the local container and for all the fault tolerance mechanisms provided by WADE.
Of course the last fundamental component of the WADE platform are the ***Workflow Engine Agents*** i.e. the agents able to execute tasks defined as workflows as will be described in section 3. Workflow Engine Agents can be used as they are and in certain cases all the logics of an application can be embedded in workflows that will be executed by “naked” Workflow Engine agents. In a more general case an application can extend the base `WorkflowEngineAgent` class in order to provide domain specific features. How many performer agents to use and where (in which containers) to deploy them depends on the application requirements. In general some of the application specific agents will be (possibly extended) performer agents and will therefore be able to execute workflows, while others will be “normal” agents running usual JADE behaviours. It should be noticed that a WADE-based application may even not use Workflow Engine agents at all and just exploit the administration and fault tolerance features. In that case we end up with a WADE-based application that does not use workflows.

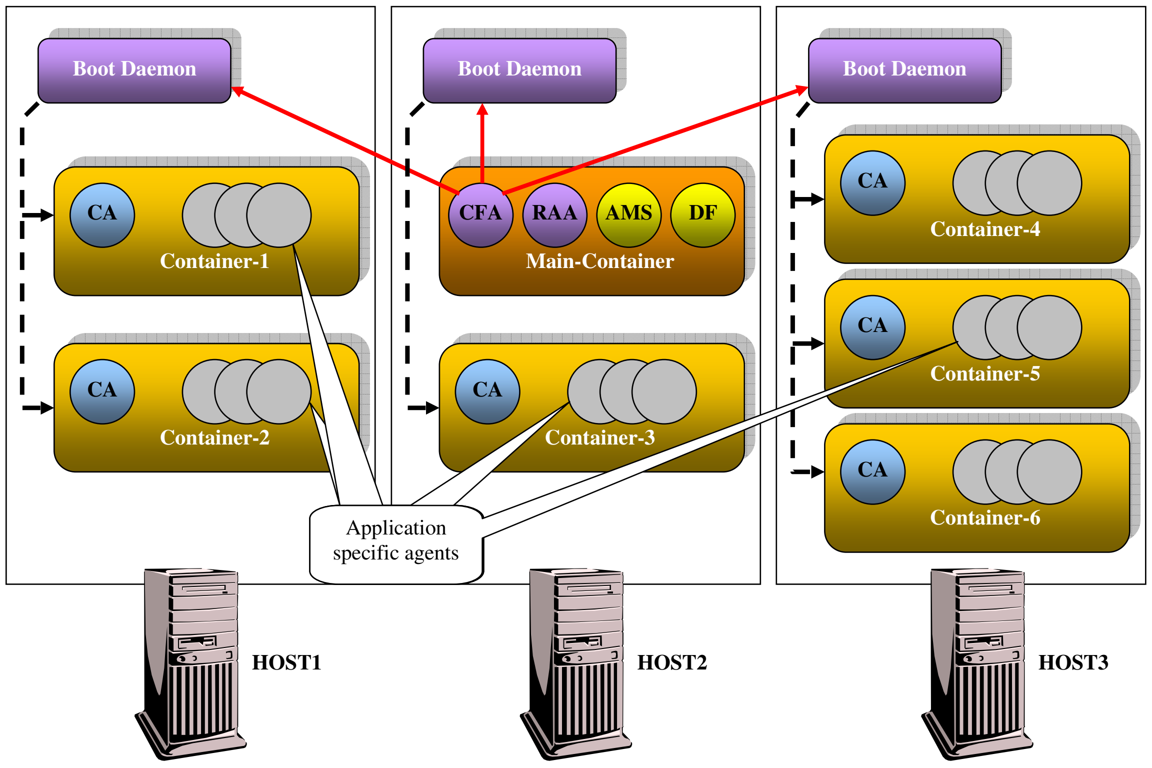


**Figure 3. The WADE Architecture**

Figure 3 also shows another WADE specific agent called ***Runtime Allocation Agent*** (RAA) active on the Main Container. The role of this agent will be discussed in chapter 5 when presenting Agent Pools and Fault Tolerance.
In many cases the application may involve other non-agent components running on the same or on separate hosts. How to manage these components and to make them interact with application agents is outside the scope of this document.

## 2.2 Application activation and termination

First of all it has to be highlighted that unlike the J2EE architecture, where a single application server can host several applications in parallel, each WADE-based application sits on top of one dedicated WADE platform.
When the application is down only the boot daemons (one per host) and the Main Container (with the JADE AMS, DF and the WADE CFA) are up and running. In this status it is possible to request the CFA to import an ***application configuration*** (`ImportConfiguration`[1] action of the Configuration Ontology). An application configuration is a file that specifies, according to an XML based format, which hosts are involved, which containers must be executed in each host and which agents must be activated in each container. The snippet below shows an example of an application configuration file involving a single host (the local host) with two containers (called Execution-Node-1 and Execution-Node-2) each one holding two Workflow Engine agents.

```
<?xml version="1.0" encoding="UTF-8"?>
<platform description="This is a sample configuration" name="Sample">
```

[1] Look at the Javadoc of the classes included in the `com.tilab.wade.cfa.ontology` for details about the parameters to be specified in the actions of the Configuration Ontology

```
  <containerProfiles>
    <containerProfile name="monitored" type="JADE">
      <properties>
        <property key="jade_core_AgentContainerImpl_enablemonitor"
                  value="true"/>
      </properties>
    </containerProfile>
  </containerProfiles>

  <hosts>
    <host name="localhost">
      <containers>
        <container name="Execution-Node-1" jadeProfile="monitored">
          <agents>
            <agent name="performer1" type="Performer Agent"/>
            <agent name="performer2" type="Performer Agent"/>
          </agents>
        </container>

        <container name="Execution-Node-2" jadeProfile="monitored">
          <agents>
            <agent name="performer3" type="Performer Agent"/>
            <agent name="performer4" type="Performer Agent"/>
          </agents>
        </container>
      </containers>
    </host>
  </hosts>

</platform>
```

Each application configuration has a name and an optional description. Application configurations available for import are kept in files called `<configuration-name>.xml` and stored by default in the `./cfg/configuration` directory.

For each agent, an application configuration specifies the name, the type (agent types will be described in section 2.3) and possibly some arguments as will be described in section 2.5.
An application configuration also supports the specification of container profiles. Each `ContainerProfile` includes either a set of JADE parameters (type="JADE") that tune the behaviour of the container itself or a set of JAVA parameters (type="JAVA") that tunes the behaviour of the JVM hosting the container.
For instance if we want to create a container in a JVM using 512MB of heap we could define in our application configuration file a container-profile of type `JAVA` as below

```
<containerProfile name="Heap512M" type="JAVA">
  <properties>
    <property key="Xmx512m"/>
  </properties>
</containerProfile>
```

In order to activate the application it is necessary to request the CFA to perform the `StartupPlatform` action of the Configuration Ontology. When serving this action the CFA reads the currently imported application configuration (also called the ***target configuration***) and:

- For each host requests the associated boot daemon to launch the specified containers (with the JADE and JVM configuration options defined by the JADE and JAVA container profiles if any) each one with a Controller Agent on top.
- For each container requests the associated Controller Agent to activate the specified agents.

The target configuration is kept internally by the CFA (by default it is stored in a file called `_target.xml`) and users are not expected to act on it directly.
Once the application is active, depending on the application needs, the containers and agents distribution may change (e.g. a new container can be launched with some agents on top to face an unexpected workload). Therefore the ***running configuration*** may differ from the target configuration. By requesting the CFA to perform the `SaveConfiguration` action of the Configuration Ontology, it is possible to take a snapshot of the running configuration and make it become the target configuration. In this way if the application is stopped and restarted its distribution of agents and containers will not change.
In order to tear down the application it is necessary to request the CFA to perform the `ShutdownPlatform` action of the Configuration Ontology. This has the effect of killing all containers, but the Main Container.
At any time it is possible to request the CFA to export the target configuration (`ExportConfiguration` action of the Configuration Ontology) to an application configuration XML file. Figure 4 summarizes the main actions of the Configuration Ontology and their effect on application configurations.

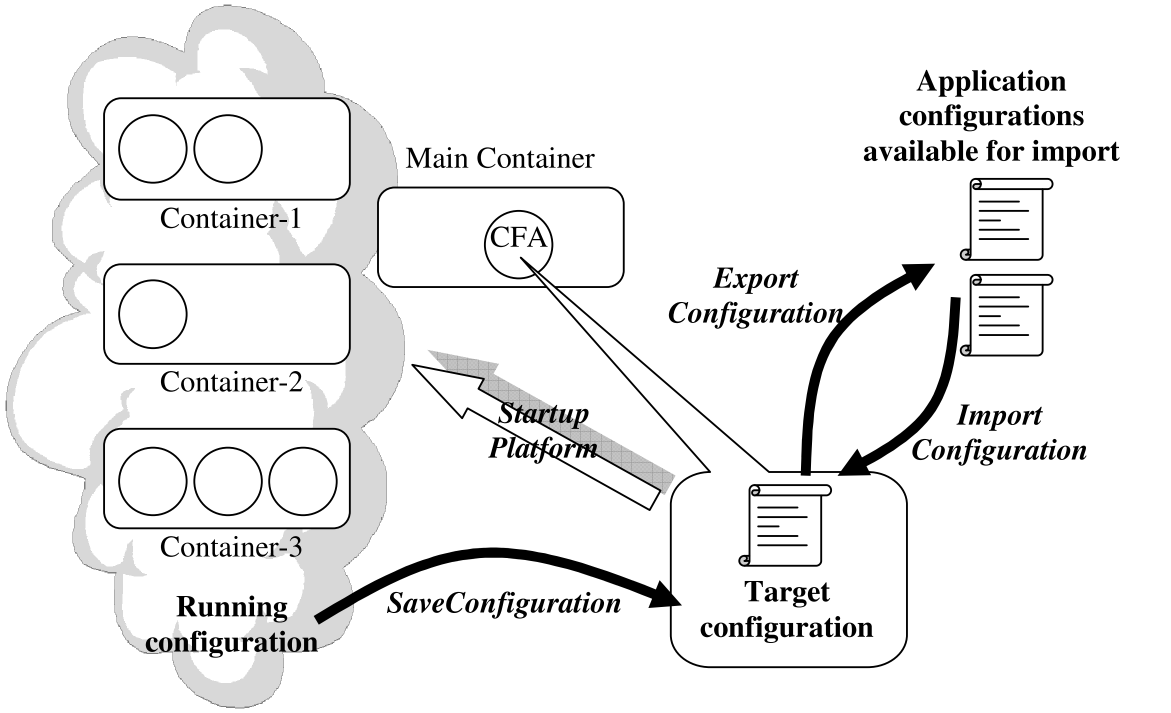


**Figure 4. Managing application configurations**

## 2.3 Getting started with the platform

Having described the architecture of WADE-based applications and the main management actions, in this section we describe how to launch the WADE platform and start playing with it.

### 2.3.1 WADE Distribution structure overview

WADE does not require any installation procedure a part from un-zipping the distribution zip file. Having done that somewhere on your disk, this is how the resulting directory tree should look like.

```
wade
   |---startMain.bat
   |---startBootDaemon.bat
   |---startMain.sh
   |---startBootDaemon.sh
   |---buildWade.xml
   |---...
   |---cfg/
          |---main.properties
          |---types.xml
          |---...
          |---configuration/
                        |---sample.xml
                        |---...
   |---lib/
          |---wade.jar
          |---jade.jar
          |---...
   |---log/
   |---projects/
```

The `.bat` and `.sh` files are the startup scripts (for Windows and Linux respectively) that can be used to start the Main Container and the `BootDaemon` process. All these scripts use ANT to properly manage the classpath. Therefore before activating them **it is necessary to have ANT (1.6.5 or later) installed and the ANT_HOME/bin directory in the PATH**.
The `buildWade.xml` file is the ANT build file that contains the targets referred by the startup scripts.

The `cfg` directory includes all configuration stuff. In particular

- the `main.properties` file is the JADE property file that includes all configurations related to the Main Container
- the `types.xml` file define agent types and roles as will be described in section 2.4
- the `configuration` subdirectory (as described in section 2.2) is used to store application configurations. A sample application configuration (actually that presented in 2.2) is provided in the `sample.xml` file.

The `lib` directory includes the `wade.jar` file and all libraries used by WADE including JADE.

The `log` directory is where log files produced by the Main Container and the local BootDaemon will be stored. In particular the BootDaemon log file contains all logs produced by the containers launched by the BootDaemon prefixed with the container name.

The `projects` directory is used to keep property files describing the location and structure of WADE-based applications that have to be launched using the WADE startup scripts as will be detailed in 2.6.

### 2.3.2 Starting up the platform

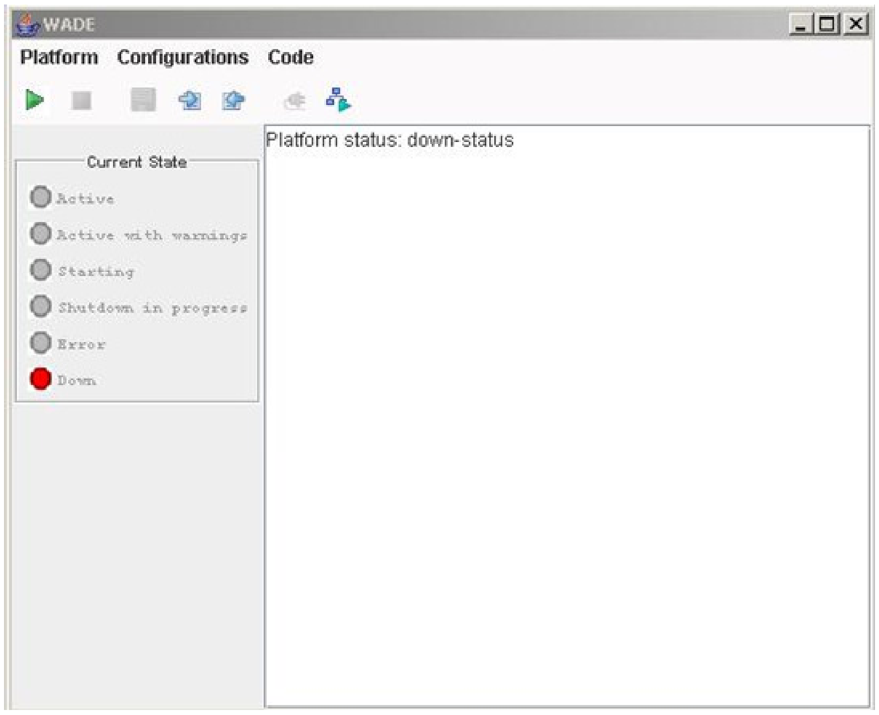
In order to start the WADE platform up on a single host it is sufficient to

- double-click on the `startMain`.bat file or execute the `startMain.sh` script depending on the underlying operating system. This activates the Main Container.
- double-click on the `startBootDaemon`.bat file or execute the `startBootDaemon.sh` script depending on the underlying operating system. This activates the BootDaemon on the local host.

To try WADE on more than one host, WADE must be installed on all hosts and the BootDaemon must be activated on each of them.

### 2.3.3 The Management Console

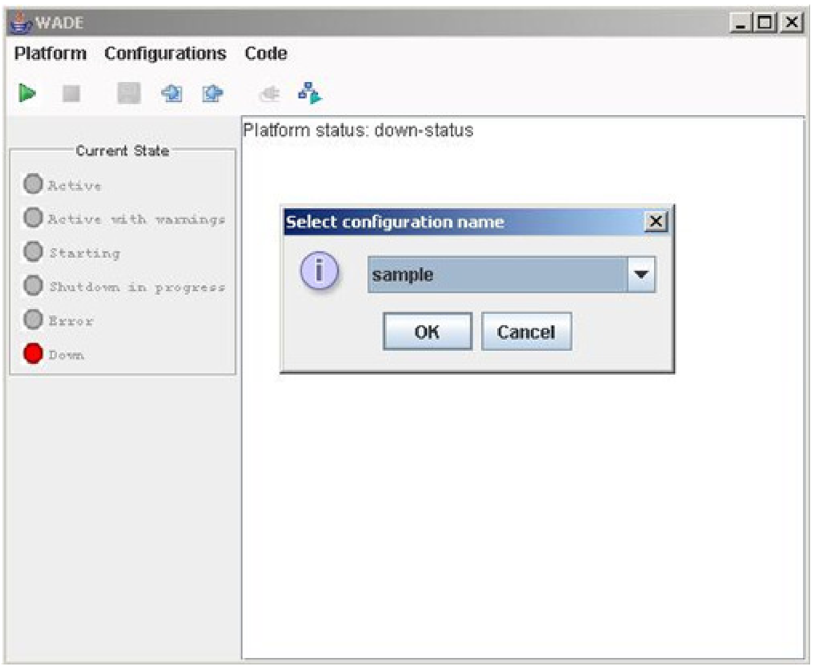
In general we expect real world applications to provide their own management console (possibly developed as a web application) properly customized according to the addressed domain and the application requirements. In order to start familiarizing with the platform however, WADE comes with a minimal management console that allows performing basic administration actions such as starting and stopping an application, importing and exporting configurations and deploying new/modified workflows without tearing the application down (this feature will be described in section 2.7).
The WADE Management Console is implemented as an agent (`com.tilab.wade.cfa.ManagementAgent`) and is activated by default on the Main Container (in order not to activate it, it is sufficient to remove the `ManagementAgent` from the list of agents to be activated at Main Container bootstrap in the `cfg/main.properties` file).
Having started the Main Container and the BootDaemon as described in 2.3.2 the WADE Management Console should appear showing that the status of the application is DOWN (see Figure 5).



Platform status: down-status

**Figure 5. The WADE Management Console**

By clicking on the `Import Configuration` button (or selecting the `Configurations` → `Import Configuration` menu item) it is possible to select and import the `sample` application configuration.

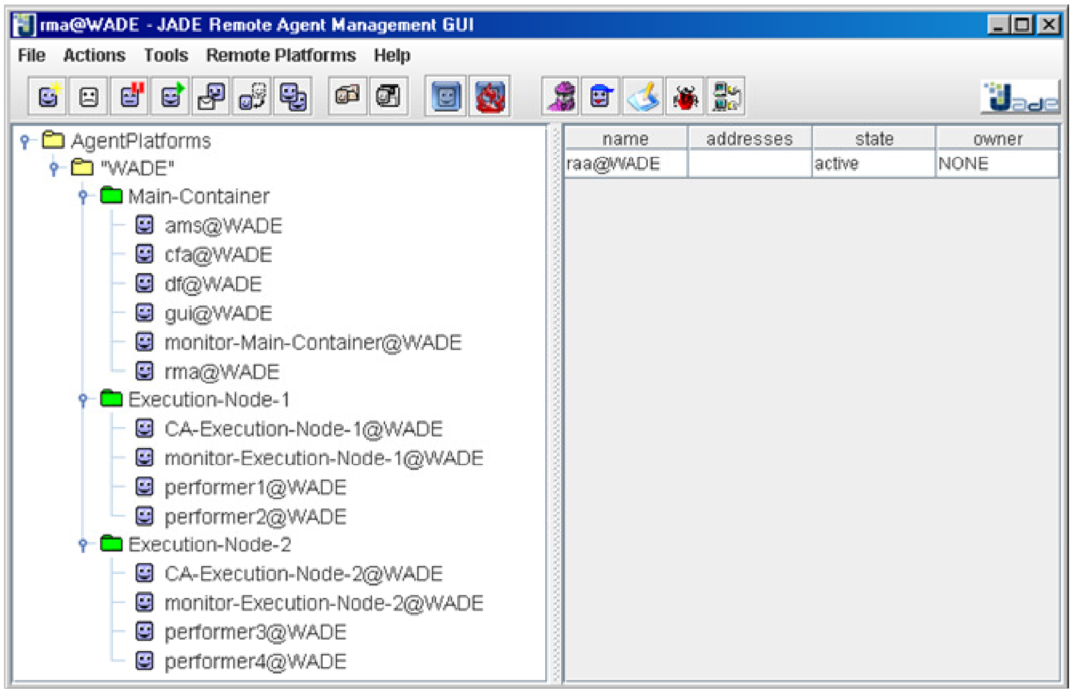


Platform status: down-status

sample

**Figure 6. Importing a configuration**

Having imported a configuration it is now possible to activate the application (note that at this point this is just a dummy application with no logics as we don't have any application specific code or workflow) by clicking on the `Startup` button (or selecting the `Platform` → `Startup` menu item). Figure 7 shows how the JADE RMA should look like once the application has started.



```
AgentPlatforms
  "WADE"
    Main-Container
      ams@WADE
      cfa@WADE
      df@WADE
      gui@WADE
      monitor-Main-Container@WADE
      rma@WADE
    Execution-Node-1
      CA-Execution-Node-1@WADE
      monitor-Execution-Node-1@WADE
      performer1@WADE
      performer2@WADE
    Execution-Node-2
      CA-Execution-Node-2@WADE
      monitor-Execution-Node-2@WADE
      performer3@WADE
      performer4@WADE
```

| name | addresses | state | owner |
|---|---|---|---|
| raa@WADE | | active | NONE |

**Figure 7. Running the sample configuration**

## 2.4 Agent types and roles

As mentioned a WADE-based application is composed of a set of agents that are instances of one or more agent types. As described in 2.1 WADE natively provides three main agent types (i.e. the ***Configuration Agent***, the ***Controller Agent*** and the ***Workflow Engine Agent***). An application often adds other domain specific agent types either extending the WADE base types (Primarily the Workflow Engine Agent) or directly extending their common base class `WadeAgentImpl.` In general we recommend WADE developers not to extends the `jade.core.Agent` class of the JADE library directly. As will be described in section 2.5, in facts, the `WadeAgentImpl` class provides a number of ready made mechanisms that are used to manage in a uniform way several aspects such as DF registration, agent attributes inspection, application termination and fault recovery.
In JADE an agent type just corresponds to a class that (directly or indirectly) extends `jade.core.Agent`. WADE explicitly introduces the notion of **agent type**. More in details an agent type has a name, a corresponding class and possibly a set of properties that will apply to all agents of that type.
The agent types involved in a WADE-based application are defined in a file called `types.xml` that must be placed in the classpath of both the Main Container and all boot daemon processes.
In order to show how the `types.xml` file looks like we provide below the default `types.xml` file defining the three WADE base types.

```
<platform>
  <agentRoles>
    <agentRole description="Administrator"/>
    <agentRole description="Workflow Executor"/>
  </agentRoles>

  <agentTypes>
    <agentType description="Workflow Engine Agent"
               className="com.tilab.wade.performer.WorkflowEngineAgent"
               role="Workflow Executor">
    </agentType>

    <agentType description="Control Agent"
               className="com.tilab.wade.ca.ControllerAgent"
               role="Administrator">
      <properties>
        <property name="autorestart" value="true"/>
        <property name="threadNumberThreshold" value="300"/>
        <property name="class-loader-root" value="./deploy"/>
      </properties>
    </agentType>

    <agentType description = "Configuration Agent"
               className="com.tilab.wade.cfa.ConfigurationAgent"
               role="Administrator">
      <properties>
        <property name="configurationsPath" value="./cfg/configuration"/>
      </properties>
    </agentType>
  </agentTypes>
</platform>
```

From the snippet above it can be noticed that the `types.xml` file also allows the definition of **agent roles** (the WADE default `types.xml` file defines 2 agent roles: ***Administrator*** and ***Workflow Executor***). An agent role can be seen as a macro-type and many agent types can refer to

the same agent role. As for agent types, also for an agent role it is possible to specify properties. These properties will be inherited by all agent types that refer to that agent role.
The agent type and agent role abstractions provide the following advantages:

- Application configuration files refer to agent types instead of agent class names that are typically less friendly to understand and remember.
- All configuration parameters that must be shared by all agents of a given type/role can be specified by means of agent type/role properties.
- WADE provides a number of utility methods (provided by the `DFUtils` class) to easily discover agents at runtime on the basis of their type/role.

### 2.4.1 Using environment variables

When specifying values for properties of agent types and roles it is possible to refer to environment variables using the syntax `${variable-name}`.
For instance the following snippet sets the directory where the CFA searches for application configurations to be imported/exported to the `applicationConfigurations` directory under the home directory of a project pointed to by the `project_home` environment variable

```
...
<agentType description = "Configuration Agent”
           className="com.tilab.wade.cfa.ConfigurationAgent"
           role="Administrator">
  <property name="configurationsPath"
            value="${project-home}/applicationConfigurations"/>
</agentType>
...
```

### 2.4.2 The TypeManager class

Agent types and roles (together with their properties) can be accessed by means of the `com.tilab.wade.commons.TypeManager` class. The `TypeManager` class makes use of the singleton pattern, that is there is a single `TypeManager` instance, accessible by means of the `getInstance()` static method of the `TypeManager` class. Once retrieved the singleton `TypeManager` it is possible to get the `AgentType` and `AgentRole` objects corresponding to a given agent type and agent role description and read associated configuration properties.
The code snippet below shows how to retrieve the value of property `configurationsPath` of the Configuration Agent type.

```
TypeManager theTypeManager = TypeManager.getInstance();
AgentType cfaType = theTypeManager.getType(“Configuration Agent”);
Map cfaProperties = theTypeManager.getProperties(cfaType);
String confPath = (String) cfaProperties.get(“configurationsPath”);
```

The `WadeAgentImpl` class described in next section further simplifies access to type-wide configurations by directly providing methods to access them as `String`, `Boolean`, `int` and so on managing the necessary format conversions.

## *2.5 WADE Agents*

In order for the management and fault tolerance mechanisms provided by WADE to work properly, agents in a WADE-based application must follow a set of rules regarding for instance how to register with the DF. These rules are automatically managed by the

`com.tilab.wade.commons.WadeAgentImpl` class. Therefore WADE developers are recommended to extend it rather than the `jade.core.Agent` class directly.
More in details the `WadeAgentImpl` class manages the following aspects.

**Arguments passed to agents at startup time**. As described in 2.2, agents that make up a WADE based application are specified in the application configuration XML file. That file also allows specifying startup arguments to be passed to each agent instance as exemplified in the snippet below.

```
...
<container name="Execution-Node-1" jadeProfile="monitored">
  <agents>
    <agent name="john" type="Book Buyer Agent">
      <parameters>
        <parameter key=”TARGET-BOOK”>
          <value>The Lord of the Rings</value>
        </parameter>
        <parameter key=”MAX-PRICE”>
          <value>50</value>
        </parameter>
      </parameters>
    </agent>
  </agents>
</container>
...
```

The `WadeAgentImpl` class makes available such arguments to subclasses in the form of a `protecetd Map` (called `arguments`). For instance, refering to the snippet above, the Book Buyer Agent can retrieve the target book to buy by just doing

```
String targetBook = (String) arguments.get(“TARGET-BOOK”);
```

Furthermore methods to access agent arguments as String, boolean, int and so on (managing all necessary format conversions) are directly provided as exemplified below.

```
int maxPrice = getIntArgument(“MAX_PRICE”, 30); // Default: 30
```

**DF registration**. A WADE agent extending `WadeAgentImpl` automatically registers with the DF providing a `DFAgentDescription` that allows the Control Agents to restart it in case of unexpected failure (either of the agent itself or of the container/host the agent lives in). Such `DFAgentDescription` includes, among others, the agent type, role, class name, location, host and all startup arguments received by the agent. Thanks to this registration it is possible to search agents on the basis of their type and role. The `com.tilab.wade.utils.DFUtils` class provides the `searchAnyByType(), searchAllByType(), searchAnyByRole()` and `searchAllByRole()` utility methods to do that very easily.

**Agent attributes runtime inspection and management**. The `WadeAgentImpl` class provides an embedded mechanism that allows exposing some agent attributes and make them readable and possibly modifiable at runtime with minimal effort. In order to make an attribute readable at runtime it is sufficient to have a getter method annotated by means of the @AttributeGetter annotation as exemplified below.

```
@AttributeGetter(name=”The name of the book I’m trying to buy”)
public String getTargetBook() {
  return targetBook;
}
```

In order to make an attribute not only readable, but also modifiable it is sufficient to add a setter method annotated by means of the @AttributeSetter annotation as below.

```
@AttributeSetter
public void setTargetBook(String tb) {
  targetBook = tb;
}
```

Runtime attribute inspection/modification can be done by requesting an agent extending `WadeAgentImpl` to perform the `GetAgentAttributes` and `SetAgentAttribute` actions of the `WadeManagementOntology`. The WOLF development environment provides a ready made GUI to inspect and modify agent attributes.

In order to support the described mechanisms, the `WadeAgentImpl` class already implements the `setup()` method of the `Agent` class. As a consequence agents extending `WadeAgentImpl` should redefine the `agentSpecificSetup()` method (rather than `setup()`) to perform their application specific initialization operations.

## 2.6 Running WADE-based applications

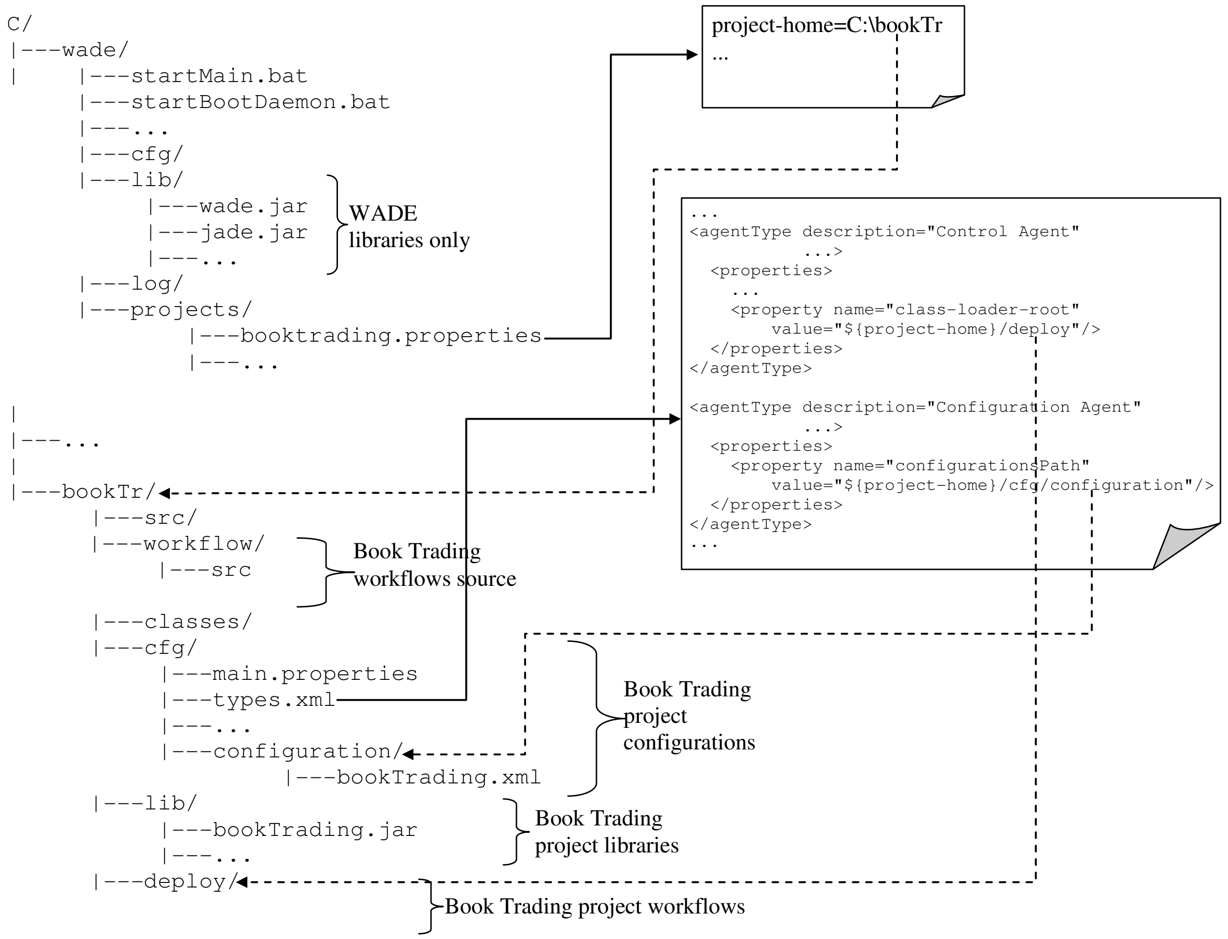
In section 2.3.2 we showed how to use the WADE startup scripts to activate the WADE platform. In general, however, a WADE-based application is not only composed of workflows that can be executed on a "naked" WADE platform, but also includes a number of classes (e.g. application specific agents) and possibly makes reference to some external libraries. In order for the WADE startup scripts to include application specific jar files into the classpath, it is of course possible to put them all (including external libraries) into the `lib` directory of the WADE installation. Similarly it is possible to modify the configuration files included in the `cfg` directory of the WADE installation to reflect application specific configurations. In general however we recommend to keep all application specific stuff separate from the WADE installation as described in the followings. In this way it is possible to deal with more than one WADE based application at the same time without mixing libraries and configuration files.
In order to do that it is sufficient to put a simple property file named `<application-name>.properties` into the `projects/` directory. This property file must specify the `project-home` property that must point to where the project (i.e. the application) is located on the disk. At this point the application Main Container and Boot Daemon can be launched by typing

```
startMain <application-name>
startBootDaemon <application-name>
```

respectively (similar command lines can be used on Linux).
Figure 8 provides an example showing how the project property file and the `types.xml` file of a Book Trading WADE based application can look like. With reference to that example the book trading Main Container and Boot Daemon can be launched by typing

```
startMain booktrading
startBootDaemon booktrading
```

respectively (similar command lines can be used on Linux).
The meaning of the `class-loader-root` property of the `Control Agent` type is described in more details in next section.



```
C/
|---wade/
|    |---startMain.bat
     |---startBootDaemon.bat
     |---...
     |---cfg/
     |---lib/
          |---wade.jar
          |---jade.jar
          |---...
     |---log/
     |---projects/
               |---booktrading.properties
               |---...

|
|---...
|
|---bookTr/
       |---src/
       |---workflow/
            |---src
       |---classes/
       |---cfg/
            |---main.properties
            |---types.xml
            |---...
            |---configuration/
                      |---bookTrading.xml
       |---lib/
            |---bookTrading.jar
            |---...
       |---deploy/
```

```
project-home=C:\bookTr
...
```

```
...
<agentType description="Control Agent"
           ...>
  <properties>
    ...
    <property name="class-loader-root"
        value="${project-home}/deploy"/>
  </properties>
</agentType>

<agentType description="Configuration Agent"
           ...>
  <properties>
    <property name="configurationsPath"
        value="${project-home}/cfg/configuration"/>
  </properties>
</agentType>
...
```

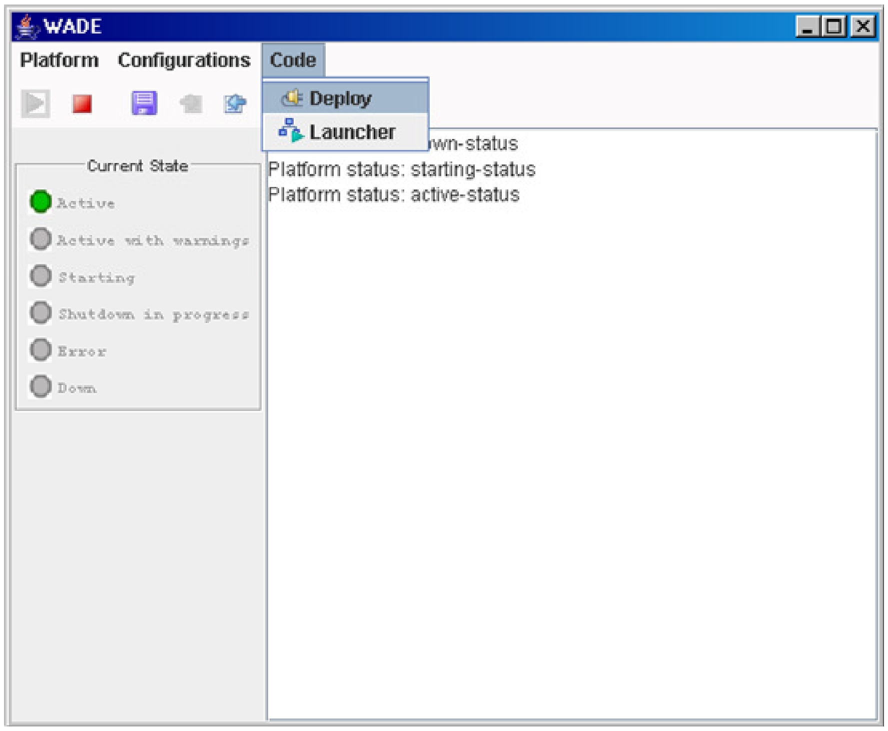
WADE libraries only
Book Trading workflows source
Book Trading project configurations
Book Trading project libraries
Book Trading project workflows

**Figure 8. Project property file**

## 2.7 Hot deployment of workflows

As mentioned in 1.1.3 WADE workflows are implemented as Java code. In order to support workflow hot deployment (i.e. the possibility of adding new workflows and modifying existing ones without the need to tearing the system down) an ad-hoc ClassLoader is used to load them. Moreover, since agents executing workflows are typically scattered across different containers, for each container there is an instance of this ClassLoader managed by the Control Agent controlling the container itself. More in details the property class-loader-root of the Control Agent type defines the directory (<project-home>/deploy by default) where jar files containing workflows must be placed

As shown in Figure 9, jar files containing new/modified workflows can be deployed at runtime by means of the WADE Management Console.



```
wn-status
Platform status: starting-status
Platform status: active-status
```

**Figure 9. Workflow hot deployment using the WADE Management Console**

**NOTE**: It should be noticed that, due to the Java ClassLoader delegation model, if the class implementing a workflow is included in the application classpath it is not loaded by the WADE Control Agent ClassLoader, but by the Java System ClassLoader. As a consequence it will not be possible to update it by means of the WADE workflow hot deployment mechanism.

# 3 Workflows

In this section we go in more details in the most relevant feature that WADE provides i.e. the support for defining system logics according to the workflow metaphor. More in details in this chapter we present how workflows are implemented in WADE, while in next chapter we will describe how to execute them. Instead this guide does not provide a section focusing on how to use Wolf to create workflows. Readers can refer to the WADE Tutorial available on line in the WADE site (http://jade.tilab.com/wade/doc/tutorial/WADE-Tutorial.pdf) for a step-by-step description about the Wolf Workflow Editor and the other relevant features that Wolf provides.

## 3.1 The workflow meta-model

As anticipated the approach adopted by WADE to support workflow development is to provide a workflow view over a Java class (with a proper structure) that can therefore be managed exploiting the full power of the Eclipse IDE. As a consequence WADE does not use internally any standard workflow definition language, but a proprietary formalism based on the Java language. However, in order not to re-invent the wheel and to facilitate import/export operations, WADE adopts a workflow meta-model quite similar to that defined in the **XPDL** standard specified by the Workflow Management Consortium (www.wfmc.org). The main elements that compose this meta-model are described hereafter.

A task that is being described is called a ***Process***. A process is composed of a set of ***Activities*** each one corresponding to the execution of given operations. A process defines a single ***Start Activity*** (specifying the execution entry point) and one or more ***End Activity*** (specifying the execution termination points). Each non-end activity has one or more outgoing ***Transitions***, possibly associated to a condition, leading to another activity in the process. Once the execution of the operations included in a given activity is terminated, the conditions of all outgoing transitions are evaluated. As soon as a condition holds the corresponding transition is fired and the execution flow goes on with the operations included in the destination activity.

A process can have one or more ***Formal Parameters*** defining the type of required inputs and expected outputs. At process invocation time proper values must be provided for input parameters and, at the end of the execution, the values produced as output parameters are returned to the requester.

Depending on the included operations, there are different types of activity. The most relevant ones are

- ***Tool activities***. The operations included in a tool activity consist in invoking one or more external tool generically identified as ***Applications***. Applications are computational entities defined outside the workflow process and wrapped by a uniform interface. Similarly to processes, applications have formal parameters defining required inputs and expected outputs. In general we expect applications to be provided to e.g. interact with an external system or appliance or to execute a ready made computation, but developers are free to use the Application abstraction as they like.
- ***Subflow activities***. The operations included in a subflow activity consist in the invocation of another workflow process. The execution of the subflow takes place in a separate computational space and (as will be described in section **Errore. L'origine riferimento non è stata trovata.**) can be even carried out by a different performer agent (possibly running in a remote host) with respect to that performing the main process.
- ***WebService activities***. The operations included in a Web Servicve activity consist in invoking a web service.

- ***Code activities***. The operations included in a code activity are specified directly by a piece of Java code embedded in the workflow process definition. It should be noticed that, unlike tool, subflow and route activities, code activities do not belong to the XPDL meta-model and are a proprietary WADE extension.
- ***SubflowJoin.*** By default subflows are executed synchronously, i.e. the main workflow blocks until the subflow completes and then goes on. Alternatively it is possible to execute a subflow asynchronously, that is the main workflow proceeds just after launching the subflow. The operations included in a Subflow Join activity consist in blocking the main workflow until a previously launched asynchronous subflow completes and getting the result.
- ***Route activities.*** These are empty activities (no operation is executed when the workflow visits such an activity) that can be used to simplify the implementation of complex flows.

Finally the process makes reference to a set of ***Data Fields*** each one having a name, a type and possibly an initial value. Data fields can be referenced wherever in the process e.g. in the conditions associated to the transitions, as actual parameters for application and subflow invocations and in the pieces of code triggered by code activities.

Figure 10 shows an example summarizing the main elements of the XPDL meta-model.

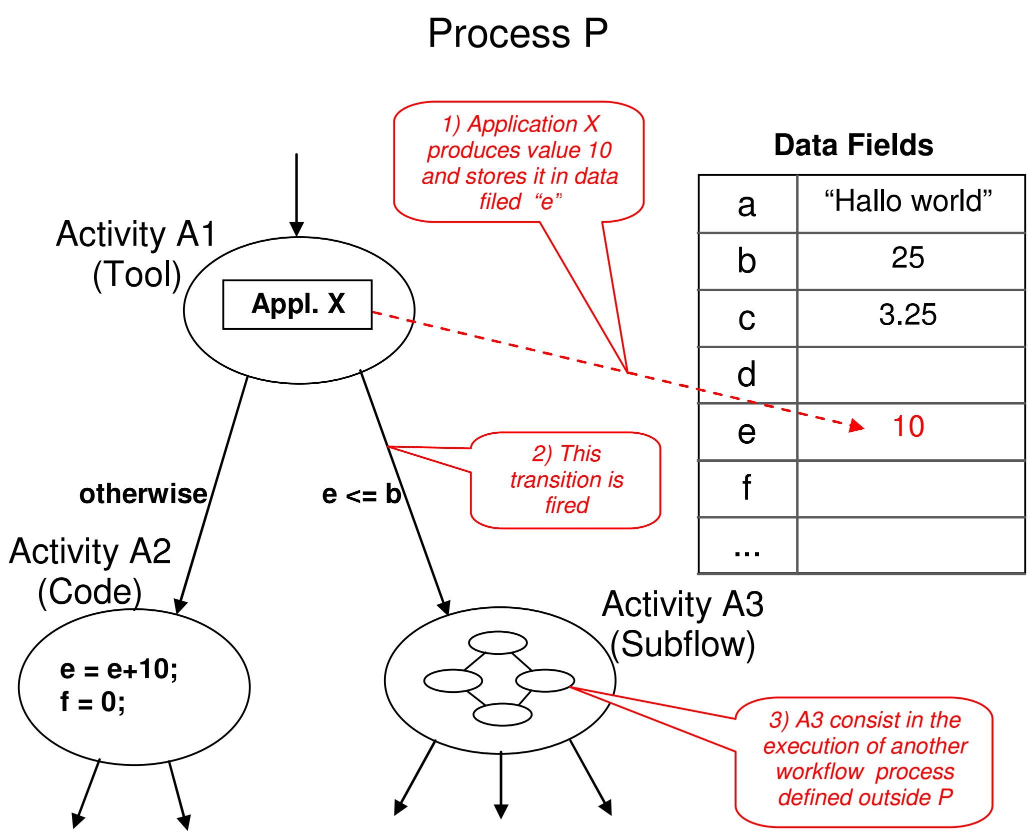

Process P
1) Application X produces value 10 and stores it in data filed "e"
Data Fields
a
"Hallo world"
b
25
c
3.25
d
e
10
f
...
Activity A1 (Tool)
Appl. X
otherwise
e <= b
2) This transition is fired
Activity A2 (Code)
e = e+10;
f = 0;
Activity A3 (Subflow)
3) A3 consist in the execution of another workflow process defined outside P

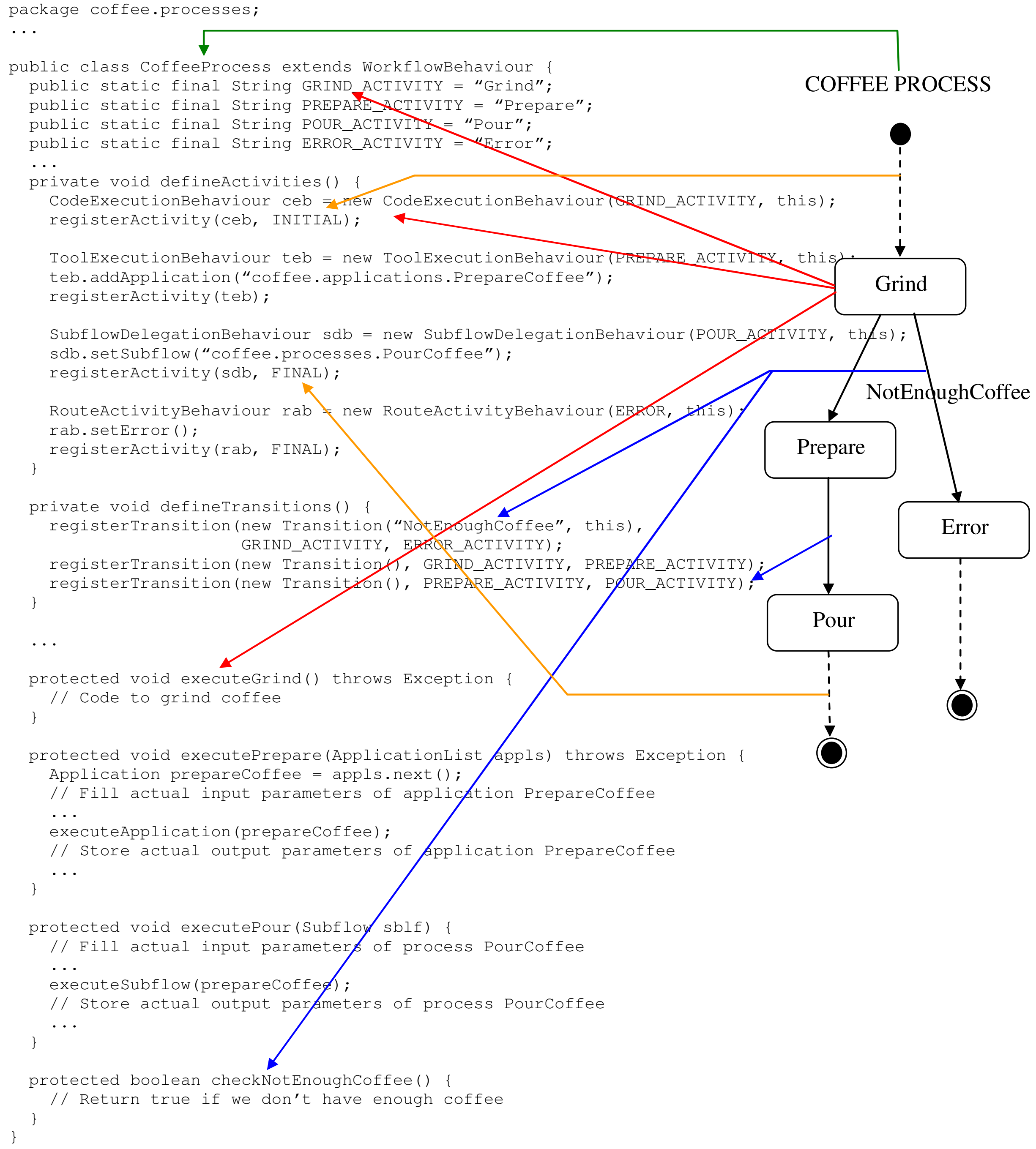
**Figure 10. Main elements in the XPDL meta-model**

## 3.2 Workflow elements implementation

Having described the main elements that make up a workflow process, in this section we show the structure of the Java code that actually implements them. It is important to note that, even if they could, we expect developers not to write the pieces of code that are described in this section directly, but to use the graphical workflow editor provided by WOLF to do that. However understanding the Java-based workflow definition formalism adopted by WADE may be useful to prevent unexpected behaviours and to investigate problems.

### 3.2.1 The WorkflowBehaviour class

Each workflow process is implemented by a Java class that extends (directly or indirectly) the `com.tilab.wade.performer.WorkflowBehaviour` base class. The `WorkflowBehaviour` class on its turn extends the JADE `FSMBehaviour` class and provides on top of it an API consistent with the meta-model presented in 3.1. In particular the `registerActivity()` and `registerTransition()` methods allow adding activities and transitions to the process. Even if in principle this is not strictly necessary it is highly recommended to register all process activities in a `private void` method called `defineActivities()` and all transitions in a `private void` method called `defineTransitions()`. This is because the workflow graphical editor included in Wolf searches for these methods to detect process activities and transitions to show.
The `registerActivity()` method gets the behaviour implementing the registered activity as an argument. More in details there are different classes corresponding to the different types of activity presented in 3.1. These are `ToolExecutionBehaviour` to be used to register a tool activity, `SubflowDelegationBehaviour` to be used to register a subflow activity, `WebServiceInvocationBehaviour` to be used to register a WebService activity, `CodeExecutionBehaviour` to be used to register a code activity and so on. The actual operations to be performed in a registered activity (no matter of its type) are specified in a `void` method of the workflow class. That method has a name constructed by adding the “execute” prefix to the name of the corresponding activity. Each activity behaviour is just responsible for invoking the related method when the activity is visited.
Similarly the `registerTransition()` method gets a `Transition` object as an argument. In case the transition has an associated condition, this is implemented by a `boolean` method of the workflow class. The Transition object is just responsible for invoking that method when the transition condition must be evaluated. the name of the method implementing a condition is constructed by adding the “check” prefix to the condition name.
Figure 11 summarizes the above correspondences.



```
package coffee.processes;
...

public class CoffeeProcess extends WorkflowBehaviour {
  public static final String GRIND_ACTIVITY = “Grind”;
  public static final String PREPARE_ACTIVITY = “Prepare”;
  public static final String POUR_ACTIVITY = “Pour”;
  public static final String ERROR_ACTIVITY = “Error”;
  ...
  private void defineActivities() {
    CodeExecutionBehaviour ceb = new CodeExecutionBehaviour(GRIND_ACTIVITY, this);
    registerActivity(ceb, INITIAL);

    ToolExecutionBehaviour teb = new ToolExecutionBehaviour(PREPARE_ACTIVITY, this);
    teb.addApplication(“coffee.applications.PrepareCoffee”);
    registerActivity(teb);

    SubflowDelegationBehaviour sdb = new SubflowDelegationBehaviour(POUR_ACTIVITY, this);
    sdb.setSubflow(“coffee.processes.PourCoffee”);
    registerActivity(sdb, FINAL);

    RouteActivityBehaviour rab = new RouteActivityBehaviour(ERROR, this);
    rab.setError();
    registerActivity(rab, FINAL);
  }

  private void defineTransitions() {
    registerTransition(new Transition(“NotEnoughCoffee”, this),
                       GRIND_ACTIVITY, ERROR_ACTIVITY);
    registerTransition(new Transition(), GRIND_ACTIVITY, PREPARE_ACTIVITY);
    registerTransition(new Transition(), PREPARE_ACTIVITY, POUR_ACTIVITY);
  }

  ...

  protected void executeGrind() throws Exception {
    // Code to grind coffee
  }

  protected void executePrepare(ApplicationList appls) throws Exception {
    Application prepareCoffee = appls.next();
    // Fill actual input parameters of application PrepareCoffee
    ...
    executeApplication(prepareCoffee);
    // Store actual output parameters of application PrepareCoffee
    ...
  }

  protected void executePour(Subflow sblf) {
    // Fill actual input parameters of process PourCoffee
    ...
    executeSubflow(prepareCoffee);
    // Store actual output parameters of process PourCoffee
    ...
  }

  protected boolean checkNotEnoughCoffee() {
    // Return true if we don’t have enough coffee
  }
}
```

**Figure 11. Workflow class structure**

### 3.2.2 Data Fields and Formal Parameters

Data Fields of a workflow process are implemented as fields of the workflow class. For instance, with reference to the coffee workflow shown in Figure 11, we could have a data field containing the amount of ground coffee in grams produced by the GRIND activity. This is implemented as an `int` field of the `CoffeProcess` class.

Workflow formal parameters need to be accessed by the workflow in a similar way to data fields. Therefore they are implemented, like data fields, as fields of the workflow class. In order to let WADE know that they are formal parameters, however, they must be annotated by means of the

FormalParameter annotation included in the com.tilab.wade.performer package. This annotation has several attributes indicating for instance the mode (i.e. INPUT or OUTPUT) or whether the parameter is mandatory (default) or optional. For instance the coffee process can take an input formal parameters nCups indicating the number of cups to be prepared. This would be implemented as an int field annotated as below

```
public class CoffeeProcess extends WorkflowBehaviour {
  ...
  // Formal parameters
  @FormalParameter(mode=FormalParameter.INPUT)
  private int nCups;
  ...
  // Data fields
  private int groundCoffeeGr;
  private int coffeeGrPerCup = 10;
  ...
  protected boolean checkNotEnoughCoffee() {
    return groundCoffeeGr < coffeeGrPerCup * nCups;
  }
}
```

### 3.2.3 BuildingBlock invocation

As mentioned, for each registered activity (but route activities) there is a corresponding method in the workflow class. While for code activities this method actually specifies the operations to be performed in the activity, for tool, subflow and WebService activities, the corresponding method just specifies the invocation of the indicated element. Such elements therefore can be seen as ***building blocks*** that are composed by the workflow to implement a complex process. The invocation of a building block (no matter if it is a subflow, a web service or an application) is composed of three steps:

1. fill actual input parameters.
2. invoke the indicated building block.
3. extract actual output parameters and store them in suitable workflow data fields for later processing

In the case of a tool activity, where it is possible to invoke more than one application, these steps are repeated for each application to be invoked. Therefore, unlike methods corresponding to code activities that take no arguments, the methods corresponding building block invocations take an argument representing the building block(s) to invoke.
With reference to the coffee example, let’s assume the PrepareCoffee application has the following formal parameters:
coffeeType (INPUT) of type String indicating the type of coffee to produce (e.g. “Espresso”, “American” ...)
cupsNumber (INPUT) of type int indicating the number of cups to prepare
coffeeJug (OUTPUT) of type CoffeeJug (a domain specific class) indicating the jug containing the prepared coffee.
Here is how the executePrepare() method corresponding to the PREPARE activity would look like.

```
public class CoffeeProcess extends WorkflowBehaviour {
  ...
  // Formal parameters
  @FormalParameter(mode=FormalParameter.INPUT, index=0)
  private int nCups;
  ...
  // Data fields
  private int groundCoffeeGr;
  private int coffeeGrPerCup = 10;
  private CoffeeJug jug;
  ...
  protected void executePrepare(ApplicationList appls) throws Exception {
    Application prepareCoffee = appls.next();
    // Fill input parameters
    prepareCoffee.fill("coffeeType", "Espresso");
    prepareCoffee.fill("cupsNumber", nCups);

    // This actually invokes the application execution
    executeApplication(prepareCoffee);

    // Store output parameters
    jug = (CoffeeJug) prepareCoffee.extract("coffeeJug");
  }
  ...
}
```

The `Application` class provides overloaded `fill()` methods to deal with all types of object without the need of casting. On the other hand when storing output parameters into a data field a proper cast to the data field type and possibly a primitive conversion (e.g. from `Integer` to `int`) must be done explicitly.
Handling actual parameters of a subflow or web service is done exactly the same way since the `Subflow` class and the `WebService` class provides identical `fill()` and `extract()` methods.

### 3.2.4 Layout information

All information related to the appearance of a workflow as shown in the graphical editor provided by Wolf (e.g. the fact that the box representing a given activity is placed in a given position within the screen, or that the arrow representing a given transition has some routing points) are specified by means of proper annotations associated to the workflow class. Clearly Wolf fully manages all these annotations and developers do not need to care or know anything about them.

### 3.2.5 Workflow inheritance

Considering the landscape of workflow management tools, one of the most characterizing features of WADE is certainly the possibility of creating new workflows by extending existing ones. In facts while the workflow approach is intrinsically procedural, inheritance is a typical characteristic of Object Orientation.
The simplest extension mechanism is based on Java class inheritance: being WADE workflows Java classes, it is possible to redefine the methods associated to the process activities thus modifying the related operations according to the needs of the extended workflow. Similarly it is possible to redefine the methods that implement transition conditions.
Beside that basic extension mechanism, however, WADE allows modifying the very flow of the parent process. In order to support that, the WorkflowBehaviour class provides the `deregisterActivity()` and `deregisterTransition()` methods that, in conjunction with the `registerActivity()` and `registerTransition()` methods described in section 3.2.1, allows modifying the process execution flow at will. For instance we could create an extended version of the `CoffeeProcess` presented in section 3.2.1 as shown in Figure 12. The Sweet Coffee Process adds the `AddSugar` activity and the related transitions. Moreover it removes the transition from the `Prepare` activity to the `Pour` activity. The `defineActivities()` and

`defineTransitions()` method of the `SweetCoffeeProcess` class reflect these modifications.

```
package coffee.processes;
...

public class SweetCoffeeProcess extends CoffeeProcess {
  public static final String ADD_SUGAR_ACTIVITY = “AddSugar”;
  ...
  private void defineActivities() {
    CodeExecutionBehaviour ceb = new CodeExecutionBehaviour(ADD_SUGAR_ACTIVITY, this);
    registerActivity(ceb, INITIAL);
  }

  private void defineTransitions() {
    deregisterTransition(PREPARE_ACTIVITY, POUR_ACTIVITY, false);
    registerTransition(new Transition(), PREPARE_ACTIVITY, ADD_SUGAR_ACTIVITY);
    registerTransition(new Transition(), ADD_SUGAR_ACTIVITY, POUR_ACTIVITY);
  }

  ...

  protected void executeAddSugar() throws Exception {
    // Code to add sugar
  }
  ...
}
```

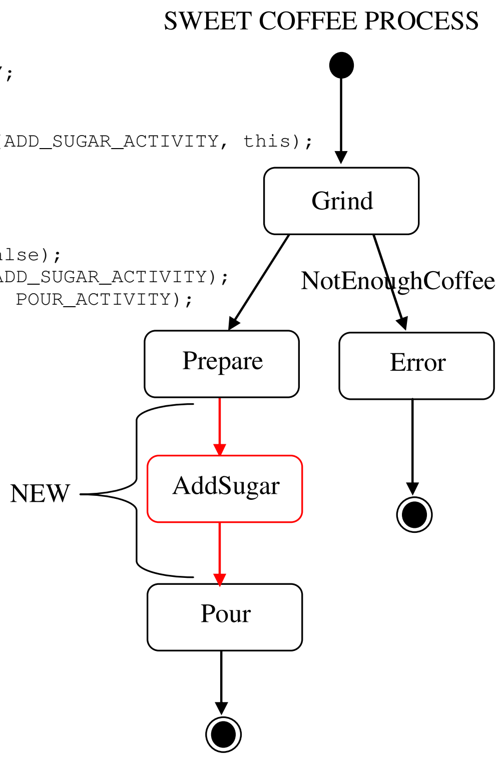


**Figure 12. Workflow extension**

The `changeActivityOrder()` method allows changing the order of an activity: for instance an activity that is `FINAL` in the parent workflow and must become `INTERMEDIATE` in an extended workflow that adds an additional step at the end.
Finally all layout information can be changed at will.

# 4 Executing workflows

As already mentioned WADE does not provide a single Workflow Engine. Rather it embeds “micro workflow engines” into agents giving them the ability to execute tasks defined according to the workflow metaphor. As a consequence, launching a workflow means requesting an agent to execute it. The interaction between the requester agent and the executor agent follwos a FIPA-request protocol as depicted in Figure 13.

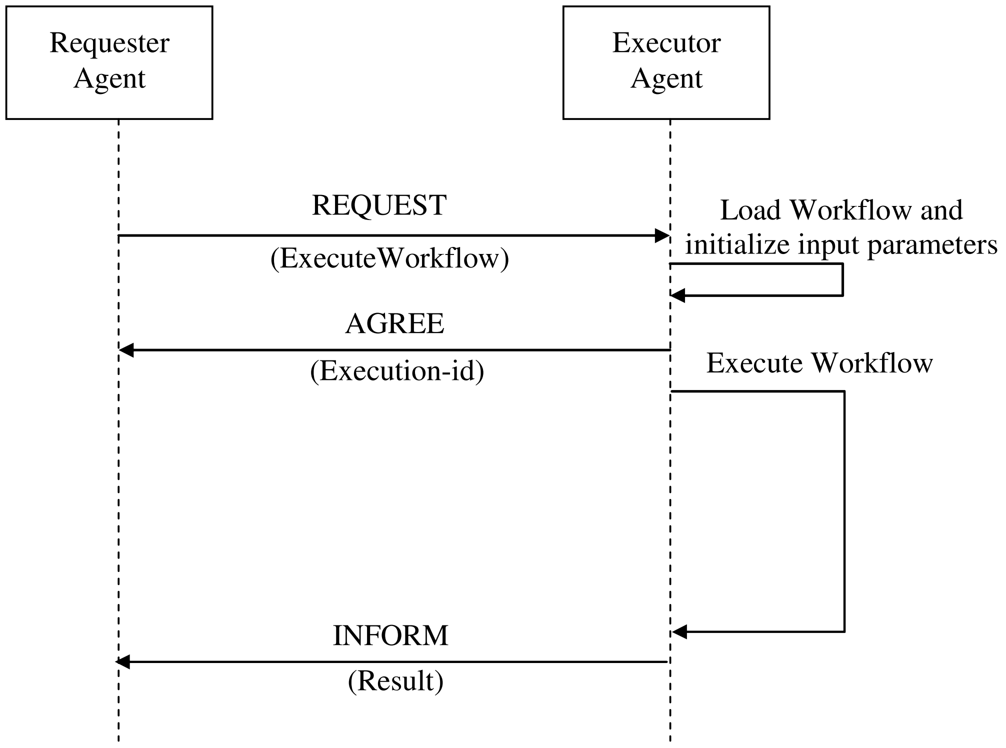


**Figure 13. Workflow execution request interaction**

The `ExecuteWorkflow` action, defined in the `WorkflowManagementOntology` (as all other workflow execution related concepts), mainly includes the `WorkflowDescriptor` specifying the workflow to execute, the actual parameters and optionally (but specifying it is a recommended practice) a session-id that must be unique and that will be propagated to any subflow triggered by the workflow that is been launched.
If the workflow class is is correctly loaded and instantiated, the executor agent generates an `execution-id` (unique within the executor agent) that can be used later on to act on the workflow (e.g. to kill it). This `execution-id` is returned to the requested within the `content` slot of an AGREE message. Note the difference between the `session-id` and the `execution-id`: the former is choosen by the requester, must be unique within the whole system and will be preserved in subflows; the latter is choosen by the executor, is unique only within the executor agent and changes across subflows (i.e. if a subflow is launched, this will have a different `execution-id`).
Once the workflow has been executed the executor agent sends back an INFORM holding the list of actual output parameters. In case of execution failure, instead of an INFORM message, a FAILURE message will be sent back holding the failure reason.

## 4.1 Launching workflows from code: the DispatchingCapabilities class

In order to request an agent to execute a workflow from your code it is of course possible to create the `ExecuteWorkflow` object and to implement a `FIPA-request` protocol with the intended executor as described in previous section. However WADE provides a ready-made class, called `DispatchingCapabilities`, that simplifies that. The code snippet below, taken (and simplified) from the WorkflowTracer example included among the WADE examples distribution, shows how to use it.

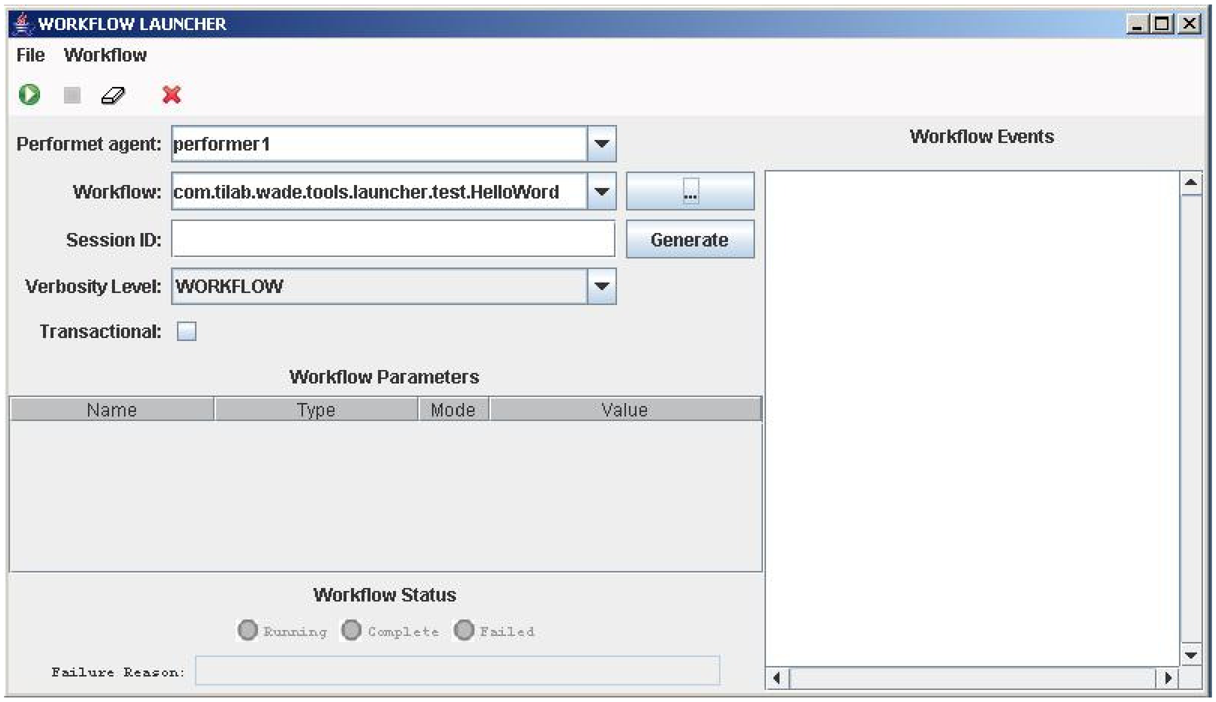
```
// Create a DispatchingCapabilities instance and link it to the local agent
DispatchingCapabilities dc  = new DispatchingCapabilities();
dc.init(this);

...

try {
  // Create the WorkflowDescriptor specifying the workflow to execute
  // and the   parameters
  WorkflowDescriptor wd = new WorkflowDescriptor("tracer.ThreeStepWorkflow");
  Map<String, Object> parameters = new HashMap<String, Object>();
  parameters.put("message", "Hello World!");
  wd.setParametersMap(parameters);

  // Launch the workflow
  dc.launchWorkflow(executor, wd, new WorkflowResultListener() {

    public void handleAssignedId(AID executor, String executionId) {
      // The executor correctly loaded the workflow and started the execution.
    }

    public void handleExecutionCompleted(List results,
                                         AID executor,
                                         String executionId) {
      // Workflow execution successfully completed
      System.out.println("Workflow completed successfully");
      // Get output parameters
      Map<String, Object> params = ElementDescriptor.paramListToMap(results);
      int length = (Integer) params.get("messageLength");
      System.out.println("Output parameter messageLength = "+length);
    }

    public void handleExecutionError(ExecutionError er,
                                     AID executor,
                                     String executionId) {
      // Workflow execution failed.
    }

    public void handleLoadError(String reason) {
      // Executor failed/refused to load the workflow
    }

    public void handleNotificationError(AID executor, String executionId) {
      // Generic communication error. This should never happen
    }

  }, null);
}
catch (Exception e) {
  e.printStackTrace();
}
```

From the code snippet above one can notice that:

- The fully qualified name of the workflow class must be indicated to specify the workflow to launch.
- Actual parameters can be filled in form of a `Map` and then set to the `WorkflowDescriptor`
- The `launchWorkflow()` method of the `DispatchingCapabilities` class can be used to trigger the execution of the indicated workflow on a given `WorkflowEngineAgent`.
- A `WorkflowResultListener` can be specified to be notified, among others, about the workflow execution completion.
- Workflow results (i.e. output parameters), if any, are provided as a `List` of `Parameter` objects. The `paramListToMap()` static method can be used to convert it into a `Map` thus simplifying the retrieval of output parameter values.

## 4.2 Using the LauncherAgent to launch workflows

In order to try the execution of workflows, WADE provides a simple utility agent called Workflow Launcher (`com.tilab.wade.tools.launcher.LauncherAgent` class) that allows launching workflows and tracing their execution. The Workflow Launcher Agent can be started exactly as all other agents (e.g. using the JADE RMA or specifying it among the agents to be activated at bootstrap time in the `main.properties` file - see section 2.3.1). Alternatively the WADE Management Console presented in section 2.3.3, allows starting the Workflow Launcher Agent by just clicking the `Workflow Launcher` button.



Performet agent: performer1
Workflow: com.tilab.wade.tools.launcher.test.HelloWord
Session ID:
Verbosity Level: WORKFLOW
Transactional:
Failure Reason:

**Figure 14. The Workflow Launcher Agent GUI**

The GUI of the Workflow Launcher Agent, depicted in Figure 14, allows specifying, among others, the workflow to launch, the agent that will be requested to execute it and the actual parameters (if any).

# 5 Advanced features

This section describes some advanced features that belong to the set of mechanisms provided by WADE to manage the complexity of the distribution, but were not presented in chapter 2, to make that chapter easier to read and understand.

## *5.1 Agent Pools*

In many cases, especially when an application has to serve a heavy load of requests, it is useful to have a set of agents of the same type to distribute the requests on. Since in many cases such sets can count tens, hundreds or even thousands of agents, specifying them all one by one in the application configuration file, as described in section 2.2, can be really annoying.
In order to deal with these situations in a friendly way, WADE supports the notion of Agent Pool. Agent Pools are defined in application configuration files as exemplified below

```
<platform name="Test-pool">

  <hosts>
    ...
  </hosts>

  <agentPools>
    <agentPool name="testPool" size="10" type="Workflow Engine Agent">
      <parameters>
        <parameter key="testParameter">
          <value>test value</value>
        </parameter>
      </parameters>
    </agentPool>
  </agentPools>

</platform>
```

The above snippet indicates that 10 agents of type Workflow Engine Agent will have to be launched. Such agents will be called `<agent-pool-name>-n` and each of them will receive an argument `testParameter` whose value is “`test value`”.
It should be noticed that the specification of an Agent Pool is very similar to the specification of an Agent. A part from the size attribute, however, Agent Pools are not associated to a specific host and container. WADE will take care of starting the agents of a pool in the available containers according to “agent allocation policies”. More in details, when executing the `StartupPlatform` action (see section 2.2) the Configuration Agent first creates all containers and agents explicitly indicated in the target configuration and then, for each agent pool (if any), delegates the creation of agents to the Runtime Allocator Agent (RAA). The latter, already briefly mentioned in section 2.1, manages a set of agent allocation policies and, whenever an agent must be created and there is no explicit indication about the container where to start it (as in the case of agents in a pool), it is requested to create that agent according to its policies.

### 5.1.1 Agent allocation policies

Agent allocation policies managed by the RAA are specified in an XML file called `raa.xml` and located in the `cfg/raa/` directory. The snippet below provides an example of how the `raa.xml` file looks like.

```
<allocationRules>
  <allocationRule>
    <!-- Match all agents of type Foo with argument arg1 set to Bar -->
    <condition>type==Foo;arguments.arg1==Bar</condition>
    <!-- Allocate them round-robin across all containers with Java-Profile P -->
    <policy name="com.tilab.wade.raa.policies.JavaProfileBasedRRPolicy">
      <properties>
        <property name="javaProfile" value="P"/>
      </properties>
  </allocationRule>

  <allocationRule>
    <!-- Match all agents -->
    <condition>name==*</condition>
    <!-- Allocate them round-robin across all containers -->
    <policy name="com.tilab.wade.raa.policies.RoundRobinPolicy"/>
  </allocationRule>
</allocationRules>
```

When the RAA is requested to create an agent somewhere in the platform, it scans the rules defined in the `raa.xml` file. Each rule is composed of a condition and an allocation policy. When processing a given rule the RAA matches the agent to be started against the condition. If the match is OK then the RAA uses the allocation policy to select a container where to start the agent. If the policy returns an existing container and the agent creation succeeds, the process stops. Otherwise the RAA moves to the next rule.
The snippet above for instance would result in starting all agents of type `Foo` with the argument `arg1` set to `Bar` in containers with ContainerProfile (of type JAVA) `P`. All other agents would be allocated round-robin across all available containers.
An allocation policy is a Java class implementing the `com.tilab.wade.raa.AgentAllocationPolicy` interface or extending the `com.tilab.wade.raa.policies.BaseAgentAllocationPolicy` base class. Developers interested in defining their allocation policy according to application specific requirements just need to provide new policy classes (see the javadoc of the `AgentAllocationPolicy` interface for details). However WADE already provides a set of ready-made policies (such as the `RoundRobinPolicy` and the `JavaProfileBasedRRPolicy` mentioned in the snippet above) that can be used to cover many realistic situations. Details on how such policies work and how to configure them can be found in the related javadoc.

## *5.2 Agent and containers fault tolerance*

Fault tolerance is a very important issue especially when dealing with real-world application. Using a distributed approach (as typically happens when building an application on top of JADE/WADE) provides a mean to deal with unexpected HW faults.
JADE already includes suitable fault tolerance mechanisms (such as the Main Container Replication Service described in chapter 9 of the book "Developing Multi Agent Systems with JADE") that allows the platform to survive to a fault of a container or host. Such mechanisms however work at the platform level and not at the application level. That is they ensure that the mechanisms provided

by the platform, such as message delivery and agent creation/destruction, continue to work even after an unexpected fault of one of the hosts where a JADE-based application is running. Of course if an agent implementing a given application specific piece of functionality F was running on the crashed host, the application (even if its remaining agents will still be able to exchange messages and exploit other platform services) may not work anymore as functionality F is no longer available.

In order to support fault tolerance at the application level, WADE provides an additional mechanism that enable automatically restarting all agents that suddenly disappear due to HW or SW faults. This mechanism is implemented by the Control Agents. As described in section 2.1, each container (but the Main Container) holds a Control Agent that is responsible for supervising the resources in the local container. Furthermore Control Agents coordinate themselves so that a single leader is elected.
The "*autorestart*" mechanism works as follows.

- If an agent suddenly dies (this may happen if there is a software bug in the agent's code that causes an uncaught exception), the Control Agent of the local container automatically restarts it.
- If an entire container suddenly dies (e.g. because the container process is killed), of course the local Control Agent dies too. The leader Control Agent then takes care of restarting the whole container (this operation is actually performed by the BootDaemon on the host where the container was active upon a request from the leader Control Agent) with all its agents.
- If the dead container included the leader Control Agent, other Control Agents elect a new leader that takes care of restarting the dead container.
- If the recreation of the whole container fails (this is always the case when a container disappeared due to a fault of the underlying HW) the leader Control Agent restarts all agents that where living in the dead container through the Runtime Allocator Agent described in section 5.1. The latter will then recreate all dead agents according to its agent allocation policies.

It should be noticed that the autorestart mechanism provided by WADE is only responsible for restarting the dead agents passing them the same arguments they were launched with. Application developers are responsible for implementing application specific mechanisms to restore the internal state of the agents after a fault/restart. In order to facilitate that, the `WadeAgentImpl` class (that as mentioned in section 2.5 should be extended by all application agents) provides the `getRestarted()` method that can be used in the `agentSpecificSetup()` method to distinguish between a normal startup and a restart after a crash..