Manual

JIAC – Java-Based Intelligent Agent Componentware

Version 5.2.4

JIAC Development Team

2018 / 06

DAI-Labor
Technische Universität Berlin

Contents

1 Introduction

1.1 What this manual is about

This manual is a pragmatical guide to agent development with the *Java-Based Intelligent Agent Componentware*, or *JIAC*. Herein, we describe how to start with JIAC, describe the basic concepts of JIAC and show some easy customizations and additions, which allow to utilize JIAC in a wide range of application domains.

Future versions of this manual will describe more advanced features of JIAC, such as its scripting language *JADL* [1], agent migration, agent management, and tools.

1.2 Introduction to JIAC

JIAC is a framework for developing *multi-agent systems* (*MAS*) and services [5]. The motivation for JIAC was:

- to ease the development of complex, distributed applications,
- to support system development in heterogeneous environments, and
- to deepen the knowledge in management of multi-agent systems.

Additional constraints lead the development of JIAC V:

- Always use standards when available.
- Do not reinvent the wheel.
- Relate to real-world software development whenever possible.

The first version (JIAC V) has been used in the Multi-Agent Contest 2008[1], where we have learned a lot about making things easy for programmers (i.e., ourselves).

The current version can be though of as a middle-ware, built around the agent metaphor, to develop distributed applications, to integrate heterogeneous environments and to integrate into different environments, and it is manageable by humans, to a certain extend.

The current version of JIAC incorporates the following features [6]

[1] http://www.multiagentcontest.org/

- Spring-based component system
- ActiveMQ-based messaging
- JMX-based management
- Transparent distribution
- Service-based interaction
- Semantic service search and selection
- Support for flexible and dynamic reconfiguration in distributed environments (component exchange, agent cloning, strong agent migration, fault tolerance)

1.2.1 Why JIAC?

We have noticed in many projects that people always think of agent development as something very new and totally different. It is not! Just the opposite! When you know a programming language like Java it becomes even easier to develop applications and services using an agent framework like JIAC. You can think of people, what they are capable of, what they are talking with each other and how they work together, and implement that! And you can use all tools, libraries and methodologies you are used to.

1.3 Acquiring JIAC

We recommend using the *Apache Maven build manager* for the management of JIAC projects. This way, you only have to specify the dependency to JIAC in your project description, and JIAC, together with all of its dependencies, will automatically be acquired from the respective repositories.

1.3.1 Java

First of all, you will need a Java installation, version 1.8 or higher. You can download Java from `http://www.java.com/`. If you are not familiar with Java, you will find sufficient information on the Web.

1.3.2 Apache Maven

You can download the Apache Maven build manager from `http://maven.apache.org/`.

After installing maven, you can *compile*, *test*, *build*, *install*, and *deploy* your projects very easily from the command line, using the command `mvn <goal>`, with `<goal>` being one of `compile`, `test`, `package`, or `install`, respectively. Therefore, each project needs a `pom.xml` (Project Object Model) in the project's root directory, holding information such as the project name, dependencies, and repositories from where to get these dependencies.

Maven will automatically download the dependencies (recursively) and store downloaded libraries in the `.m2/` directory, which is located in your home folder. Some of the information from the `pom.xml` common to all of your projects, such as the repositories to use, can be put in another file, `settings.xml`, located in this directory. We will not go into too much detail regarding Apache Maven in this manual. Please consult the respective documentation for more info.

If you know what you are doing, you can also use a different build management system, such as Gradle. However, we will only provide support for Maven.

1.3.3 JIAC All-in-One JAR

Alternatively to using Maven for resolving the dependencies to JIAC, you can also download and use a JIAC All-in-One JAR.[2] This Jar includes the core components of JIAC together with all their dependencies, such as Spring, ActiveMQ, and others. This makes it particularly easy to get started with JIAC, in particular if you are not familiar with Maven – just add the JAR to the class path of your project. Still, we recommend using Maven at least when combining JIAC with other dependencies, as otherwise you might end up with multiple versions of the same dependency on the class path.

1.3.4 Eclipse

We recommend using Eclipse[3] for developing JIAC applications. Besides generally being a powerful Java IDE, the JIAC Toolipse, a collection of Tools for JIAC developers, is based on Eclipse, as well. You can, however, also use any other Java IDE for developing JIAC applications.

Note: When using Apache Maven together with the Eclipse Development Environment, we *strongly* recommend using the Eclipse plugin for Maven,[4] and not the Maven plugin for Eclipse. The Eclipse plugin for Maven allows you to run `mvn eclipse:eclipse` in your project directory whereupon Maven will automatically generate Eclipse's `.project` and `.classpath` files from Maven's `pom.xml`. These are required to import Maven projects into your Eclipse workspace.

1.3.5 Example Project

Usually, a Maven project is structured as follows:

- source directories `src/main/java` and `src/main/resources` holding the main Java files and resources (e.g. icons, configuration files, ...) needed for the project
- test directories `src/test/java` and `src/test/resources` holding Java source files and resources needed for (unit-) testing the project

[2] http://jiac.de/Downloads/jiac/agentCore-5.2.4-jar-with-dependencies.jar
[3] http://www.eclipse.org
[4] http://maven.apache.org/eclipse-plugin.html

- the `pom.xml`

Listing 1.1 shows a simple `pom.xml` that can be used for your first JIAC projects.

Listing 1.1: Simple pom.xml for JIAC projects

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org
       /2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM
       /4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
2
3    <modelVersion>4.0.0</modelVersion>
4
5    <groupId>de.dailab.jiactng.examples.manual</groupId>
6    <artifactId>helloworld</artifactId>
7    <version>1.0.0-SNAPSHOT</version>
8    <packaging>jar</packaging>
9    <name>JIAC V Hello World Example</name>
10
11   <dependencies>
12     <dependency>
13       <groupId>de.dailab.jiactng</groupId>
14       <artifactId>agentCore</artifactId>
15       <version>5.2.4</version>
16     </dependency>
17     <!-- more dependencies -->
18   </dependencies>
19
20   <repositories>
21     <repository>
22       <id>dai-open</id>
23       <name>DAI Open Repository</name>
24       <url>http://repositories.dai-labor.de/extern/content/repositories/dai-open/<
             /url>
25       <snapshots>
26         <enabled>false</enabled>
27       </snapshots>
28     </repository>
29     <!-- more repositories -->
30   </repositories>
31
32   <build>
33     <plugins>
34       <plugin>
35         <groupId>org.apache.maven.plugins</groupId>
36         <artifactId>maven-compiler-plugin</artifactId>
37         <configuration>
38           <source>1.8</source>
39           <target>1.8</target>
40         </configuration>
41       </plugin>
42     </plugins>
43   </build>
44
45 </project>
```

After some header information, the JIAC module “agentCore” is listed as the only dependency (see line 11–18). This module provides the most important features of the JIAC agent framework, which will be topic of the following section of this manual.

JIAC is deployed on the DAI Open Repository, which is included as a repository in the `pom.xml` (see line 20–30):

http://repositories.dai-labor.de/extern/content/repositories/dai-open/

The `build` block (see line 32–43) specifies which Maven modules can be used for this project, and how they are configured. In this example, only basic compiling functionality is included; other plugins can be used for automatically creating an assembly for the project, too. Again, refer to the Maven documentation for more information.

Now we can run the command `mvn package` from inside the project directory. Maven will start downloading JIAC and its dependencies into the local repository (located in your `/.m2/` directory) and will eventually report that the build process was a success.

Of course, the project is still empty. In the next chapter, we will start filling it with content.

2 JIAC Programming Basics

This chapter explains the basic concepts that are used while implementing an application with JIAC (see also Figure 2.1). A typical *JIAC Application* consists of *AgentNodes*, *Agents*, and *AgentBeans*.

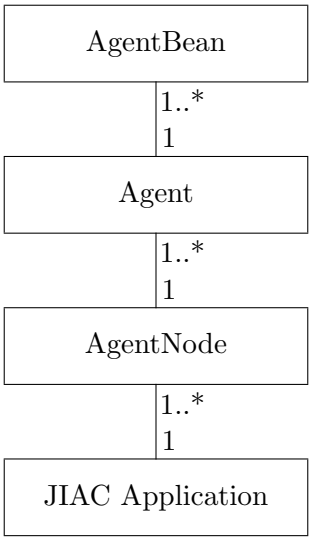


Figure 2.1: JIAC basic concepts and their structural relationships.

An *AgentNode* is a Java VM providing the runtime infrastructure for agents, such as discovery services, white and yellow pages services, communication infrastructure. A JIAC application consists of one or more AgentNodes. Normally, there is one AgentNode per physical machine. The AgentNode comes ready-to-run, but can be adapted to the needs of the target environment and can also be extended by additional components, so-called AgentNodeBeans.

Each AgentNode may run several *Agents*. Agents provide services to other agents and comprise life-cycle, execution cycle and a memory. An agent can use infrastructure services in order to find other agents, to communicate to them and to use their services. Skills and abilities of the agent can be extended by so-called AgentBeans.

AgentBeans is the mean to implement the functionality. They are plugged into agents and provide services (so-called Actions) to other agents. AgentBeans have a life-cycle.

In the following, we will illustrate these basic concepts in two examples: *HelloWorld*, the basic agent saying "Hello World", and *PingPong*, two agents calling out "Ping" and "Pong" to each other.

2.1 A First Example

We start with a simple example, with one agent that says *Hello World*.

2.1.1 Hello World

A Java class, the so-called AgentBean, defines not the entire agent, but just one aspect of it: a behavior or a set of capabilities, depending on what the Bean does. In our case, the Bean prints "Hello World" on the terminal when it is executed (Listing 2.1, to be put under *src/main/java/examples/helloworld/HelloWorldBean.java*):

Listing 2.1: Hello World Agent Bean

```
1  package examples.helloworld;
2  import de.dailab.jiactng.agentcore.AbstractAgentBean;
3
4  public class HelloWorldBean extends AbstractAgentBean {
5
6    public void execute() {
7      System.out.println("Hello World!");
8    }
9
10 }
```

When *is* it executed? The Bean's `execute()` method is meant to process one time or periodic tasks. The Bean's `executionInterval` is set to `1000` in the configuration file, meaning that this is done once a second. Of course, there are more things you can do with Beans, which we will come back to later.

The configuration file defines the setup of the JIAC agents and agent nodes, using one or more Spring XML files. Here, we have a `HelloWorldBean`, given to an `HelloWorldAgent`, which sits on a `HelloWorldNode` (Listing 2.2, to be put under *src/main/resources/hello_world.xml*):

Listing 2.2: Hello World Agent Configuration

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    xsi:schemaLocation="http://www.springframework.org/schema/beans
5    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
6
7    <import resource="classpath:de/dailab/jiactng/agentcore/conf/AgentNode.xml" />
8    <import resource="classpath:de/dailab/jiactng/agentcore/conf/Agent.xml" />
9
10   <bean name="HelloWorldNode" parent="NodeWithDirectory">
11     <property name="agents">
12       <list>
13         <ref bean="HelloWorldAgent" />
14       </list>
15     </property>
16   </bean>
17
18   <bean name="HelloWorldAgent" parent="SimpleAgent" scope="prototype">
19     <property name="agentBeans">
20       <list>
21         <ref bean="HelloWorldBean" />
```

```
22        </list>
23      </property>
24    </bean>
25
26    <bean name="HelloWorldBean" class="examples.helloworld.HelloWorldBean" scope="
          prototype">
27      <property name="executionInterval" value="1000" />
28    </bean>
29
30  </beans>
```

Now let's start the agent, or more precisely, the agent node (Listing 2.3).

Listing 2.3: Starting an agent bean

```
1  // start node, wait a few seconds, and stop the node
2  SimpleAgentNode node = (SimpleAgentNode) new ClassPathXmlApplicationContext("
       hello_world.xml").getBean("HelloWorldNode");
3  Thread.sleep(5000);
4  node.shutdown();
```

First, we start the AgentNode using `ClassPathXmlApplicationContext` (a class provided by the Spring framework) with our configuration file as argument. While the main thread sleeps, the HelloWorldBean will be executed a few times, each time printing "Hello World" to the terminal, before the node is finally shut down.

Now that we're over with the single agent *Hello World* example, let's develop a *Multi-Agent System* (MAS).

2.1.2 Ping Pong: The MAS Hello World

Lets look at the real MAS Hello World: Ping Pong. Here is the scenario: We have a node with two agents, PingAgent and PongAgent. PingAgent is continually sending Pings to the PongAgent, and upon receiving a Ping, the PongAgent replies with a Pong. While still a very simple example, it covers most of the basic programming aspects of JIAC: the agent configuration, AgentBeans, actions, ontology, facts, the agent memory, and communication amongst agents.

Listing 2.4: Ping Pong AgentNode Configuration

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    xsi:schemaLocation="http://www.springframework.org/schema/beans
5    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
6
7    <import resource="classpath:de/dailab/jiactng/agentcore/conf/AgentNode.xml" />
8    <import resource="classpath:de/dailab/jiactng/agentcore/conf/Agent.xml" />
9    <import resource="classpath:de/dailab/jiactng/agentcore/conf/JMSMessaging.xml"/>
10
11   <bean name="PingPongNode" parent="NodeWithJMX">
12     <property name="agents">
13       <list>
14         <ref bean="PingAgent" />
15         <ref bean="PongAgent" />
16       </list>
17     </property>
```

```
18    </bean>
19
20    <bean name="PingAgent" parent="SimpleAgent" scope="prototype">
21      <property name="agentBeans">
22        <list>
23          <ref bean="PingBean" />
24        </list>
25      </property>
26    </bean>
27
28    <bean name="PongAgent" parent="SimpleAgent" scope="prototype">
29      <property name="agentBeans">
30        <list>
31          <ref bean="PongBean" />
32        </list>
33      </property>
34    </bean>
35
36    <bean name="PingBean" class="examples.pingpong.PingBean" scope="prototype">
37      <property name="executionInterval" value="1000" />
38      <property name="logLevel" value="INFO" />
39    </bean>
40
41    <bean name="PongBean" class="examples.pingpong.PongBean" scope="prototype">
42      <property name="logLevel" value="INFO" />
43    </bean>
44
45  </beans>
```

The configuration is given in Listing 2.4. We have one node holding two agents. The node has to support communication, but the `CommunicationBean` is inherited from `SimpleAgent`, so we get this automatically. However, you have to add the `PingBean` and the `PongBean`, implementing the behavior described above. Finally, we use the agents' built-in logging mechanism to print some `INFO`.

Now let's have a look at the `PingBean` (Listing 2.5):

Listing 2.5: Ping Bean

```
1  package examples.pingpong;
2  import de.dailab.jiactng.agentcore.AbstractAgentBean;
3
4  public class PingBean extends AbstractAgentBean {
5
6    private IActionDescription sendAction = null;
7
8    @Override
9    public void doStart() throws Exception {
10     super.doStart();
11     log.info("PingAgent - starting....");
12     log.info("PingAgent - my ID: " + this.thisAgent.getAgentId());
13     log.info("PingAgent - my Name: " + this.thisAgent.getAgentName());
14     log.info("PingAgent - my Node: " + this.thisAgent.getAgentNode().getName());
15
16     // Retrieve the send-action provided by CommunicationBean
17     IActionDescription template = new Action(ICommunicationBean.ACTION_SEND);
18     sendAction = memory.read(template);
19     if (sendAction == null) {
20       sendAction = thisAgent.searchAction(template);
21     }
22     // shorter: retrieveAction(ICommunicationBean.ACTION_SEND);
```

```
23
24      // If no send action is available, check your agent configuration.
25      // CommunicationBean is needed
26      if (sendAction == null)
27        throw new RuntimeException("Send action not found.");
28    }
29
30    @Override
31    public void execute() {
32
33      // Retrieve all Pong Agents
34      List<IAgentDescription> agentDescriptions = thisAgent.searchAllAgents(new
            AgentDescription());
35      for (IAgentDescription agent : agentDescriptions) {
36        if (agent.getName().equals("PongAgent")) {
37
38          // Send a 'Ping' to each of the PongAgents
39          JiacMessage message = new JiacMessage(new Ping("ping"));
40          IMessageBoxAddress receiver = agent.getMessageBoxAddress();
41
42          // Invoke sendAction
43          log.info("PingAgent - sending Ping to: " + receiver);
44          invoke(sendAction, new Serializable[] { message, receiver });
45        }
46      }
47    }
48  }
```

Here we have a new method: `doStart()`, which is part of the agent's *life-cycle*. We will have a closer look at the agent's life cycle in Section 2.3; this method is called when the agent is started. What it does in our example: After printing some status information about the agent, it retrieves the `send` action provided by the `CommunicationBean`. Thus, an action provided by one Bean can be retrieved and used in another Bean without needing a direct reference to that Bean. Moreover, actions can also be made available to other agents! We will go into more detail on actions in Section 2.4.

What do we do with the `send` action? We send a series of Pings to the Pong agent! For this purpose, we have to search for all agents known to this agent using the respective method and look for agents whose name is "PongAgent".

Our ontology consists of two concepts or classes: *Ping* and *Pong*. We create a Ping (Listing 2.6) and put it as payload into a `JiacMessage`. The JiacMessage is like an envelope for the actual message, providing e.g. information about who it came from. Communication in JIAC and everything related to this topic will be explained in Section 2.6. Now we can invoke the `send` action with the message and the receiver (which we get from the agent description) as input parameters.

Listing 2.6: Ping Class

```
1  public class Ping implements IFact {
2
3    private String message;
4
5    public Ping(String pingMessage) {
6      this.message= pingMessage;
7    }
8
9    public String getMessage() {
```

```
10      return message;
11    }
12
13    public void setMessage(String message) {
14      this.message = message;
15    }
16  }
```

The message holding the Ping has been sent and will arrive at the `PongAgent` as a *Fact* in its knowledge base, or also called memory, so let's have a look at the `PongBean` (Listing 2.7):

Listing 2.7: Pong Bean

```
1  public class PongBean extends AbstractAgentBean {
2    [ ... ]
3
4    @Override
5    public void doStart() throws Exception {
6      super.doStart();
7      [ same as PingBean ]
8
9      // listen to memory events, see MessageObserver implementation below
10     memory.attach(new MessageObserver(), new JiacMessage(new Ping("ping")));
11   }
12
13   private class MessageObserver implements SpaceObserver<IFact> {
14
15     public void notify(SpaceEvent<? extends IFact> event) {
16       if (event instanceof WriteCallEvent<?>) {
17         WriteCallEvent<IJiacMessage> wce = (WriteCallEvent<IJiacMessage>) event;
18         // a JiacMessage holding a Ping with message 'Ping' has been
19         // written to this agent's memory
20         log.info("PongAgent - ping received");
21
22         // consume message
23         IJiacMessage message = memory.remove(wce.getObject());
24
25         // create answer: a JiacMessage holding a Ping with message 'pong'
26         JiacMessage pongMessage = new JiacMessage(new Ping("pong"));
27
28         // send Pong to PingAgent (the sender of the original message)
29         log.info("PongAgent - sending pong message");
30         invoke(sendAction, new Serializable[] { pongMessage, message.getSender()
               });
31       }
32     }
33   }
34 }
```

The `doStart()` method is similar to that of the PingBean, as we will need the `send` action again. Further, we will attach a listener, so-called *SpaceObserver*, to the agent's memory, which is notified each time something is read, written to, removed from, or updated in the memory. In this case, we are only interested in messages that have a Ping as payload, so we can narrow it down to this by providing the template accordingly.

The way the agent's memory works in detail will be explained later in Section 2.5, but for now let's have a look at the observer, implemented below. From the template, we already know that the notification has something to do with a message holding a Ping,

so we just have to see what kind of `SpaceEvent` we are notified of (the agent memory is a "tuple space", thus the name SpaceEvent). In case of a `WriteCallEvent` we remove the message from the memory. Then, we create a new message with a Pong and send it to the sender of the original message.

Now you can run the Ping-Pong example using a similar starter script as the one for Hello World. It is left as an exercise to the reader to extend the Ping Bean so it receives the Pongs, that have been sent in reply to the Pings, and to print them to the terminal. You can also make a *Ping-Peng-Pong*. And, finally, just start the Ping-Pong-Node twice and see what happens.

In this Section we have seen how to implement and run some very simple JIAC agents, using the basic notions of agent configuration, agent Beans, actions, communication, and the agent's memory. In the remainder of this Chapter we will introduce each of these aspects in more detail.

2.2 Agent Configuration

Agent is the main concept in all agent-oriented techniques and frameworks. In JIAC, an agent is an active part in a multi-agent system and interacts with other agents to achieve one or more goals.

The JIAC agent has two basic components: *memory* and *execution*. Both parts are essential for making an agent, and are already defined in the basic `SimpleAgent` (Listing 2.8).

Listing 2.8: JIAC Basic Configuration

```
1  <bean name="SimpleAgent" class="de.dailab.jiactng.agentcore.Agent" abstract="
       true">
2    <property name="memory" ref="Memory" />
3    <property name="execution" ref="SimpleExecutionCycle" />
4    <property name="executionInterval" value="10"/>
5  </bean>
```

Memory is the facts-base of the agent. The default implementation is a tuple space and can be exchanged. Every AgentBean has a direct reference to the memory and thus it is the shared memory of AgentBeans or their blackboard. How to use memory is described in 2.5.

Execution is the virtual engine or control of how the agent works internally. The default implementation is a simple execution cycle and can be exchanged. The `SimpleExecutionCycle` mainly does two things:

- call the `execute()` method of other AgentBeans if required
- control action invocation, result delivery and session handling.

The `SimpleExecutionCycle` runs in a loop with a default period of 10 milliseconds. Use the `executionInterval` property to change this.

An agent can be extended using AgentBeans to provide application specific functionality and implementation. Therefore, if you want to add AgentBeans, you use the list property `agentBeans`. Listing 2.9 shows an example from a virtual cowboy, who percepts the virtual world and then decides to either explore the world, drive some cows to the corral or steal some cows from other cowboys, every behavior implemented using a different AgentBean.

Listing 2.9: `agentBeans` Property

```
<bean name="CowboyAgent" parent="SimpleAgent" scope="prototype">
  <property name="agentBeans">
    <list>
      <ref bean="ServerCommBean" />
      <ref bean="PerceptionBean" />
      <ref bean="ExplorerBean" />
      <ref bean="DriverBean" />
      <ref bean="ThiefBean" />
    </list>
  </property>
</bean>
```

See the following chapter 2.3 for more details on AgentBeans.

2.3 Agent Beans

The usual way to extend agents with new behaviors and capabilities is to create an Agent Bean that offers the desired functionality. All Agent Beans need to implement certain interfaces for life-cycle- and management-operations. Fortunately, most of this is rather generic, and in most cases you can simply extend the class `AbstractAgentBean`, which implements the necessary interfaces and provides some useful fields:

- `protected Log log`: The logger-instance. Can be used to create log messages.
- `protected IAgent thisAgent`: A reference to the agent object. Can be used to perform operations on the agent.
- `protected IMemory memory`: A reference to the agents memory. Can be used to store and retrieve data.

One useful trait of Agent Beans is to provide Actions, which will be topic in Section 2.4. Further, they can perform some operation when the agent changes its state, depending on its life-cycle, or periodically, depending on its Execution Cycle.

2.3.1 The Lifecycle

The life-cycle (Fig. 2.2) represents the states an agent can be in. Like the agent and the agent node, each Agent Bean implements the interface `ILifecycle` which is used for controlling the Bean's life-cycle in accordance to the agent's life-cycle. The interface declares methods such as `init()`, `start()`, `stop()`, and `cleanup()`, for changing between life-cycle states. The class `AbstractLifecycle`, which is the super class of

`AbstractAgentBean`, implements these methods and provides a number of additional methods, such as `doInit()`, `doStart()`, etc., where you can hook in code that shall be done when changing to this life-cycle state, like looking up needed Actions, connecting to some data base and other initialization and/or finalization work.

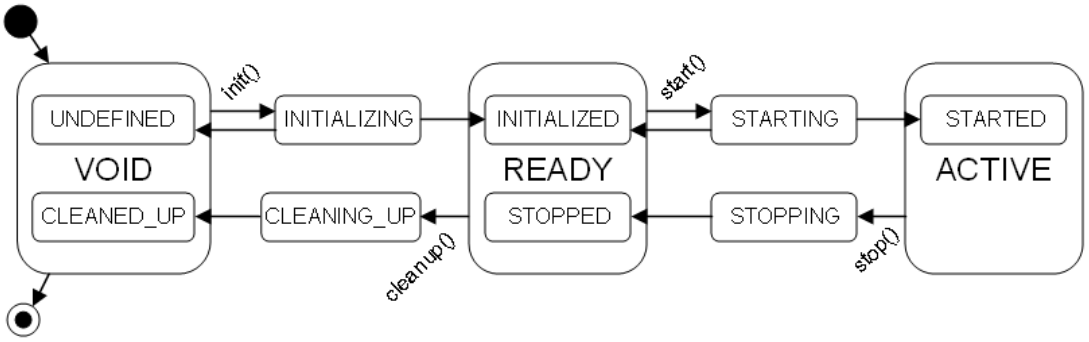


Figure 2.2: Life-cycle

2.3.2 The Execution Cycle

Sometimes, one wants the Bean to do something periodically, e.g. to check whether some condition applies. For this purpose, the `AbstractAgentBean` provides the `execute()` method. For each of the agent's Beans, the `execute()` method is executed periodically by the agent's Execution Cycle, given that both the agent and the Agent Bean are in the state STARTED. The execution interval (the minimum interval between two calls of the `execute()` method in milliseconds) has to be specified in the configuration file (see e.g. Listing 2.10). If the execution interval is not set, the Bean's `execute()` method will *not* be called.

Listing 2.10: Starting an agent bean

```
1 <bean name="ExampleBean" class="examples.ExampleBean" scope="prototype">
2   <property name="executionInterval" value="1000" />
3 </bean>
```

2.4 Actions

One very useful trait of JIAC Agent Beans is to provide Actions. The difference between methods and Actions is that an Action can be invoked by any other Bean of that agent (and, depending on the Action Scope, by other agents as well). Further, Actions are invoked asynchronously, so the agent can e.g. delegate some work to another agent, by invoking the respective action, and will be notified whenever the action has been performed.

The different actions are described in an `ActionDescription`, specifying the name of the action, its input and output values (an action can have more than one output), and

other information, like the AgentBean that provided that action, and the scope in which the action is available.

In a nutshell, actions in JIAC work like this:

1. each agent can declare and provide actions via the `IEffector.getActions` method
2. depending on their scope, those action are distributed to other agents on the same node or on other nodes
3. an action is retrieved by sending a template, providing e.g. the name of the action, to the directory and receiving any actions that match that template
4. the action is invoked, using e.g. the `invoke` method; this will create a `DoAction` object in the agent's memory, encapsulating the action and its parameters
5. if the provider of the action is a different agent, the `DoAction` is automatically sent to that other agent, ending up in its memory
6. if the action belongs to the agent himself, then the DoAction is passed to the `IEffector.doAction` method of the agent bean that provided the action
7. the `doAction` method will execute the action, possibly delegating to other methods of that bean, or even to other agents, and eventually assemble and return an `ActionResult` with the result of the computation and/or an error message
8. that `ActionResult` is again placed in the agent's memory, messaged to the agent that invoked the action (in case it's not the same agent), eventually triggering a `ResultReceiver` for handling the result of the invocation

All this happens behind the scenes; you usually do not need to worry about it.

2.4.1 Using Actions

Generally, using an action involves two to three steps (the last one being optional):

- create a template,
- find the actual action,
- invoke the action,
- get the action result.

We have already seen an example of using an action in the Ping Pong example, where we used the `send` action provided by the Communication Bean to send the Ping. A shorter way to the same effect is using the helper method `retrieveAction(actionName)`. For more complex templates, e.g. also specifying an actions parameter types or visibility scope besides its name, you have to use a proper template.

All the agent's actions are represented by an Action Description in the agent's memory, and thus can be looked up using `memory.read`. However, actions provided by other agents have to be looked up in the Directory. For this, use `thisAgent.searchAction(actionDescription)`. However, you can just as well use `thisAgent.searchAction` for both the agent's own and for other agent's actions, it works for both.

Listing 2.11: Invoking an Action

```
1 // create template and find action
2 IActionDescription template = new Action(MyBean.ACTION_DOSOMETHING);
3 IActionDescription act = thisAgent.searchAction(template);
4
5 // invoke action and process result
6 invoke(act, new Serializable[] {param1, param2, ...}, new ResultReceiver() {...});
```

In the example we created a new action with a name corresponding to some existing action. The created action can not be executed directly because it is only a template and not bound to any implementation. For performing the action, you need to find the actual action which fits to the template. You can then invoke the action, either using `invoke` or `invokeAndWaitForResult`. The former will invoke the action asynchronously and can be passed a `ResultReceiver` providing a callback for processing the action's result. The latter will invoke the action synchronously and return the result directly.

Note: Do *not* use `invokeAndWaitForResult` in the `doStart()` method! As written above, the agent's Execution Cycle will perform actions only when the agent is in the STARTED state, meaning that in this case, the agent would dead-lock. Generally, you should *not* search for actions in `doStart` either, as actions by other agents will only get published in the starting phase, so depending on the order in which agent start, some actions may not yet be available when searching for them. Better search for actions directly when you need them, e.g. in the `execute` method, possibly caching and retrieving previously found actions.

Also, the agent can dead-lock if use `invokeAndWaitForResult` from within another action. The reason is that the `SimpleAgent` agent will only start the next action when the execution of the current action has completed, but the action can not complete until the other action has completed. To prevent this, you can use `NonBlockingAgent` instead of `SimpleAgent` in your agent configuration file. If you use this method for an action which has no return parameter then this method will not wait for a result.

2.4.2 Providing Actions

We now know how to *use* Actions. If you want your Agent Bean to *provide* an Action, there are two possible ways to do this:

The "hands-on" approach is to extend the class AbstractAgentBean and additionally implement the `IEffector` interface. This interface requires you to implement two methods. The first, `getAction()`, is called during initialization of the agent and is used to retrieve the list of actions that your Bean provides. The method must simply return a list of Action objects, that describe the offered actions. The second, `doAction(DoAction)`,

is called by the agent's ExecutionCycle whenever it finds a DoAction object in the agent's memory, so this is where you should delegate to the actual implementation of the Action depending on the provided DoAction object.

A much simpler approach, that is more convenient for everyday-use, is to extend the class `AbstractMethodExposingBean`. This class is an extension of `AbstractAgentBean` and provides a mechanism based on Java Annotations, to expose usual Java methods as Actions within an agent. All you have to do is to put the `@Expose` annotation to your method. The rest, i.e. providing the Action descriptions and calling the appropriate method when finding the corresponding `DoAction`, will be handled by the super class. An example for an annotated Method would look like this:

Listing 2.12: Exposing an Action

```
1 public static final String ACTION_DOSOMETHING = "package.MyBean#doSomething";
2 @Expose(name=ACTION_DOSOMETHING, scope=ActionScope.NODE)
3 public Foo doSomething(String text, int number) {
4   // do something
5   return foo;
6 }
```

The name given in the annotation is the name by which your action will be registered within the platform. We suggest that you choose these names carefully within larger projects. The name is optional, and if you don't provide a name, the system will simply use the fully qualified classname followed by the name of the method. Still, as a JIAC *coding convention* we recommend you to explicitly provide an action name and to store this name in a public field of the class, as shown in Listing 2.12. This way, it is very easy to see which methods are exposed by a given class and to retrieve the actions using that field, instead of typing the action name.

In addition to the Action name, we have also specified an Action scope. The Action scope NODE will make the Action available to every agent on the node. The default scope is AGENT.

Multiple Return Values One characteristic of JIAC actions is that they can have more than one return value. If you implement the `IEffector` Interface yourself, you will assemble the `ActionResult` yourself, so you are free to put whatever you want into the result. When using the `@Expose` annotation, you have to explicitly specify the return types, as seen in the next listing, otherwise the (singular) return type is determined by whatever the annotated Java method returns. In this case, just return the several values together in one array; the `doAction` method will then un-pack that array.

Listing 2.13: Exposing an Action

```
1 public static final String ACTION_DO_MANY_THINGS = "package.MyBean#doManyThings";
2 @Expose(name=ACTION_DO_MANY_THINGS, returnTypes={Foo.class, Bar.class})
3 public Serializable[] doManyThings(String text, int number) {
4   // do many things
5   return new Serializable[] {foo, bar};
6 }
```

2.5 Agent Memory

The default implementation of the agent's memory is a simple tuple space and provides a light-weight, easy-to-use and extensible Java tuple space implementation. The *SimpleSpace* tuple space may hold any Java objects and provides basic tuple space functionality. It uses *Java Reflection* to match the template objects to the objects in memory, comparing the several attributes of the different objects. It does *not* use `equals`. What's important to note: All the attributes that should be available for the matcher to compare either have to be `public`, or provide *both* `get` and `set` methods. A `final` attribute that has just a `get` method but no `set` method will *not* be considered when matching!

In principle, `SimpleSpace` can hold any Java object, but we have restricted *memory* to hold only objects that implement the `IFact` interface, which is an extension to `java.io.Serializable`. (We want developers to explicitly model the ontology, instead of just putting lots of anonymous Strings and Lists into the memory.) You have a set of four operators to work on the space: *write, read, update, remove*, which we will explain in the following. Access to memory is directly granted for AgentBeans through the `memory` variable.

The example: Imagine a Gold digger agent that perceives a two-dimensional grid world. A `Field` has `x,y` coordinates and a boolean variable that tells whether the field `hasGold` or not (Listing 2.14).

Listing 2.14: A Gold digger world field

```
1   public class Field implements IFact {
2
3       /** Fields must be public, or have a getter AND a setter. */
4       public Boolean hasGold;
5       public Integer x;
6       public Integer y;
7
8       /**
9        * <code>Field</code> constructor, also used to create templates
10       * for tuple spaced matching
11       *
12       * @param hasGold
13       * @param x
14       * @param y
15       */
16      public Field(Boolean hasGold, Integer x, Integer y) {
17          this.hasGold = hasGold;
18          this.x = x;
19          this.y = y;
20      }
21  }
```

If you search for an object with the method `memory.read(template)` then all object of the same type in the memory will be compared with the given `template` by comparing the template's attributes that are not `null` with the objects' parameters. If they fit then the object will be returned. For example, if only the `name` attribute of the object is not

`null` in the template then only the names of the other objects will be compared with the given name. Thus, an attribute that is `null` in the template acts as a wildcard and matches with any attribute value in the actual objects in memory.

Similarly, note that we are not using primitive data types, like `int` or `boolean`, but the according wrapper classes `Integer` and `Boolean`. This has the benefit that we can create a template with some of those fields set to `null` and use `memory.read` to find all the elements matching the template.

2.5.1 memory.write()

Now that we have perceived some information from our grid world, we make mental notes of some fields in the world (Listing 2.15):

Listing 2.15: write() to memory

```
1  memory.write(new Field(true, 2, 2));
2  memory.write(new Field(true, 2, 3));
3  memory.write(new Field(false, 4, 2));
4  memory.write(new Field(false, 5, 5));
```

2.5.2 memory.read()

For recalling a certain field, we can use a set of read operators. First, we want to remember the field at 2,2. For this, we use the `memory.read(template)` method. The space will return the first object that matches the template, or `null` if none matches (Listing 2.16).

Listing 2.16: read() from memory

```
1  memory.read(new Field(null, 2, 2));
```

The space API also allows you to wait until the fact is memorized or the call times out: `memory.read(template, timeout)`.

In case we want to remember all gold fields we use the method `readAll(template)`. A typed `java.util.Set` will be returned (may not contain any entry, but is never `null`; Listing 2.17):

Listing 2.17: readAll() from memory

```
1  memory.readAll(new Field(Boolean.TRUE, null, null));
```

Finally, if you want to retrieve all objects of a certain type from the tuple space, you may use the method `readAllOfType(class)`. This method returns a set of instances of the given class.

2.5.3 memory.remove()

Objects must be explicitly removed from the memory. All remove operators return the removed objects.

In our example, we want to remove the Field at coordinates 2,2 because it is no longer valid (Listing 2.18). The method will remove the first object that matches the template, and return it, if it is in the memory, or `null`, if no match exists.

Listing 2.18: remove() from memory

```
1  memory.remove(new Field(null, 2, 2));
```

Additionally, the call should only return when an object that matches the template is in the memory, or the call times out: `memory.remove(template, timeout)`.

Finally, we want to remove *all* objects that match a certain criterion. We use the `removeAll()` method to do this. In our example, row two should be removed (Listing 2.19):

Listing 2.19: removeAll() from memory

```
1  memory.removeAll(new Field(null, null, 2));
```

2.5.4 memory.update()

To *update* certain facts in memory, we supply a template that will match the interesting objects, and then another pattern that defines what to change. In the example, our digger agent has collected all gold, and the fields that formerly had gold, should be updated with the no-gold information (Listing 2.20):

Listing 2.20: update() memory

```
1  memory.update(new Field(Boolean.TRUE, null, null), new Field(Boolean.FALSE,
2  null, null));
```

2.5.5 Space Events

Memory will fire `SpaceEvent`s when some AgentBean has called an operation on it. There are four events that are fired:

- `WriteCallEvent` – a new object has been written to memory; the object is given in the event
- `UpdateCallEvent` – some objects have been updated in memory (related to the template given in the event)

- `RemoveCallEvent` – one objects has been removed from memory (related to the template given in the event)
- `RemoveAllCallEvent` – all objects (related to the template given in the event) have been removed from memory

You will receive `SpaceEvents` when you use a `SpaceObserver` as described in the next section.

2.5.6 Space Observer

You may *attach* a `SpaceObserver` to the memory to get notified when some AgentBean has worked on it.

First, you create your own SpaceObserver. Then, you attach it to memory. This observer will be notified on all operations on the memory. (Listing 2.21)

Listing 2.21: attach() a SpaceObserver to memory

```
1  SpaceObserver<IFact> myObserver = new SpaceObserver<IFact>() {
2
3    public void notify(SpaceEvent<? extends IFact> event) {
4      if (event instanceof WriteCallEvent) {
5        //do something
6      } else if (event instanceof UpdateCallEvent) {
7        //do something else
8      }
9    }
10 };
11 memory.attach(messageObserver);
```

If you want the SpaceObserver to only get notified on certain objects and their changes, you can use a second `attach` method that has an additional parameter for a template that describes the objects you are interested in. In Listing 2.22, we are interested in objects and changes of the second row of our world.

Listing 2.22: attach() a SpaceObserver to memory with template

```
1  memory.attach(messageObserver, new Field(null, 2, null));
```

To force the SpaceObserver to stop notifying you, you can detach the observer from memory: `memory.detach(myObserver);`

2.6 Communication

The default implementation of JIAC's communication components relies on ActiveMQ (http://activemq.apache.org/). The message broker is a component of the AgentNode and supplies messaging between all agents on its-self and on other nodes.

2.6.1 CommunicationBean

Each agent that need to communicate with other agents needs a *CommunicationBean*, providing the basic functionalities for sending and receiving messages.

If you use the `SimpleAgent` definition as parent for your configuration, the agent is automatically equipped with a CommunicationBean. If you use a full configuration though, you have to provide a communication property (Listing 2.23).

Listing 2.23: Add a CommunicationBean

```
1   <bean name="MyAgent" parent="SimpleAgent" scope="prototype">
2       <property name="agentBeans">
3           <list>
4               ...
5           </list>
6       </property>
7   </bean>
8
9   <bean name="MyAgent" class="de.dailab.jiactng.agentcore.Agent" scope="
        prototype">
10      <property name="memory" ref="Memory" />
11      <property name="execution" ref="SimpleExecutionCycle" />
12      <property name="executionInterval" value="10"/>
13      <property name="communication" ref="CommunicationBean" />
14      <property name="agentBeans">
15          <list>
16              ...
17          </list>
18      </property>
19  </bean>
```

The `CommunicationBean` registers the `IMessageBoxAddress` of the agent at the broker, in order to be able to send messages to that agent directly. Furthermore, the `CommunicationBean` provides some actions that can be used by the agent containing the `CommunicatioBean`:

- **register/unregister** – an address together with a template that is used as filter for incoming messages. Messages that do not fit to the specified template will be ignored.
- **join/leave** – a group. This is kind of a message channel for multi-cast communication. You send to and receive from all agents that joined the group.
- **send** – a direct or group message to either an agent or a group of agents.

In 2.1.2, we have already used the *send* action of the `CommunicationBean`. A recapitulation of what we did there:

- `search` – the *send* action in the memory
- `invoke` – the *send* action with two parameters: first, the *message*, second, the *receiver*, which is either an agent or a group of agents under the same group address

The following code listing shows how to retrieve the communication address for an individual agent and for a message group:

Listing 2.24: Acquiring communication adresses for individual agents and for message groups

```
IAgentDescription agent = ...
IMessageBoxAddress receiver = agent.getMessageBoxAddress();
IGroupAdress groupAdress = CommunicationAddressFactory.createGroupAddress(
    channel);
```

2.6.2 MessageBroker

By default, all AgentNodes in an IP subnet find each other, and all the contained agents can talk to each other. If you want to change this behavior, modify the broker settings.

To separate the agents of your application change the discovery channel of the broker in the agent-node configuration file as shown in Listing 2.25.

Listing 2.25: Change AgentNode Configuration to separate your agent nodes and agents

```
<bean name="MyBroker" parent="ActiveMQBroker" scope="prototype"
    lazy-init="true">
  <property name="connectors">
    <set>
      <ref bean="MyTCPConnector" />
    </set>
  </property>
</bean>

<bean name="MyTCPConnector" parent="ActiveMQTransportConnector"
    scope="prototype" lazy-init="true">
  <property name="transportURI" value="tcp://0.0.0.0:0" />
  <property name="discoveryURI" value="smartmulticast://default?group=myChannel"
  />
</bean>
```

To change the topology of your application, in case you need a gateway node or similar, you may configure one node as master and others as slaves. This is transparent to agents, meaning that it does not matter how the broker is configured on the user side. Just change the configuration of one node to be the master (Listing 2.26) and the other nodes to be the slaves (Listing 2.27).

Listing 2.26: Configure one node to be the master

```
<bean name="StaticMasterConnector"
    class="de.dailab.jiactng.agentcore.comm.broker.ActiveMQTransportConnector"
    scope="prototype">
  <property name="transportURI" value="tcp://0.0.0.0:6789" />
</bean>
```

Listing 2.27: Configure one node to be the slave

```
<bean name="StaticSlaveConnector"
    class="de.dailab.jiactng.agentcore.comm.broker.ActiveMQTransportConnector"
    scope="prototype">
  <property name="networkURI" value="static:(tcp://master.I.P.number:6789)"/>
  <property name="duplex" value="true" /> <property name="networkTTL" value="255
      " />
  <property name="transportURI" value="tcp://0.0.0.0:0" />
</bean>
```

3 The JIAC Library

In this chapter, we introduce and briefly describe a number of supplementary modules that enhance JIAC with some additional features and functionality.

3.1 Reactive Agents and Rules

The module `de.dailab.jiactng.ruleEngine` allows to include a Drools rule engine[1] into a JIAC agent. It uses Drools in version 5.3.1 and allows an agent's reactive behaviour to be easily modelled in Drools Java-based syntax and to be deployed or removed at runtime.

To include the Rule Engine Bean into your JIAC agent, import this into your Spring configuration file:

Listing 3.1: Importing Rules Config

```
1 <import resource="classpath:de/dailab/jiactng/ruleengine/conf/RuleEngineBean.
      xml" />
```

This configuration defines a number of agent bean:

- `StatefulRuleEngineBean`: Wrapper for stateful Drools rule engine, providing actions for deploying rules, adding facts, firing rules, etc.
- `StatelessRuleEngineBean`: Stateless variant of the same rule engine wrapper
- `SituationBean`: Rule-Engine bean synchronizing itself with the agent's memory
- `TimerBean`: Push the current time and date into the rule engine for easier creating of time-based rules

By default, the beans will check the rules in their `execute` method, which is executed each second. Also, rules to be deployed when the agent is initialized can be given in the configuration using the `rulesFiles` list property. Those files are expected to be found on the classpath.

3.1.1 Rule Engine Bean

The rule engine bean is a simple wrapper for the Drools rule engine. It provides JIAC actions for deploying and undeploying rules, for adding and updating facts in the rule

[1] Drools: http://drools.jboss.org/

engine's working memory, for setting global variables and of course for firing the rules. Those actions are declared in the `IJiacRuleEngineBean` interface.

Rules can be deployed from a number of sources: When invoking the `deployRules` action, the parameter can either be a URL pointing to the Drools rules file, or a string, holding the content of that file (or some dynamically created rules not backed by any actual file), or a readily parsed Drools `Package` object.

The state of the rule engine – the currently deployed rules and the facts in the rule engine's working memory – can be observed using the `RuleEngineMonitor`. To start an instance of the monitor, you can add this to your code:

Listing 3.2: Adding Rule Engine Monitor

```
1 RuleEngineMonitor monitor= new RuleEngineMonitor();
2 monitor.registerRuleEngine(referenceToRuleEngineBean);
3 monitor.setDefaultCloseOperation(WindowConstants.HIDE_ON_CLOSE);
4 monitor.setVisible(true);
```

3.1.2 Situation Bean

The Situation Bean is another variant of the rule engine bean. Instead of exposing the rule engine's operations as JIAC actions and passively waiting for those actions to be invoked, it integrates with the agent's memory, synchronizing any facts that are added to or removed from the memory with the rule engine's working memory, and vice versa.

This way, the `SituationBean` can be used for realizing reactive behaviour in JIAC agents. For example, a rule can check for incoming JIAC messages, or for any other IFact or combination of IFacts in the agent's memory, and then alter those IFacts, remove the message from the agent's memory, or add a `DoAction` to the memory, thus scheduling the execution of an action.

While most of this could also be realized using a `SpaceObserver`, the situation bean has the advantage that (a) some situations can easier be expressed in Drools syntax than using a number of loops and `if/else` checks in an observer, and (b) those reaction rules can be deployed at runtime.

Support for time-based rules is rather limited in Drools, thus we added the `TimerBean`. When added to the JIAC agent, this bean will periodically update a `TimeFact` in the agent's memory, which is then synched to the rule engine's working memory. This `TimeFact` can then be used in a rules condition to easily test for the current time, e.g. `TimeFact(dayOfWeek in ("Saturday", "Sunday"))` or `TimeFact(timeOfDay > "12:00:00" && < "16:00:00")`

3.1.3 Example

A simple example rule for the `SituationBean` is shown in listing 3.3. This rules does not have any practical relevance; it exists solely for illustrating some of the possibilities offered by the JIAC rule engine bean and particularly the situation bean.

Listing 3.3: Example Rule

```
1  rule "Example Rule"
2  when
3      action : Action(name == "DoSomething")
4      TimeFact(dayOfWeek in ("Saturday", "Sunday"),
5               timeOfDay > "12:00:00" && < "16:00:00")
6      stuff : SomeStuff(foo == "Bla", bar < 42)
7      jiacMessage : IJiacMessage()
8      eval("channel".equals(jiacMessage.getHeader(IJiacMessage.Header.SEND_TO)))
9      payload : SomeOtherFact(interesing == true) from jiacMessage.payload
10 then
11     System.out.println("Example-Rule fired");
12     insert(new DoAction(action, null, new Serializable[] {stuff, payload}));
13     retract(jiacMessage);
14 end
```

The rule's condition checks for the existence of a certain `Action` in the agent's memory, and using the `TimeFact` it checks whether the current time and data match some condition. It further looks for a `JiacMessage` being sent to a particular message channel and with a certain kind of payload.

If all those things are matched in the agent's memory, the rule's effect is to first write some logging information, then to create a `DoAction` object using the formerly matched action and facts and to add it to the agent's memory, thus scheduling the execution of that action, and finally to remove the message from the memory.

For a comprehensive introduction to Drools and documentation of the Drools syntax, please refer to the Drools documentation[2]. There also exists an Eclipse plugin for Drools editing support, featuring a graphical editor for creating and validating rules, and other things.

3.2 Migration

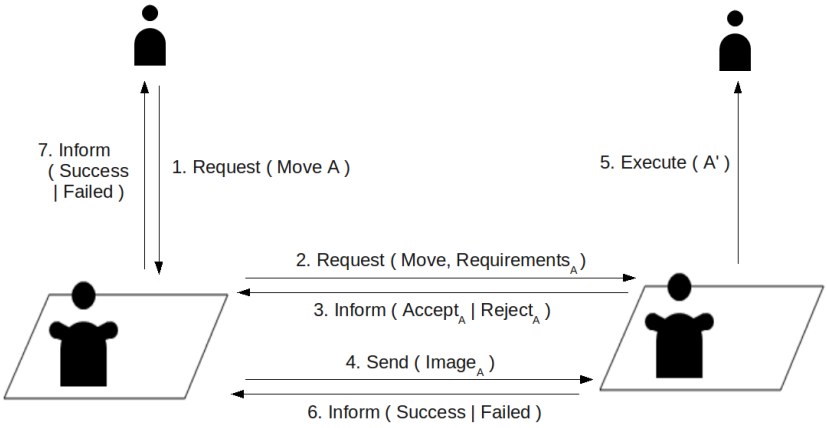
The module `de.dailab.jiactng.basicServices.agentMigration` provides JIAC nodes and agents with support for weak and strong migration and cloning. The module consists of a specific AgentNodeBean adding migration capabilities to the node, an abstract Agent implementing the required life-cycle states on the side of the agent, and a process for saving and restoring the migrated agents and a protocol for sending that agent to the target node (see Figure 3.1).

For an agent to be migratable, the agent has to meet certain prerequisites:

- it has to be a `MobileAgent`, not a `SimpleAgent`
- the nodes has to be a `NodeWithMobility` (extending `NodeWithJMX`)
- all the fields of the agent's beans that are to be migrated have to (i) be annotated with `@Migratable`, (ii) be serializable and (iii) be public or provide accordingly named public getter and setter methods.

[2]Drools Documentation: http://drools.jboss.org/documentation

Figure 3.1: Migration protocol



3.3 RESTful Services

The `RESTfulServices` module provides a convenient way to expose actions of agents as a RESTful interface[3]. It is designed to work out of the box and with already existing agent systems. Nonetheless it is flexible enough to let the developer specify the REST semantics by himself.

The most common use-case is to build a web-application that provides means for interaction with agents via the web-browser. The `RESTfulServices` module supports the JSON[4] data format by default, and thus is compatible with nearly every browser today. A typical architecture is depicted in Figure 3.2. The REST architecture allows for independent development of agents and web applications, therefore many steps of the development cycle can be parallelised. The integration into the browser enables the developer to use modern state of the art web technologies and provides a variety of possibilities for human-agent interaction.

3.3.1 Prerequisites

As first step the dependency has to be added to the `pom.xml` file, which is shown in Listing 3.4.

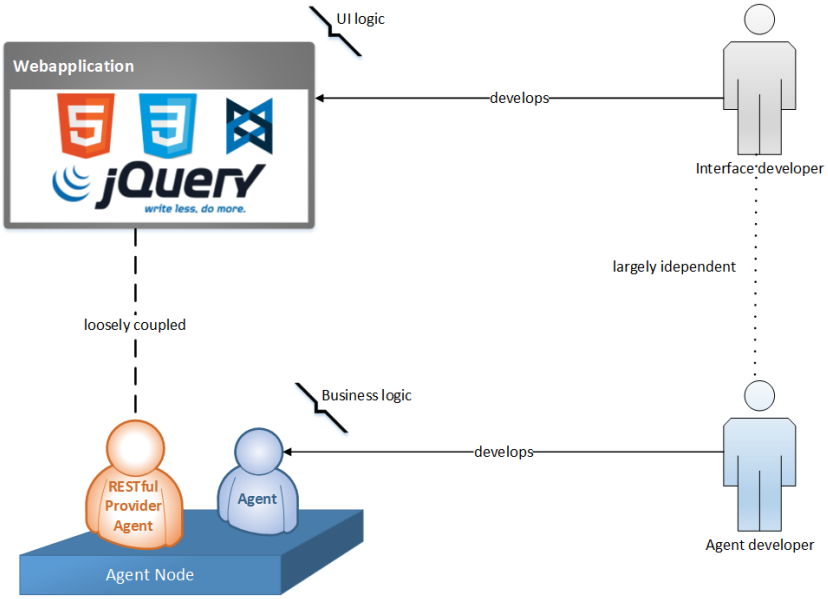
Listing 3.4: Include RESTful Services dependency

```
<dependency>
    <groupId>de.dailab.jiactng.basicServices</groupId>
    <artifactId>RESTfulServices</artifactId>
    <version>${jiac.version}</version>
</dependency>
```

[3] Representational state transfer: http://en.wikipedia.org/wiki/Representational_state_transfer

[4] JSON: http://json.org/

Figure 3.2: RESTful Architecture



Secondly, the `RESTfulProvider Agent` needs to be included in one or more nodes of the multi-agent system. Additionally the `WebServer` module needs to be present for the REST interface to work. It will also define the address and port under which the interface will be accessible. Listing 3.5 shows how this part of the config could look like.

Listing 3.5: Example RESTful Service configuration

```
<import resource="classpath:de/dailab/jiactng/basicservices/webserver/conf/
    webserver.xml" />
<import resource="classpath:de/dailab/jiactng/rsga/conf/rsga.xml" />

<bean name="RESTPlatform" parent="NodeWithJMX">
    <property name="agentNodeBeans">
      <list merge="true">
        <ref bean="WebServer" />
      </list>
    </property>
    <property name="agents">
      <list>
        <ref bean="RESTfulProviderAgent" />
        <ref bean="TestAgent" />
      </list>
    </property>
</bean>
```

Now the `RESTfulProvider` can expose actions it is able to see and lies within its own scope, which can be configured with the `scope` property.

3.3.2 Usage

The `RESTfulProvider` converts all of the found actions to a REST semantic automatically. Additionally it is possible to define these yourself by using JAX-RS annotation[5]. They are included by default and require no further configuration. Listing 3.6 shows some exemplary actions.

Listing 3.6: RESTful Example

```
1  // AbstractRESTfulAgentBean is needed for the JAX-RS annotations, otherwise it is
       not required
2  public class TestRESTfulAgentBean extends AbstractRESTfulAgentBean {
3
4      @POST
5      @Path("/add")
6      @Expose(scope = ActionScope.WEBSERVICE)
7      // Will become /TestAgent/add
8      public int add(@QueryParam("x") int x,
9                     @QueryParam("y") int y) {
10         return x + y;
11     }
12
13     // Will become: /TestAgent/getContact
14     @Expose(scope = ActionScope.WEBSERVICE)
15     public Contact getContact(Integer id) {
16       return contacts.get(id);
17     }
18 }
```

We assume that this agent bean belongs to an agent named `TestAgent`. The pattern to access the REST interface is `http://[address]/api/[agentName]/[actionName|path]`. Here, `/api` is the `contextPath` which can be set in the properties of the `RESTfulProviderBean`.

3.3.3 Invocation

Arguments can be passed as form-data, either with the parameter name, or `arg[x]` for unnamed arguments with x as the argument position. Complex data-types have to be in the JSON format. Listing 3.7 shows how an typically request could look like.

Listing 3.7: RESTful Invocation

```
1  curl --data "arg0={\"name\":\"Max\"}" http://localhost:8080/api/TestAgent/
       createContact
```

It is also possible to send multiple arguments in the JSON format on the request body, as seen in Listing 3.8. In this case it is important to set the header `Content-Type: application/json`, so the agent knows the format of the received data.

Listing 3.8: RESTful Invocation

```
1  curl --data "{\"x\":\"1\", \"y\": 2}" --header "Content-Type: application/json"
       http://localhost:8080/api/TestAgent/add
```

A list of all found actions and their parameter can be retrieved via http://localhost:8080/api/service.json, which can be very helpful for testing purposes.

[5] JAX-RS: https://jax-rs-spec.java.net/

3.3.4 Agent WebViewer

To simplify the developing process when working with the REST interface, there is handy tool based entirely on the RESTful module itself. The `AgentWebViewer` provides a web frontend to view and interact with all running agents, that are exposed via the `RESTfulProvider`. To use it, simply include the dependency in the `pom.xml` and add a `WebServer` plus the `AgentWebViewer` to your node. (When the node has an AgentWebViewer, no addition RESTfulProvider agent is required on the same node.)

Listing 3.9: Agent WebViewer dependency

```
<dependency>
    <groupId>de.dailab.jiactng.tools</groupId>
    <artifactId>AgentWebViewer</artifactId>
    <version>${project.version}</version>
</dependency>
```

Listing 3.10: Agent WebViewer config

```
<import resource="classpath:de/dailab/jiactng/tools/agentviewer/conf/
    AgentWebViewer.xml" />

<bean name="TestNode" parent="NodeWithJMX">
    <property name="agentNodeBeans">
      <list merge="true">
        <ref bean="WebServer" />
      </list>
    </property>
    <property name="agents">
            <list>
                <ref bean="AgentWebViewer" />
                <ref bean="TestAgent" />
            </list>
        </property>
</bean>
```

To access the web application open http://localhost:8080/ in your browser. For more information and configuration options please refer to the JavaDoc or README file of the `RESTfulServices` module or `AgentWebViewer`

3.4 Business Process Execution

JIAC can be used for executing business processes modelled in BPMN using the "Visual Service Design Tool" in two ways:

- VSDT diagrams can be exported to JIAC Agent Beans using one of the VSDT's export wizards,
- VSDT diagrams can be directly interpreted by a JIAC agent, using the VSDT interpreter bean.

For more about the VSDT and how to use it, please refer to http://www.jiac.de/development-tools/jiac-toolipse/visual-service-design-tool/. A formal description of the mapping from BPMN to (JIAC) agents can be found in [4].

3.4.1 Using the Transformation

How to use the VSDT and how to trigger the export wizard to trigger the transformation to JIAC agent beans is explained in the VSDT's manual [2], linked on the above web site.

Once the agent beans have been created, the only thing needed to do is to include them in one of your JIAC agents (refer to them in that agent's Spring configuration file).

The generated agent beans will extend the `AbstractWorkflowBean`, which is part of the `agentCore` module, so no further dependencies are required. The workflow is mapped to an according method in the agent bean, that is triggered depending on that processes' start event:

- *None* start event: the workflow is executed exactly once when the agent is started.
- *Timer* start event: the workflow is executed at specific times, or in specific intervals
- *Message* start event with *MessageChannel* implementation: the workflow is executed when a message matching the template is received on the given message channel
- *Message* start event with *Service* implementation: the workflow method is exposed as a JIAC action, using the name given in the service description, and executed whenever that service is invoked.

More details about the transformation can be found in [3].

3.4.2 Using the Interpreter

To use the VSDT interpreter agent, you have to add a dependency to your project, as shown in Listing 3.11.

Listing 3.11: VSDT Interpreter dependency

```
<dependency>
    <groupId>de.dailab.jiactng.basicServices</groupId>
    <artifactId>vsdtInterpreter</artifactId>
    <version>${jiac.version}</version>
</dependency>
```

Now you can add the `ProcessEngineBean` to one of your agents. Optionally, you can also specify a number of VSDT diagram files to be loaded as soon as the agent starts. In this case, you have to provide the exact name of the participant whose processes the agent is supposed to execute, and, separated with an `@` sign, the path to the `.vsdtd` file, relative to the classpath (Listing 3.12).

Listing 3.12: VSDT Interpreter Configuration

```
<bean parent="ProcessEngineBean">
    <property name="executionInterval" value="200" />
    <property name="showUI" value="true" />
    <property name="vsdtFiles">
```

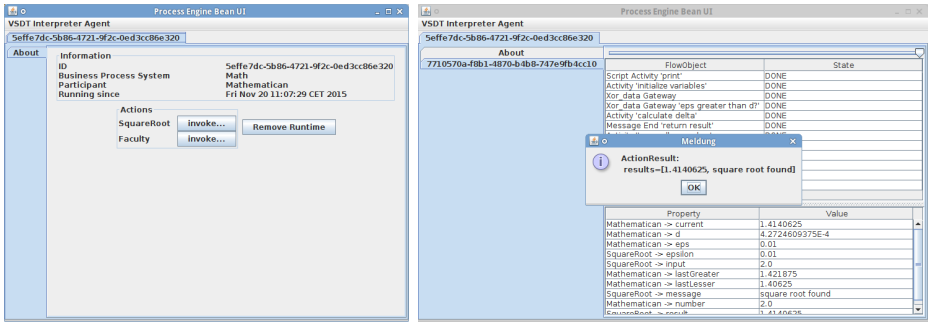


Figure 3.3: Interpreter UI. Left: Runtime tab; Right: Instance tab.

```
        <list>
            <value>Name of Role@/classpath/relative/path/to/file.vsdtd</value>
        </list>
    </property>
</bean>
```

The interpreter provides a number of actions for managing the deployed processes:

- `ACTION_REGISTER_RUNTIME` to register a new interpreter runtime, given the content of the VSDT file, as a string, and the name of the participant; returns the ID of the newly created runtime
- `ACTION_REMOVE_RUNTIME` remove the runtime identified by the given ID
- `ACTION_GET_RUNTIMES` return currently installed interpreter runtimes, as a dictionary mapping IDs to short descriptions

Besides those management actions, the interpreter agent will also expose any actions defined by the installed interpreter runtimes, corresponding to the workflows that have a message start event with service-implementation. For others start events, the interpreter bean registers a number of observers, triggering those processes accordingly.

In each step of the agent's execution cycle, each of the currently executed processes are advanced by one step (i.e. checking one event, or executing one task, etc.). That means that that bean's execution interval will determine how fast or how slow the processes are executed.

Besides the above mentioned actions, another way to interact with the interpreter bean is using a simple UI, as shown in figure 3.3, that can be activated by setting the `showUI` property in the configuration. The default behaviour is to show the UI, but it can safely be closed without stopping the agent, and will not show on a system in "headless" mode, e.g. on a pure server.

The UI can be used to deploy new interpreter runtimes to the process engine bean, by selecting the according files and one of the available participants, to list and remove existing runtimes, and to trigger actions provided by those runtimes[6]. Also, for each

[6] Actions can only be triggered from the interpreter UI if all parameters are primitive, i.e. Strings, numbers, etc., or if those parameters can be supplied in a simple MVEL expression.

runtime, the currently running instances are listed in individual tabs, each showing the current state of each flow object in the process, as well as the current value of each property. The instance tabs also provide a slider tools to "rewind" the interpreter-ui to previous steps, which is useful for debugging. Note that the "rewind" slider will only rewind the UI, not the actual state of the interpreter.

3.5 Agent Introspection

The `agentIntrospection` module can be used for adding a convenient introspection and debugging UI to arbitrary agents. Just add the dependency to your `pom.xml` file (see Listing 3.13).

Listing 3.13: Agent Introspection dependency

```
<dependency>
    <groupId>de.dailab.jiactng.basicServices</groupId>
    <artifactId>agentIntrospection</artifactId>
    <version>[5.2.0,5.2.99]</version>
</dependency>
```

You can then import the `Introspection.xml` file into your configuration and either add a dedicated `IntrospectionAgent` to the node, or add the `IntrospectionBean` to any existing agent (see Listing 3.14).

Listing 3.14: Introspection Agent and Bean

```
<import resource="classpath:de/dailab/jiactng/conf/Introspection.xml" />

<ref bean="IntrospectionAgent" /> <!-- add this to the node -->
<ref bean="IntrospectionBean" />  <!-- or this to any agent -->
```

Each introspection bean will display a simple Java Swing UI when the agent starts. This UI shown different tabs (see Figure 3.4).

- *Agent:* This tab shows basic information about the agent the introspection bean is attached to, such as its name, message box address, its own actions and beans, etc.
- *Actions:* This tab shows all the actions the agent can "see", both its own as those of other agents if they have the right action scope. The actions can be invoked, using MVEL[7] to parse and evaluate any parameter values.
- *Memory:* This tab shows the current content of the agent's memory, e.g. known actions, messages waiting to be processed, or anything else that has been added to (and not yet removed from) the agent's memory.
- *Memory History:* Complementary to the memory tab, this tab shows the history of memory events (after the *Activate* button has been clicked); such as messages being received, or DoActions being executed.

[7] https://github.com/mvel/mvel

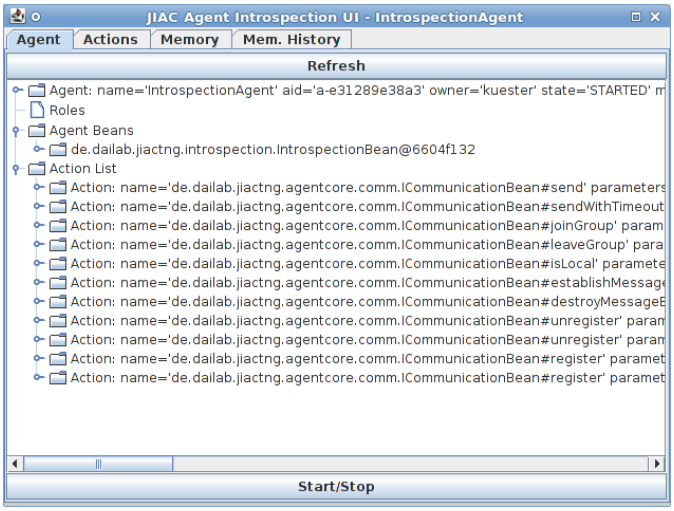


Figure 3.4: Introspection UI.

The introspection UI is very useful for both debugging and runtime monitoring, e.g. for manually invoking actions, for checking whether there are objects building up in the agent's memory (e.g. messages that have been received and processed, but never removed), or for understanding what's happening when sending and receiving messages or invoking actions.

Bibliography

[1] Benjamin Hirsch, Thomas Konnerth, and Axel Heßler. Merging agents and services – the JIAC agent platform. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Tools and Applications*, pages 159–185. Springer, 2009.

[2] Tobias Küster. *The Visual Service Design Tool, Version 1.4.5.* DAI-Labor, TU Berlin, November 2015. http://www.jiac.de.

[3] Tobias Küster, Axel Heßler, and Sahin Albayrak. Towards process-oriented modelling and creation of multi-agent systems. In Fabiano Dalpiaz, Jürgen Dix, and M. Birna van Riemsdijk, editors, *Engineering Multi-Agent Systems*, volume 8758 of *LNAI*, pages 163–180. Springer International Publishing, 2014.

[4] Tobias Küster, Marco Lützenberger, and Sahin Albayrak. A formal description of a mapping from business processes to agents. In Matteo Baldoni, Luciano Baresi, and Mehdi Dastani, editors, *Engineering Multi-Agent Systems*, volume 9318 of *LNAI*, pages 153–170. Springer International Publishing, 2015.

[5] Marco Lützenberger, Thomas Konnerth, and Tobias Küster. Programming of multiagent applications with JIAC. In Paulo Leitão and Stamatis Karnouskos, editors, *Industrial Agents – Emerging Applications of Software Agents in Industry*, pages 381–400. Elsevier, 2015.

[6] Marco Lützenberger, Tobias Küster, Thomas Konnerth, Alexander Thiele, Nils Masuch, Axel Heßler, Jan Keiser, Michael Burkhardt, Silvan Kaiser, Jakob Tonn, Michael Kaisers, and Sahin Albayrak. A multi-agent approach to professional software engineering. In Massimo Cossentino, Amal El Fallah Seghrouchni, and Michael Winikoff, editors, *Engineering Multi-Agent Systems*, volume 8245 of *LNAI*, pages 156–175. Springer Berlin Heidelberg, 2013.