

JIAC — Java Intelligent Agent Componentware

**Version 5.1.0
Manual**

CC ACT
DAI-Labor, TU Berlin

2010 / 06

Contents

1	Introduction	3
1.1	What this manual is about	3
1.2	Introduction to JIAC	3
1.2.1	Why JIAC?	4
1.3	Acquiring JIAC	4
1.3.1	Java	4
1.3.2	Apache Maven	4
1.3.3	Eclipse	5
1.3.4	Example Project	5
2	JIAC Programming Basics	7
2.1	A First Example	8
2.1.1	Hello World	8
2.1.2	Ping Pong: The MAS Hello World	9
2.2	Agent Configuration	13
2.3	Agent Beans	14
2.3.1	The Lifecycle	14
2.3.2	The Execution Cycle	15
2.4	Actions	15
2.4.1	Using Actions	15
2.4.2	Providing Actions	17
2.5	Agent Memory	17
2.5.1	memory.write()	18
2.5.2	memory.read()	18
2.5.3	memory.remove()	19
2.5.4	memory.update()	19
2.5.5	Space Events	20
2.5.6	Space Observer	20
2.6	Communication	21
2.6.1	CommunicationBean	21
2.6.2	MessageBroker	22

1 Introduction

1.1 What this manual is about

This manual is a pragmatical guide to agent development with 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 JADL, agent migration, agent management, and tools.

1.2 Introduction to JIAC

JIAC is a framework for developing Multi-Agent Systems and Services (MAS). The motivation for JIAC was:

- to ease the development of complex, distributed applications,
- to support system development in heterogenous 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 learnt a lot about making something easy for programmers (i.e. ourselves). The current version is a kind of middleware, build on the agent metaphor, to develop distributed applications, to integrate heterogenous environments and to integrate into different environments, that is manageable to a certain extend by humans.

The current Version of JIAC incorporates the following features:

- Spring-based component system
- ActiveMQ-based messaging

¹<http://www.multiagentcontest.org/>

- 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 you 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, preferably in Version 1.5 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 (and the dependencies' dependencies) and store them in the directory `.m2/` 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.

1.3.3 Eclipse

We recommend using Eclipse² 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.

Note: When using Apache Maven together with the Eclipse Development Environment, we *strongly* recommend using the Eclipse plugin for Maven, and not the Maven plugin for Eclipse.³ With it you can run `mvn eclipse:eclipse` in your project directory and Maven will automatically generate Eclipse's `.project` and `.classpath` files from Maven's `pom.xml`.

1.3.4 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
- 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
2   /2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM
3   /4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4
5   <modelVersion>4.0.0</modelVersion>
6
7   <groupId>de.dailab.jiactng.examples.manual</groupId>
8   <artifactId>helloworld</artifactId>
9   <version>1.0.0-SNAPSHOT</version>
10  <packaging>jar</packaging>
11  <name>JIAC V Hello World Example</name>
12
13  <dependencies>
14    <dependency>
15      <groupId>de.dailab.jiactng</groupId>
16      <artifactId>agentCore</artifactId>
17      <version>5.1.0</version>
18    </dependency>
19    <!-- more dependencies -->
```

²<http://www.eclipse.org>

³In case this sentence was too confusing: <http://maven.apache.org/eclipse-plugin.html>

```

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/<
25       /url>
26     <snapshots>
27       <enabled>>false</enabled>
28     </snapshots>
29   </repository>
30   <!-- more repositories -->
31 </repositories>
32
33 <build>
34   <plugins>
35     <plugin>
36       <groupId>org.apache.maven.plugins</groupId>
37       <artifactId>maven-compiler-plugin</artifactId>
38       <configuration>
39         <source>1.6</source>
40         <target>1.6</target>
41       </configuration>
42     </plugin>
43   </plugins>
44 </build>
45 </project>

```

After some header information, the JIAC module “agentCore” is listed as the only dependency. 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`:

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

The build block 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

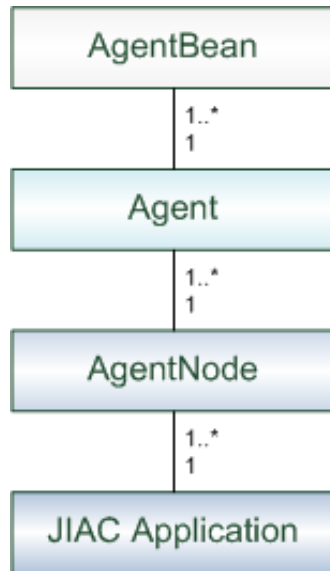


Figure 2.1: JIAC basic concepts and their structural relationships

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*.

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 lifecycle, 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 lifecycle.

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, so-called `AgentBean`, defines not the entire agent, but just one aspect of it: a behaviour, 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):

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 `executeInterval` 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):

Listing 2.2: Hello World Agent Configuration

```
1 <beans>
2
3 <import resource="classpath:de/dailab/jiactng/agentcore/conf/AgentNode.xml" />
4 <import resource="classpath:de/dailab/jiactng/agentcore/conf/Agent.xml" />
5
6 <bean name="HelloWorldNode" parent="Node">
7     <property name="agents">
8         <list>
9             <ref bean="HelloWorldAgent" />
10        </list>
11    </property>
12 </bean>
13
14 <bean name="HelloWorldAgent" parent="SimpleAgent" singleton="false">
15     <property name="agentBeans">
16         <list>
17             <ref bean="HelloWorldBean" />
18        </list>
19    </property>
20 </bean>
```

```

21
22 <bean name="HelloWorldBean" class="examples.helloworld.HelloWorldBean" singleton
    ="false">
23   <property name="executeInterval" value="1000" />
24 </bean>
25
26 </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(path/
   to/configFile).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's 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 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/
   dtd/spring-beans.dtd">
3
4 <beans>
5
6   <import resource="classpath:de/dailab/jiactng/agentcore/conf/AgentNode.xml" />
7   <import resource="classpath:de/dailab/jiactng/agentcore/conf/Agent.xml" />
8
9   <bean name="PingPongNode" parent="NodeWithJMX">
10    <property name="agents">
11      <list>
12        <ref bean="PingAgent" />
13        <ref bean="PongAgent" />
14      </list>
15    </property>
16  </bean>
17
18  <bean name="PingAgent" parent="SimpleAgent" singleton="false">
19    <property name="agentBeans">

```

```

20     <list>
21         <ref bean="CommunicationBean" />
22         <ref bean="PingBean" />
23     </list>
24 </property>
25 </bean>
26
27 <bean name="PongAgent" parent="SimpleAgent" singleton="false">
28     <property name="agentBeans">
29         <list>
30             <ref bean="CommunicationBean" />
31             <ref bean="PongBean" />
32         </list>
33     </property>
34 </bean>
35
36 <bean name="PingBean" class="examples.pingpong.PingBean" singleton="false">
37     <property name="executeInterval" value="1000" />
38     <property name="logLevel" value="INFO" />
39 </bean>
40
41 <bean name="PongBean" class="examples.pingpong.PongBean" singleton="false">
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, so equip each agent with a `CommunicationBean`, next to the `PingBean` and the `PongBean`, implementing the behaviour 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 public class PingBean extends AbstractAgentBean {
2
3     private Action sendAction = null;
4
5     @Override
6     public void doStart() throws Exception {
7         super.doStart();
8         log.info("PingAgent - starting...");
9         log.info("PingAgent - my ID: " + this.thisAgent.getId());
10        log.info("PingAgent - my Name: " + this.thisAgent.getName());
11        log.info("PingAgent - my Node: " + this.thisAgent.getNode().getName());
12
13        // Retrieve send action provided by CommunicationBean
14        sendAction = retrieveAction(ICommunicationBean.ACTION.SEND);
15
16        // If no send action is available, check your agent configuration.
17        // CommunicationBean is needed
18        if (sendAction == null)
19            throw new RuntimeException("Send action not found.");
20    }
21
22    @Override
23    public void execute() {
24
25        // Retrieve all Pong Agents

```

```

26     List<IAgentDescription> agentDescriptions = thisAgent.searchAllAgents(new
        AgentDescription());
27     for (IAgentDescription agent : agentDescriptions) {
28         if (agent.getName().equals("PongAgent")) {
29
30             // Send a 'Ping' to each of the PongAgents
31             JiacMessage message = new JiacMessage(new Ping("ping"));
32             IMessageBoxAddress receiver = agent.getMessageBoxAddress();
33
34             // Invoke sendAction
35             log.info("PingAgent - sending Ping to: " + receiver);
36             invoke(sendAction, new Serializable[] { message, receiver });
37         }
38     }
39 }
40 }

```

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 which's payload is a `Ping`, 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 (AgentBeans): *memory* and *execution* (Listing 2.8). Both parts are essential for making an agent.

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 factbase 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 perceps 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

```
1 <bean name="CowboyAgent" parent="SimpleAgent" singleton="false">
```

```

2     <property name="agentBeans">
3         <list>
4             <ref bean="ServerCommBean" />
5             <ref bean="PerceptionBean" />
6             <ref bean="ExplorerBean" />
7             <ref bean="DriverBean" />
8             <ref bean="ThiefBean" />
9             <ref bean="CommunicationBean" />
10        </list>
11    </property>
12 </bean>

```

See the following chapter 2.3 for more details on AgentBeans.

2.3 Agent Beans

The usual way to extend agents with new behaviours and capabilities is to create an Agent Bean that offers the desired functionality. All Agent Beans need to implement certain interfaces for lifecycle- 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 Lifecycle, or periodically, depending on its Execution Cycle.

2.3.1 The Lifecycle

The Lifecycle (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 Lifecycle in accordance to the agent's Lifecycle. The interface declares methods such as `init()`, `start()`, `stop()`, and `cleanup()`, for changing between Lifecycle 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 Lifecycle state, like looking up needed Actions, connecting to some data base and other initialization and/or finalization work.

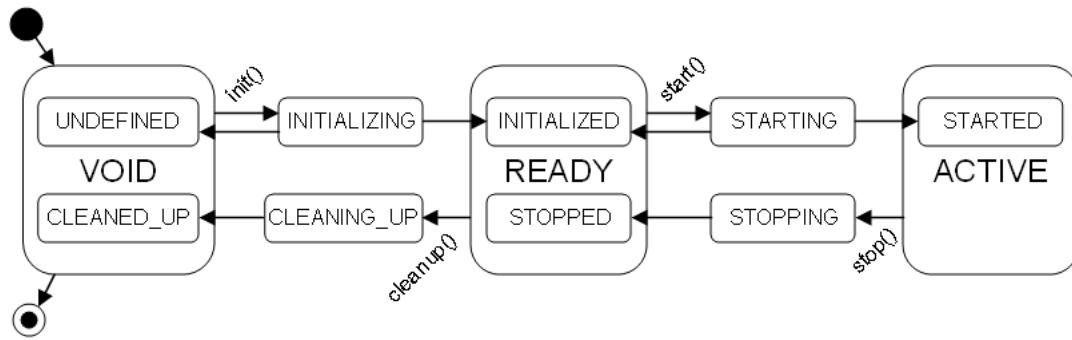


Figure 2.2: Lifecycle

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 Agent 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 milli seconds) 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" singleton="false">
2   <property name="executeInterval" 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.

2.4.1 Using Actions

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

- find the 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. However, here we have used only the first two steps: finding and invoking the action.

Listing 2.11: Invoking an Action

```

1 Action sendAction= retrieveAction(ICommunicationBean.ACTION_SEND);
2 [...]
3 invoke(sendAction, new Serializable [] {message, receiver});

```

Both `retrieveAction()` and `invoke()` are convenience methods provided by the class `AbstractAgentBean`, and there are a few more, which we will not discuss here in detail. In most cases you can resort to these methods, but for gaining an understanding of how JIAC works it is important to know what the above code *really* does:

Listing 2.12: Invoking an Action (the long way)

```

1 // retrieve action
2 Action template= new Action(ICommunicationBean.ACTION_SEND);
3 Action act= memory.read(template);
4
5 // invoke
6 DoAction doAct= act.createDoAction(new Serializable [] {message, receiver}, null);
7 memory.write(doAct);

```

All the actions known to the agent are represented by an Action description in the agent's memory, so we can retrieve them from there using a template with the desired action's name, which is exactly what `retrieveAction` does. We will explain the agent's memory in more detail in Section 2.5. Then, a `DoAction` object is created from the action description with the given parameters and written to the agent's memory. While the Action object can be seen as the agents *knowledge* of that action, the `DoAction` represents the *intention* to perform this action.

Actions are invoked asynchronously. The agent's Execution Cycle, which is also responsible for executing the Agent Beans' `execute()` methods, periodically checks the agent's memory for `DoAction` objects and will eventually perform the respective actions.

This leads us to the third step: How do we get the result from the action invocation? You might have noticed that there is another parameter in the call to `createDoAction()`. Here you can provide a `ResultReceiver`, which's method `receiveResult` will be called when the action has been performed, making available the result of the action invocation. If you want to call the Actions synchronously, you can do so by calling `invokeAndWaitForResult` instead of `invoke`.

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 will deadlock.

2.4.2 Providing Actions

We do 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 an `ArrayList` 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 2.13: Exposing an Action

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

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 *JAC 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 2.13. 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.

Additionally 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`.

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.

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). 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 }
```

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(Boolean.TRUE, 2, 2));
2  memory.write(new Field(Boolean.TRUE, 2, 3));
3  memory.write(new Field(Boolean.FALSE, 4, 2));
4  memory.write(new Field(Boolean.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, we can say that the call should return only when an object is in the memory that matches the template 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):

2.5.4 `memory.update()`

To `update()` certain facts in memory, we give a template that will match the interesting objects and then another pattern that says what to change. In the example, our digger

Listing 2.19: removeAll() from memory

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

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 `SpaceEvents` 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` – some objects have 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)

If you want the `SpaceObserver` only get notified on certain objects and their changes you can use a second `attach` method that has a second 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.

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

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);

```

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

```

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

```

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 for all agents on his node and between nodes.

2.6.1 CommunicationBean

The first thing you do is to add a *CommunicationBean* in the configuration file to those agents that want to talk to other agents (Listing 2.23).

Listing 2.23: Add a CommunicationBean

```

1 <bean name="MyAgent" parent="SimpleAgent" singleton="false">
2     <property name="agentBeans">
3         <list>
4             <ref bean="CommunicationBean" />
5             ...
6         </list>
7     </property>
8 </bean>

```

The *CommunicationBean* registers the *IMessageBoxAddress* of the agent at the broker so you are able to send direct messages to that agent. Furthermore, the *CommunicationBean* provides some actions that can be used by the agent that has the *CommunicationBean* been added:

- **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 message channel for multicast 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`. Here is again what we did there:

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

2.6.2 MessageBroker

By default, all `AgentNodes` in a IP subnet find each other and all 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 `agentnode` configuration file as shown in Listing 2.24.

Listing 2.24: Change `AgentNode` Configuration to separate your `agentnodes` and agents

```

1  <bean name="MyBroker" parent="ActiveMQBroker" singleton="true"
2  lazy-init="true">
3      <property name="connectors">
4          <set>
5              <ref bean="MyTCPConnector" />
6          </set>
7      </property>
8  </bean>
9
10 <bean name="MyTCPConnector" parent="ActiveMQTransportConnector"
11 singleton="true" lazy-init="true">
12     <property name="transportURI" value="tcp://0.0.0.0:0" />
13     <property name="discoveryURI" value="smartmulticast://default?group=myChannel"
14     /> </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 doesn't matter how the broker is configured on the user side. Just change the configuration of one node to be the master (Listing 2.25) and the other nodes to be the slaves (Listing 2.26).

Listing 2.25: Change AgentNode Configuration to separate your agentnodes and agents

```
1 <bean name="StaticMasterConnector"  
2   class="de.dailab.jiactng.agentcore.comm.broker.ActiveMQTransportConnector"  
3   singleton="false">  
4   <property name="transportURI" value="tcp://0.0.0.0:6789" />  
5 </bean>
```

Listing 2.26: Change AgentNode Configuration to separate your agentnodes and agents

```
1 <bean name="StaticSlaveConnector"  
2   class="de.dailab.jiactng.agentcore.comm.broker.ActiveMQTransportConnector"  
3   singleton="false">  
4   <property name="networkURI" value="static:(tcp://master.I.P.number:6789)" />  
5   <property name="duplex" value="true" /> <property name="networkTTL" value="255"  
6     " />  
7   <property name="transportURI" value="tcp://0.0.0.0:0" />  
8 </bean>
```