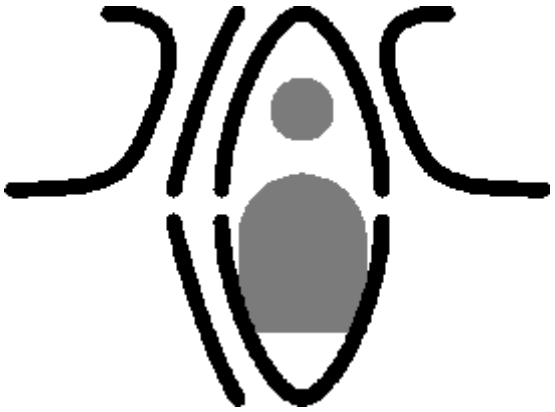
JIAC Tutorial 2



Dipl. Inform. Thomas Konnerth,
Cand. Inform. Robert Woll

September 2, 2003

Contents

1. Introduction

1.1 JIAC

JIAC stands for JAVA Intelligent Agent Componentware. More exactly JIAC is a framework allowing you to easily develop a multi-agent-system. In this tutorial we presume some knowledge about multi-agent-systems and their areas of application. If you haven't got a clue of what multi-agent-systems are all about we do strongly recommend you to visit the internet first and make yourself an idea of it. Though, even if you don't really know a lot about multi-agent-systems you will hopefully be able to understand most of this tutorial.

As its name already says JIAC is entirely written in the JAVA programming language. Though, if you want to use JIAC you will have to learn JADL first.

JADL is the JIAC Agent Description Language and it is based on first order logic. While this may trouble some of you in the beginning, the intention of this tutorial is to prove that JADL isn't actually that hard to understand. We do even hope to enthuse you a little bit for this slightly different way of programming. In the end JADL is quite elegant and it can save you a lot of time compared to the usage of JAVA.

1.2 A multi-agent-system scenario

There are quite a lot of areas of application for a multi-agent-system. You can use agents in production control systems, suppy-chain-management, search-engines, social simulation, etc...

Actually, agents seem to be an interesting approach for all kinds of distributed applications with a strong need for scalability and flexibility. Though, you might state that this kind of problems has already been dealt with for a long time in the field of distributed systems, you will see that using an ready-to-go multi-agent-system framework may save you a lot of time when developing such a distributed system. Furthermore agents are much more than just parts of a distributed system: agents are autonomous, they may act towards goals by planning and scheduling and they shall be able to solve certain kinds of unforeseen problems.

As a very popular field of interest, we will focus on an e-commerce scenario in the following tutorial. We will create a platform on which sellers, producers and customers may trade diverse products. With each of them playing a certain role and having certain goals, we do think that agents do perfectly fit into such a domain.

1.3 How do we proceed?

In the following chapters we will introduce you into the very basic concepts of JIAC and especially JADL. After a short digression to the system requirements, we will directly see how we can get our first agent running. In order to build our desired trading platform, we will also need to describe the field of interest, the scenarios domain. For this purpose we will see how to write an ontology, which is JIACs way do describe common concepts agents may talk and reason about. Having written an ontology, we start to describe the actions that our agents may take within the application. This will finally be the last step in this tutorial. If you get there, you shall be able to write your first agent-system application in JIAC.

Enjoy!

2. Configuring your system

2.1 System requirements

This documentation asumes, that you use a JIAC-Release based on *JIAC 4.3* (June 2003) or newer.

The above release has been tested with *Java TM 2 JDK 1.4.1_02* on *Windows XP*, *SunOS 5.7* and Linux (*Red Hat 8.0*) and requires at least *Java TM 2 JDK 1.4.0* (not tested or supported).

If you want to use the full functionality, i.e running distributed marketplaces on different computers, you must have access to a *Netscape LDAP-Server* (Version 1.2).

2.2 Installing JIAC

Of course, running JIAC requires that you have an appropriate version of the Java-Runtime-Environment installed. If you want to develop applications with JIAC, you will need a full developers kit for java.

The installation of JIAC itself is quite simple. We recomend that you get the starter-pack from: `http://euklid.cs.tu-berlin.de/jiac-soft/pub/jiac43/4.3.0/`. Simply unpack this file into a directory of your choice. When you do this, make sure that you extract the Zip-file with full path-names, so that all files are in the right place.

Afterwards, you have to set the `JAVA_HOME` environment variable, if it is not already set. This variable should contain the path of your Java-directory. The standard value for this under a Windows-system would be `C:\j2sdk1.4.1_02`
Note: This variable must not point to the `j2sdk1.4.1_02\bin` directory, but to the directory above.

2.3 Configuring the Framework

One of the most important configurations for JIAC is that of the LDAP-Server. If you want to use an LDAP-Server, you have to tell JIAC where to find that server.

This is done, by setting the appropriate properties for all platform managers. To do so, you can either edit the .platform files of all managers, or you can create a file called *jiac.agent*. This file holds global properties, that are overwritten by the local properties of individual platforms. So you can use this file for a simple default-configuration of all platforms.

The location for this file depends on your system. For Windows-Users, this file should be placed in the folder that is used by windows for your personal settings. On a german Windows XP-System that would be `C:\Dokumente und Einstellungen\`*username*. If you are running a linux-system, the file would have to be placed im your home-directory.

Getting back to the configuration of the LDAP - the two properties you have to set are:

`de.dailab.jiac.component.DFProviderBean.ldapHost=`*hostname*
`de.dailab.jiac.component.DFProviderBean.ldapPort=`*TCP-port*

If you want to use some special LDAP-server, you may need to set the password for that server, using:

`de.dailab.jiac.component.DFProviderBean.rootPW=`*password*

For further information on this, please consult the JIAC-release site.

2.4 Files

You are going to create and modify a couple of files during the course of this tutorial. Here is a short list of files and their contents for you:

*.onto	Ontologies, Functions, Comparisions
*.jadl	Actions, Services, Protocolls, Objects, cond, send, receive
*.goal	Goals
*.platform, *.agent	Properties
*.java	AgentBeans, Role-Interfaces

We recomend, that you put *Objects* and *Goals* into their own files (one for each). It's also helpfull to put a *Service*, and each side of the *Protocoll* into seperate files (that's three files). The *cond-*, *send-* and *receive*-planelements should be in the same files, as the Actions/Protocolls that use them.

2.5 Documentation

You may find further information about JIAC on our homepage:

`http://euklid.cs.tu-berlin.de/jiacportal/`

This includes another Tutorial, a Programmers Guide and of course the JavaDoc for the JIAC-API, which you will definetly need.

3. The basic JIAC-Agent

In this chapter we will get an agent running on a platform. Referring to our scenario let's say the agent represents a company which is producing some goods. Thus, we will simply call our agent "producer". For the beginning our producer-agent won't have any special problem-oriented functionality but it will be fully functional in terms of a JIAC-agent, and that will be our motivation for this first chapter.

3.1 Parts of an Agent

One of JIAC's most important characteristics is that it is component-based, i.e. each agent is just a set of components which are put together. To give you an idea, we will give you a short introduction into different types of components, including:

- agentbeans (Java-classes that are similar to JAVA-Beans)
- planelements (actions and services)
- ontologies (domain descriptions)

3.1.1 Agentbeans

All components that are implemented in JAVA are called *agentbeans* [1] within JIAC. All agentbeans share a common interface which enables them to receive and send messages between each other. Furthermore, each agentbean has its own life-cycle management.

The strength of this approach is the ease of use when finding a solution to a specific problem. By putting only those components together which are really needed you may avoid overhead and - which is much more important - the well-defined interfaces for a component allow you to easily write your own Components and plug them into an agent.

[1] Even though these *agentbeans* are similiar to JAVA-Beans, they are not quite the same as they support a more specialized set of functions.

Essentially each agent needs some mandatory components. Its functionality may then be extended by adding as many other Beans as needed. We won't present to you all components provided by JIAC at this place, but just the mandatory components you really need to have heard about for the beginning. You can group them in three blocks, which are:

- The most essential Beans affecting only the agent itself:
  - FactBean - Holds all objects known to the agent (global memory)
  - GoalBean - Manages the goals an agent has
  - SelectionBean - Selects actions that may fulfill active goals
  - SchedulerBean - Manages the execution-order of selected actions
  - ExecutionBean - Interprets the instructions that make up actions
- The Beans for communication with other agents:
  - CommunicationBean - Manages communication with other JIAC-agents
  - TCPIPCommunicationBean - Allows communication via TCP/IP
  - JVMCommunicationBean - Allows communication via the local JVM
- The Beans needed for setting up a multi-agent-system
  - AMSUserBean - Manages interactions with the platform-manager
  - CommunityBean - Manages interactions with other agents
  - AMSProviderBean (only mandatory for platform-agents - FIPA-AMS)
  - DFProviderBean (only mandatory for platform-agents - FIPA-DF)

Finally there is the DebuggerBean which actually isn't really a mandatory Bean. It is only used for testing purposes. This is why it is usually included in the initial state of the agent configuration.

3.1.2 Plans & Services

Plans describe actions agents may achieve. Without any plan an agent couldn't do anything. In our fourth chapter you will learn how to write such an action. For this time all you need to know is that there are actions the agent achieves on its own and that there are actions which assume an interaction with another agent.

Actions between two agents are called services and play an essential role in the definition of what an agent theoretically is: an agent is an autonomous entity that interacts with other agents via some communication.

In JIAC communication is directly mapped to services.

So if there weren't any services for an agent to use, it wouldn't be an agent anymore. As the agent we want to implement in this first example runs on a platform, there is some mandatory interaction with another agent: the platform-agent.

Before we go further, remember the following: A service is an ”inter-action” between two agents where one agent plays the role of a service-user and the other agent that of a service-provider.

A service needs to be described by three plans: one describing the service itself, one for the user-part and one for the provider-part.

The service-plan must be known by both agents the two other protocol-plans must only be known by the agent playing the corresponding role, i.e. the user-protocol respectively the provider-protocol.

3.1.3 Ontologies & Objects

Something essential to the notion of agency are ontologies. Agents communicate through a common language - thus using a common syntax. But what they also need to agree on the semantics.

If you want two people to speak about the same thing using the same word, they need to have the same concept of what this word means.

An ontology formally describes such concepts. Thus, two agents both knowing the same ontology may communicate using instances of their common concepts. These instances are called objects. You will learn more about ontologies in the next chapter.

Something very important to understand is that the factbase *of an agent is its only work space. The factbase holds all objects, as is something like the global memory of an agent. Only changes that are made on objects in the factbase are relevant to the agent as a whole.*

For ontologies this means: if you don’t have an ontology, you can’t create objects, thus the fact base is empty, thus the agent is unable to do anything! This is to say that every agent needs ontologies and every agent needs objects to work on. These objects may get into the factbase while the agent is running or when it gets started. In the next chapter you will learn how to create objects.

3.2 Configuration of an Agent

Generally, if you want to run an JIAC agent, all you need is a platform configuration file for the platform the agent shall run on and the configuration file for the agent itself. Such a configuration file has the postfix ”.agent” respectively ”.platform”. In JIAC such a ”platform” is an agent itself. Thus, for our purpose of simply getting

an agent running we do just need to know how to configure an agent and write one agent configuration for the platform and one for the agent.

Generally, an agent configuration file contains the following information:

- some general agent settings (permissions, mobility, initialization state)
- agentbeans
- plans
- services
- intial content of the factbase
- debugg settings
- platform settings (only for platform-agents)

You can find a template for an agent configuration file in the appendix: 9.1.1

3.3 Running the Agent

3.3.1 Writing the configuration file

In reference to the configuration template we do now create the two configuration files we need for our example. We will name these files ”trading.platform” and ”producer.agent”. Please take a look at their code which is located in the appendix: 9.2

Most things should be clear when comparing these files to the given template, though you may need a few explanations which we try to give here:

- File paths: As you can see you need to set a relative path for all files included within the configuration file. These paths are always the same because they refer to package names and the build system of JIAC cares for putting all these paths into the classpath.
- Plans and Services: As you know by now, a service is a just a special plan. Nevertheless: in the configuration file you need to define services and plans separately. There's a simple rule for how to handle this separation. Put all plans an agent needs - including the services - into the plan library and if the agent is a provider for a service, you have to put the service-plan into the service library, too. Thus, a service-plan of an agent being a provider for this service needs two entries in the configuration file.
- Ontologies: Taking a glance at the configuration template you will state that there are no ontologies noted. This is the case, because all actions an agent may perform depend on ontologies and thus do already refer to the ontologies they need. Therefore all ontologies are referred to indirectly and don't need to be written to the configuration file.

3.3.2 Putting it all together

Now, as already mentioned in chapter 1.2. ("Installing JIAC") you should put all files (in this case: the ".agent" and the ".platform" file) into the JIAC project template folder (extracted JIAC Starter Pack). This folder contains four subfolders:

- build (the build of your project will appear somewhere here)
- core (your projects core-files)
- thirdparty (thirdparty-libraries you may need - includin JIAC)
- tools (tools that are needed for the build-process)

All the non-agent files you write when programming a multi-agent-system have to be put into the folder "core/src". Your agent- and platform-files do belong into the the "core/etc/conf" folder. The folder "build" will be important for compiling and running the application. The two remaining may be negleged for now.

When putting your agent- and platform-files into the conf-folder, you will find two template-files in it. As these template files are actually nearly the same than those we've just presented to you, there's no need to replace them. But it would make sense to modifiy them just a little bit by changing two entries within the platform-file:

Changes for the platform-file (*core.platform*):

```
de.dailab.jiac.component.AMSProviderBean.name=core
```

should be replaced by

```
de.dailab.jiac.component.AMSProviderBean.name=Trading
```

This will change the platform's name appearing on your screen when starting. Further, replace:

```
de.dailab.jiac.component.AMSProviderBean.agents=\
    Basic

de.dailab.jiac.component.AMSProviderBean.agentsPropFile.Basic=\
    basic.agent
```

by the following code:

```
de.dailab.jiac.component.AMSProviderBean.agents=\
    Producer

de.dailab.jiac.component.AMSProviderBean.agentsPropFile.Producer=\
    basic.agent
```

This will change the agent's name appearing on your screen when starting the platform.

We strongly recommend that you do not put all of your non-agent files into the folder "core/src", but that you use some folder structure. Of course, in larger multi-agent-systems it does make sense to sort your files according to their file type. In most JIAC projects you can find the following subfolders:

- ontology - folder for all ontologies
- components - folder for all new agentbeans, each of them refering to one role
- role - folder for all self-written roles
- knowledge - folder for all plans, services and object files (with postfix ".jadl")

Usually the ".agent" and ".platform" files are located in the `/etc/conf` folder of your project. You may of course put them somewhere else, but then you will have to adapt the build-system.

The most important thing to pay attention to when structuring your files is to keep the package paths within the files consistent to your folder structure. You will see later on that all ontology files and all plans, services and object knowledge files define a package name at their beginning. The path of this package name is relative to the folder "core/src". Thus, if you've placed an ontology file within a subfolder "ontology" of the folder "core/src" the path within your ontology would just be "ontology". You'll see such an ontology in the next chapter.

Now you may run the "build"-file (".bat" respectively ".sh") located within the folder "build" of the project template. JIAC uses the *Jakarta-Ant* build-system for compiling projects because they will solve all inter-dependencies (mutual imports etc.) between classes and because it's rather inconveniant to compile everything on your own. Build-scripts are quite similar to what some of you might know as "make-files". Later on, it could be worth learning how to write such a script, but for the beginning it should be enough to use the build-script included within the starter pack.

After having started the build script, there will appear a new folder with subfolders in the "build" folder. This new folder is named *output*. It contains a folder for each project (*project-name-1.0.0 in our case*). Within this porject-folder, you will find */bin*, */conf* and */lib*, which contain the start-scripts, property files and the runtime-libraries. In the subfolder "bin", you will find a .bat (for Windows) and a .sh-file (for Linux) allowing you to start your platform.

3.4 Feedback from the Agent

Now that your platform should start, you will certainly wonder if everything works fine. There are three ways to find out:

- the platform monitor
- the console window
- the debugger GUI

As each of these three information sources is quite complex, we'll just touch on them a little bit.

3.4.1 The Monitor Tool

When the platform starts you will see a GUI on your screen showing you the platform and all the agents on it. If you point on one of these agents you will see its address. Further you have some access to the agent by being able to change its execution state and telling it to migrate to another platform.

Something much more important though is that the monitor temporarily draws connection lines between two agents every time there's a message passing. The appearance of these connection lines is not synchronized with the real speed of the the program, but the order in which the messages are delivered is correct. Also, there will appear a "failure" or a "success" on the monitor if a service has finished, indicating the result. Thus, if you've written an example within which two agents communicate using some fixed interaction protocol you will see if everything works fine by looking at the monitor tool.

Finally we must mention that the monitor tool is of course not really meant to be a debugging tool, and thus it's not the right tool to check your programs functionality. Though, for some very simple tests it may be useful.

3.4.2 Consolemessages

The console window may provide you a lot of information. As in most JAVA-applications, you may analyze the behaviour of your application by using outgoing system messages (System.out.println() in JAVA). All of these system messages appear in the console window and nowhere else. Therefore, if you want to use such messages, try to make them more easy to locate within all the other text by doing something like this:

System.out.println("##### "+yourLogMessage);

Of course, this is a *quick and dirty* solution, but it should be useful for the beginning.

3.4.3 DebuggerGUI

As we've included the DebuggerBean into our configuration files[2] there will be a debugging GUI showing up for each of the agents located on the platform. The debugging GUI is the best tool for checking your code. It may selectively log all the beans you want to get information about while running your application. Further, it may put an agent into "stepping mode", i.e. the agent does only do one step in its action script for each click you do on the GUI. This let's you track a failure or erronoeous behaviour in your application.

It would take too much time to explain all the debuggers functionalities at this place. Just notice:

> *Most failures can be found by logging the "ExecutionBean" or the "SelectionBean".*

Later on, we will explain to you why.

[2]The last agentbean in the `basic.agent` file.

4. Ontologies

An ontology is a formal description of a domain. Essentially ontologies aim at enabling programs to share knowledge through the use of a common domain description. In this section, we will give you a brief introduction to JIAC-Ontologies. For further information we recomend the JIAC-Programmers Guide.

4.1 The domain

Let's have a look at a little example: Imagine you want to describe some geometric figures, which is to say your domain would be "geometry". First you would have a look at what figures this domain includes: a rectangle, a triangle, a circle, etc. These are the elements of your domain. They are described through categories. Taking a closer look at one of these elements, you would then specifiy the characteristics describing each one of the categories. E.g. The two characteristics height, width would be the attributes of the category "rectangle". Finally you would ask yourself what operations are based on these categories? Within the domain "geometry" one may perform an operation which calculates the surface of a figure. Thus, you would define a function "calculateSurface".

In conclusion a domain is specified through categories existing within it and operations working on objects of these categories. An ontology describes a domain by formally defining its categories and functions.

In the following sections we will go back to our trading scenario and develop an ontology for the given domain.

4.2 Implementing an Ontology

In JIAC ontologies have the following form:

```
(package packageName

  (ont OntologyName:Version_Nr
```

```
    // categories

    // functions

  )

)
```

Note that usually an ontology is a component of a programming project. This is why this template indicates a package. As you can see further comments may be written as in the Java programming language. This is due to the fact that ontologies are transformed into java classes by the Ontology-Compiler. The filename of an ontology should look as follows:

ontologyName.onto

In our example we focus on a domain within which agents trade products. This is why we call our Ontology "Trading" and begin with the simple body of the ontology-file as shown above and save it to the file "trading.onto":

```
(package ontology

  // This ontology describes a domain for trading products.
  (ont Trading:Version_0

  )

)
```

4.2.1 Categories

The next step to take when writing an ontology is to define categories. A category has the following form:

```
(cat CategoryName

  (attributeName1 attributeType1)

  (attributeName2 attributeType2)

)
```

You can just put any number of categories into an ontology. Also the order is not important.

JADL provides the following basic types:

- int - note: though it's called "int" it's a long number in JAVA code

- real - note: ”real” is represented as a double number in JAVA code
- bool
- string
- agentname - an agents address like `jamesbond@secretplace.com:5007`
- url - note, representation as follows: `protocol://host:port/path`
- timestamp - represents a date or offset (`[+]DDMMYY`**`T`**`hhmmssmmm`)

When using these types or your very own written category objects always remember one very important thing:

> *If a variable in your JADL code has no concrete value bound to it, and you want to use it at some moment, you will get an constant called 'unknown'.*

This comes from JADL being based on a ternary logic which can also operate with unknown values. Whereas in imperative programming languages you may directly assign values to variables, JADL can't directly do this but uses predicates which - by resolution - bind values to variables. Therefore you might think of *unknown* as a special value a variable may have: if you think of boolean values e.g. remember: they haven't only got two, but three values in JADL: true, false and unknown.

Let's write a category for our trading-ontology and use some of the mentioned basic types: the category ”Product”. As we suppose an abstract product to be sufficiently described through the two attributes ”name” and ”price” we would write the following code:

```
(cat Product

  (name string)

  (price real)

)
```

Furthermore there are several keywords you may use together with an attribute. These are the three most common keywords you may encounter when reading JADL code:

- defined - indicates that this variable may be used as a unique key attribute
- needed - indicates that the value of this attribute must not be unbound, thus if you want to create an object of its category you must indicate a value for this attribute!

- fixed - indicates that the value of this attribute can't be changed once it has been set

Let's apply them on our "Product" as follows:

```
(cat Product

  (name string defined needed fixed)

  (price real needed)

)
```

It should make sense that the name of a product ought to be unique, fixed and especially exist.

Besides the keywords we've just presented to you, it is possible to set the initial and default value. For our example we will not give you a detailed explanantion, but just an example:

```
(cat Product

  (name string (init "bla") defined needed fixed)

  (price real (default 0.00) needed)

)
```

As you may guess, these options assign concrete values to the variables. If these values are not of a basic type you would have to use an object as concrete value. For further information, please refer to the programmers guide. Later on you will see how to write such an object.

Long things short. Let's have a look at the following category declarations:

```
(cat MultipleProduct

  (ext Product)

  (amount int needed)

)

(cat Offer

  (products Product[] needed)

)
```

```
(cat Stock

  (products MultipleProduct[])

  (full bool)

)
```

Each of the three categories shows you an interesting thing:

1. The category "MultipleProduct" extends the category "MultipleProduct" thus inherits all its attributes. (keyword `ext`)
2. The category "Offer" declares one of its attributes to be a list. As you can see, too, this list consists of objects of another category i.e. you may, of course, not only use basic types as attribute types but also complex types.

JADL provides exactly two data structures for collections: sets (represented by {}) and lists (represented by []). These may also be nested, i.e. you may construct a two-dimensional list of integers by declaring int[][], or a set of two-dimensinal lists using int{}{}.

4.2.2 Functions & Comparisons

Within a special domain you might identify some very common operations on objects. For ease of use JADL provides the possbility of writing such "functions" and "comparisons" within the code of an ontology itself.

A function may be assigned parameters and returns a value of some type. Whereas some programming languages such as C or Java may declare methods that do not return any value ("void" methods) JADL functions do always return a value of some type. If you want to write a function which doesn't need to return anything you should make use of dummy variables. Note that a function does not explicitly need any parameters, it may also be constant. A function in JADL looks like this:

```
(fun returnType functionName parameterType1 ...)

#begincode

  // within the begincode and the endcode tags you may write JAVA code

#endcode
```

The interesting thing about such a function is that you have to use JAVA code in order to define a function. Something which may seem a little strange in the

beginning are the types of objects used within this JAVA code. JIAC uses its very own JAVA classes for wrapping objects.

All objects of category types are wrapped into a so called KObjects.

In order to give you an impression of how this works in practice please study the following example:

```
(fun real getPrice Product)

#begincode

  // getting the parameter
  KObject product = (KObject)a0.getValue();

  // extracting the product's price
  double price = product.getValue(Product_price);

  return price;

#endcode
```

Here's the explanation:

1. In the first step the function accesses the given parameter. Whereas JAVA declares the parameters' names in the method header, JADL functions identify each parameter by its index.

a0 is the first parameter, a1 the second, etc.

   If the parameter is of a complex type like in our case (a "Product") you will have to cast it into a KObject or a KObjectRef. The difference between both will not be explained in detail here. Just remember:

KObjectRef always works for categories.

   If the parameter is not a complex type like `real`, `int`, `bool`, etc. it strictly depends on the type whether to cast the variable or not. A rule of thumb would be, that you don't have to cast the simple types you already know. `real`, `int`, `bool` and `string` give you primitive java-values[1]. But `agentname`, `URL` and `timestamp` are delivered as `KAgentName`, `KURL` and `KTimeStamp`.

[1] Remember that JADLs `int` is a `long` in JAVA, and JADLs `real` is a `double`.

2. In the second step we extract the value of the attribute "price" of the Product-object. As this might look a little complicated it's all easy to understand taking a look at what the Ontology-Compiler does. As already mentioned before the compiler makes JAVA classes out of the ontology files. More exactly each ontology will result in three JAVA classes:

   - One class for constructors of category objects each of them taking all "needed" attributes of this category as parameters.
   - One class collecting all the ontologies' functions.
   - One class collecting all the names of categories, attributes and functions.

   For our example with the ontology name Trading:Version_0 these would be the files:

   - Trading_Version_0.class
   - TradingFun_Version_0.class
   - TradingIfc_Version_0.class

   Note that the names of these classes do strictly depend on the ontology's inner name and not the name of the file the ontology has been saved to. Going back to our example you will notice that the attribute price of the category Product is called Product_price (and is a long number) within the JAVA code. It's an all easy pattern to remember:

   - CategoryName_attributeName

   If you want to extract a value, you have to use the method:

   - getValue(*CategoryName_attributeName*)

   if respectively you want to change it use this one:

   - setValue(*CategoryName_attributeName, newValue*)

3. In the third step we return the result value.

Now that we have given you an impression of how a function looks like we'll introduce you to the comparisons. Although a comparison might as well be seen as a function JADL makes a difference between both. Fortunately, once having understood how to write a function in JADL a comparison won't trouble you. A comparison has the following form:

```
(comp comparisonName parameterType1 ...)

#begincode

  // within the begincode and the endcode tags you may write JAVA code

#endcode
```

It's no surprise that a comparison doesn't declare the return type, because it's always a boolean value. Besides this, it may just be handled like a function. Here's an easy example:

```
(comp isCheaper Product Product)

#begincode

  // getting the first parameter
  KObject product1 = (KObject)a0.getValue();

  // getting the second parameter
  KObject product2 = (KObject)a1.getValue();

  // return the comparisons result
  return product1.getValue(Product_price)
          < product2.getValue(Product_price);

#endcode
```

Before going on to the next step notice that functions and comparisons may only be accessed from within JADL "actions" or JIAC-Beans which we will discuss later on. Therefore you may write a function or comparison in JADL code and directly include it into the "action" or write the JAVA code within a Bean. We recommend to declare functions or comparisons within an ontology if a function or comparison shall be accessible to many "actions". If only one or two "actions" shall use it, write JADL code and include it into the "action".

Agentbeans should only be used if you need to connect an agent to something that is 'outside' of JIAC, e.g. a foreign API, a human user, etc.

Now you can finally put all your code together and build your complete ontology file. We've put another category in the ontology which is called "Stock" and describes a producer's stock. You can find the ontology file in the appendix: 9.2.3

4.3 Creating an object

After having seen how to write an ontology you should also know how to instantiate objects of categories. Actually there are several possibilities to do this but we will only concentrate on JADL for this time (remember: you may as well create objects from within a Bean). Let's go and begin with another example:

We'll create objects for the following categories (from `Trading.onto`):

```
(cat Product
```

```
  (name string defined needed fixed)

  (price real needed)

)

(cat Offer

  (products Product[] needed)

)
```

Here are some objects (put them to `startingKnowledge.jadl`):

```
(obj notebook Product

  (name "IBM Thinkpad X30")

  (price 2500.00)

)

(obj sellersOffer Offer

  (products [Product:

      (obj Product

        (name "Apple IBook")

        (price 2000.00)

      )

      (obj Product

        (name "Acer Travelmate")

        (price 2400.00)

      )

    ]

  )

)
```

As you can see objects have identifiers. In our example we have the objects ”notebook” and ”sellersOffer”. As the identifiers are only important for the unique adress-

ing of objects in the factbase, all objects that are but attributes to other objects don't need identifiers. E.g. the objects within the product list of the "sellersOffer" have no identifiers.

> *Remember: you must define all "needed" attributes of a category in order to succesfully create an object!*

For the rest we suppose that this example is sufficiently clear and continue with the next important question: Where to put these objects and how to do it?

At this point you need some explanation: In JADL objects do only exist within the "Fact Base". Roughly put you could say that the fact base is an agent's memory. It's the location where it saves all information. Of course, the agent itself is nothing else than a (rather complex) set of JAVA objects within the systems memory, but you'll never touch it. Firstly because the management of the agents and their threads is rather complex, and secondly because you shall not access to all of these objects for security reasons. So the only objects we work on are those located in the fact base.

There are three ways to access the fact base.

1. First, of course, you may declare objects within a JADL "action", which is to say you may modifiy the fact base while an agent is running. There are three important statements in JADL to access the fact base which we will explain later on: you may read an object out of the fact base and respectively you may write an object to the fact base or update it.

2. The second way to modify the fact base while the agent is running is to access it through JAVA methods. As the fact base itself is a JAVA object it provides you several methods to access it's data. This is important if you wish to access the fact base from within another JIAC Bean.

3. Finally, the easiest way to modify the fact base is to write an object to a file and let the agent read this file's content into its fact base when initializing itself.

For the beginning we will use the third solution because it's the easiest way to add objects to the fact base. We do just take the code we've already written and add some file information as well as a "Stock" object to it. You can find the resulting code in the appendix: 9.2.4

As you can see, we've added a package name and the name of the ontology our objects refer to. Compared to the package name of the ontology in this example the name of the file is a part of it. This is true for all non-ontology files written in JADL. In this case what we've written is just a set of objects and the postfix for such files is ".jadl". Thus the name of the file is "startingKnowledge.jadl". If you add this file's package name to the agent configuration file at the right position, the agent's fact base will contain the two objects we've defined in our file. Just put the following lines into the agents' configuration file (`basic.agent`):

```
de.dailab.control.role.FactBaseRole.objects= \
  startingKnowledge.know
```

You will probably notice the fact, that the postfix of the file has changed from ".jadl" to ".know". This is - once again - due to the JADL-compiler. As the compiler creates java-classes from all ontology files, the objects you create refer to these classes. The compiler reads the JADL code out of the ".jadl"-file, creates object code and stores this code in a ".know"-file.

With this information we should be able to proceed to the next chapter and begin writing our first action.

5. The first Action

5.1 Implementing Actions

In this chapter we will see a simple example for an action. As you probably remember what we've said in chapter 3.1.2. actions are represented by so called plans. Such a plan consists of 4 essential parts:

- A set of variables which are used by the action described. All variables used by an action must be declared within this set
- A precondition which must be true before starting the action
- An effect the action has, more exactly a state which shall be true when finishing the action
- A description of how the action reaches the intended effect

Formally, such a plan has the following form in JADL:

```
(act actionName
    (var -set of variables-)
    (pre -set of preconditions-)
    (eff -set of effects-)
    (script -description of the action-)
)
```

5.1.1 Preconditions & Effects

You may have noticed that preconditions(`pre`) and effects(`eff`) are something unusual to most other programming languages. For JIAC they are essential! The reason for this is, that agents shall reach goals on their own and do therefore need to reason about everything they know and to plan actions they could take in order to reach their intended goals.

Imagine you want to eat something. What would you do? You would supposably think about what you could do in order to satisfy your hunger. If you are at home you would probably do the following: you would go to your fridge take something out of it and eat it. The question is: why would you do so? Well, if you want to eat something some preconditions must be met: you should have the food at reach, you should be able to open your mouth, you should have teeth, etc. Let's suppose your mouth is alright, still you would need to come at reach of your food. Therefore you would need to open the fridge. But as you are still in your room and not in the kitchen you would first have to go to the fridge.

What you see in this example is that once you've got a goal you want to reach, you would think backwards from the goal state on. By there you would try to find a way which leads you to the desired state. This is exactly what an agent would do. It would look at the goal state and if it is not already reached it would try to find an action which effects exactly this state. Now, if this action's preconditions aren't true at this moment it would find all actions effecting all those preconditions for this action which aren't true yet and so on.

As you see preconditions and effects are essential to solving a planning problem. Their content refers to objects in the factbase. Thus, if a precondition requires some attribute of some object to have a certain value and this object doesn't exist in the factbase, the precondition isn't met yet.

5.1.2 The Script

The script of an action describes how an action achieves its intended effect. A script is written in JADL and requires a good comprehension of JADL's first order logic. As functions from within ontologies may easily be used in a JADL script you may quickly begin to write your first action script by just calling an ontology function. For the beginning this is the easiest way to get an executable action.

Nevertheless, JADL is quite powerfull and most things you might want to do within an action, can be programmed in JADL. Though there are examples where JADL won't help you. E.g. if you want to use a GUI for your agent or the agent needs to use a database. In these cases you can't use a script for describing the action. You will have to write a new agentbean which has to be integrated into the agent by including it into its configuration file. If you want to use such a bean within an action you need a special action-type, that is not quite the same as a script. For now, scripts should be enough.

5.1.3 A little example

Let's go back to our e-commerce scenario and see what kind of action we could need now. We've decided that it would make sense to write an action for the producer agent which fills his stocks if they aren't full anymore. Let's call this action "FillStocks".

The first question when writing an action is always: what are the action's preconditions and what is its effect? To fill one's stocks is only reasonable for a producer agent if the stocks aren't full anymore. Therefore our precondition shall be: the stocks are not full. The effect of the action is that the stocks are full.

Now you should think at how to change the stocks and thereby find all the variables needed to do this. For now, we will cheat a little and don't really change the stocks but only simulate it by producing a system message telling us so. This should be a good beginning, as the easiest way for you to check the correctness of your agent application is to use system messages like "System.out.println()" in JAVA. Therefore it makes sense to see how this works. Let's go.

A short reminder of the category *Stock* we are going to use:

```
(cat Stock

    (products MultipleProduct[])

    (full bool)

)
```

Now, look at the following code example and don't be surprised: as we've already said JADL is based on first order logic and uses terms which may often be nested:

```
(act FillStocks
    (var ?stock:Stock)
    (pre (not (att full ?stock true)))
    (eff (att full ?stock true) )

    (script
      (seq
        (bind ?foo (fun printString "Hi!"))
      )
    )
  )
```

Put this code into a file named `FillStocks.jadl`. This file should be in the folder `/core/src/knowledge/`. You are going to work with this jadl-file for the rest of this chapter, and for the whole sixth chapter.

To add the action to your agent, you need to edit the `producer.agent`-file. Just add an entry to the property `de.dailab.control.role.PlanLibraryRole.plans`. The new property will look like this:

```
# Plans
  de.dailab.control.role.PlanLibraryRole.plans=\
    de/dailab/jiac/knowledge/APService.know \
    de/dailab/jiac/knowledge/APUser.know \
    de/dailab/jiac/knowledge/AMSService.know \
    de/dailab/jiac/knowledge/AMSUser.know \
    de/dailab/jiac/knowledge/DFService.know \
    de/dailab/jiac/knowledge/DFUser.know \
    knowledge/FillStocks
```

But now we should explain the action. It's not as hard as it looks at first sight. Let's go step by step:

- The variables: variables may be identified by a "?" at their name's beginning. You need to set their type within the variables field. If you need to define several variables you don't need to put colons between them - white spaces will do.
- The precondition: We've said our precondition is the following: the stocks are not full. This is exactly what we've described in this JADL code example. The inner term beginning with the keyword "att" describes the following: The attribute "full" of the object "?stock" is true. Every "att-term" has the form:

  ```
  (att attributeName object attributeValue)
  ```

  and may be read this way. As we wanted to presume that the stock is not full we do now nest the att-term into a not-term. This is all easy. A not-term has the following form:

  ```
  (not Term)
  ```

  As usual in first order logic, the negated term is *true*, if the original term was *false*. But as JADL uses a ternary logic, the original term might also be *unknown*. Note:

The negation of an 'unknown' Term is still unknown!

  For more details on JADLs logic, have a look at the programmers guide.
- The effect: once again we use the att-term as already described before.
- The script: in the script we use a term called "bind". A bind-term has the following form:

  ```
  (bind variable Term)
  ```

  A bind-term is JADL's way to assign a value to a variable. Our example can be read as follows: The variable "?foo" shall have the return-value of the function "printString" with the given parameter "Hi!". The function "printString", which is defined in our ontology, looks as follows:

  ```
  (fun bool printString string)
  #begincode
    System.out.println(a0);
    return true;
  #endcode
  ```

It's easy to see that a method printing a string to the console window doesn't actually need to return a value, but as you may remember JADL functions do always need to return a value of some kind. Therefore we do just use a dummy return value of the type bool and a corresponding dummy variable "?foo" to which we bind this return value. By the way: of course, you don't need to use a function in order to bind a value to a variable but you may also use a concrete variable like in the following example:

```
(bind ?intVariable 7)
```

- Finally we should explain the statement "seq". All statements within it are processed sequentially. The "seq"-statement is successfull if all of its substatements are successfull. You will get some details about other statements defining a processing-and-evaluation order in the next chapter.

And now: let's see how to start the action...

5.2 How to start an action

There are two ways to get an action started:

- you may set a goal for your agent. It would then search for an action with a matching effect and start that action.
- you may start the action directly per invocation from within another action.

As we don't have another action we will set a goal.

5.2.1 Writing a goal

A goal in JADL has a very easy form:

```
(ont -ontology used-)

(goal
    (var -set of variables-)
    -goal statement which shall be true-
)
```

You can simply put this into a file of its own. We recomend that you put this file into the `../knowledge/` folder of your project. We use `startAction.goal`.

Very often a goal statement describes some state of an object which is already located in your agent's fact base. If you want to use such an object you may reference it as follows:

```
(objref ObjectName ObjectType)
```

Having created a reference you may use the object's name within the goal statement. Notice that if you do only use object references and no variables within your goal you don't need to write down the block for variables at all.

If you want to write a goal don't forget to define the ontologies containing the categories of the objects used within the goal statement. Finally save the goal with the postfix ".goal". You can find the complete goal for starting our fillStocks-action in the appendix at 9.2.5.

5.2.2 Setting up a goal

Now, as we have written a goal we do finally need to set it. Setting a goal means putting it on the agents goal stack. Notice that there is a difference between the agent knowing a goal and the agent having the same goal on its goal stack. An agent may know a lot of goals if you just put them into the agents configuration file in the plan library.

The easiest way to set a goal already contained in the agent's plan library is to use the Debugging GUI. There is a tab called "Edit Goal" which you can find on the Debugging GUI. On that tab you can choose a goal to be set. If you've declared your goal in the configuration file you should find it in that list and choose it to be set. Notice that you don't need to put your goal file into the configuration file but you may also directly write it to the text field of the "edit goal" tab and set it afterwards.

5.2.3 Checking for success

After having done this you can check the actions success by looking at the tab "Log". There should appear a "success"-notification if everything worked out fine, a "failure" if not or a notice telling you that the goal couldn't be parsed. The latter indicates that there's is some syntax error in your goal declaration.

If any exceptions occur you can see them on the console window, too. Finally, for more exact information it is recommended to make a more extensive use of the logging-system with the Debugging Bean. This is done by logging the ExecutionBean or the SelectionBean. You can set the logging options on the tab at the left. Just choose the component you want to log and set its logging detail to "high". After some time you will understand more and more of the logs and you will see that logging is the best way for debugging a JIAC agent application.

Take your time and find out...

6. More JADL

In this chapter we want to create a more or less complex JADL-script. This script is going to be run by a single agent, i.e. there will be no communication and no services involved. We will try to introduce most elements of JADL and give some explanations of what they do and how they should be used.

We assume that you managed to get the example-script from the last chapter running, and that your agent is now properly configured.

> *We will use the same goal and the same action-declaration (those from chapter 5) during this whole chapter, and execute the goal only as a means to trigger the script.*

For the last part of this chapter it will also be necessary that the factbase contains the objects we have declared in the last chapter, because we are going to work with them.

We will start by printing an object to the console, so we can debug our script later. Afterwards we will try to manipulate the object, learn about case-differentiations in JADL and finaly create a loop over our list so we can change all the products. This should give you an overview of the statements and expressions in JADL. But now let's go!

6.1 First steps with an object

As we still have our function from the last chapter, that allows us to print strings to the console let's try to print an object. If we can do this, it will be rather useful for debugging our application later. Unfortunately, JIAC is not able to convert a category into a string by itself. So we have to tell JIAC how to do it. This is obviously something specific for our category, so we should write a function in the ontology.

Let's call the function *productToString*. It does of course belong into our ontology. The header for the function would look like this:

```
(fun string productToString Product)
```

The function is implemented in java-code, just like the *printString*-function we already have. So we put down the `#begincode` and `#endcode`-statements. Now you probably remember that non-base-type parameters are always handed to functions in a somewhat queer form. We first have to cast the parameter into something we can work with. Without making any special assumptions about a parameter-object, we can always cast it to a KObjectRef. So let's do this:

```
(fun string productToString Product)
  #begincode
    KObjectRef prod = (KObjectRef)a0;
    ...
  #endcode
```

You do remember that the parameters are always handed over as variables named *a0*, *a1*, etc... don't you?

Now we start creating a string that will contain all important information about a Product. In Java we can access the attributes of a category by the `getValue()`-method. This method requires a parameter that states which attribute we want to know. Fortunately, constants for those attributes are created by the Ontology-compiler. These are written into the *OntologyIfc*-file, and yes, you can use them directly in your ontology without importing them.

The format for those constants is *Category_attribute*. Thus the constant for the name of a Product would be: `Product_name`. With this information we can start creating a string from the attributes in our Product:

```
(fun string productToString Product)
  #begincode
    KObjectRef prod = (KObjectRef)a0;
    String name = prod.getValue(Product_name);
    double price = prod.getValue(Product_price);
    return("("+name+", "+price+")");
  #endcode
```

This should do for the moment, and allow us to convert a Product to a string. If we put this string into our other function *printString*, we can now print a product from a JADL-script. Let's try this! We will simply change our existing script.

But wait - we need a Product that we can give to our new function. Well, no problem. Do you remember that we could call the *printString*-function with the

string `"Hi!"` as parameter? For the compiler this reads as: Create a new constant of the type *string* with the content *Hi!*. We can do the same with category-objects. You probably remember that we created a Stock-object in that way in the last chapter. You can of course also do this in the middle of a script. The syntax for an object-term looks like this: `(obj` *category (attribute value)** `)`. So we can create a simple Product by using:

```
(obj Product
      (name "TBird")
      (price 11.59)
)
```

If we hand this over to our function, we get our string. The new content for our `FillStocks.jadl`-file looks like this:

```
(package knowledge.FillStocks

  (ont ontology.Trading:DAI_1)

  (act FillStocks
    (var ?stock:Stock)
    (pre (not (att full ?stock true)))
    (eff (att full ?stock true))

    (script
      (var ?p:Product ?s:string ?foo:bool)
      (seq
        (bind ?p (obj Product
                      (name "TBird")
                      (price 11.59)
                 )
        )
        (bind ?s (fun productToString ?p))
        (bind ?foo (fun printString ?s))
      )
    )
  )
)
```

This script is very easy to explain. The *bind*-statement binds an evaluated term to a variable. So in the first line we create our object and bind it to the variable *?p*. In the second line, we call the *productToString*-function and bind the return-value to the variable *?s*. Finally we call the *printString*-function with our string in order to write it to the console.

The *bind*-statement is used, because it's the simplest way to make JADL evaluate the function. This is because functions in JADL are supposed to be mathematical functions, that is they are used to calculate an output-value from their input-values without side-effects. This of course does only make sense, if you actually use the output value afterwards. Therefore JADL expects you to at least catch the return value somewhere.

Furthermore, we already said that you may use nested staments in JADL, so you may put everything into a single statement, if you prefer it that way:

```
(script
  (var ?foo:bool)
  (seq
    (bind ?foo (fun printString
                  (fun productToString
                    (obj Product
                          (name "TBird")
                          (price 11.59)
                    )
                  )
                )
    )
  )
)
```

No matter which way you prefer, we can now test this script. For this we put the code into the action we already defined in the last chapter. We still use the same effect and the same goal for the test, as this is the easiest way to get everything going.

6.2 Manipulating the object

Now we actually want to change something about our product. There are different ways to achive this. But for the moment, we will introduce two different ways of manipulating objects. One way to do this is to write an ontology-function. This function would take an object and a value, and then use the appropriate java-methods to change the attribute-value of the object. Here is an example:

```
(fun Product setProductPrice Product real)
  #begincode
    KObjectRef prod = (KObjectRef)a0;
    double name = a1;
    prod.setValue(Product_price,a1);
    return(prod);
  #endcode
```

Note that if you call the function, you work on a copy of the old object (*call by value*). Thus if you want to actually have a changed object, the JADL-statement would look like this:

```
(bind ?p (fun setProductPrice ?p 2.30))
```

This statement overwrites the old value of the variable with the return-value of the function. Thus the object is changed. By the way: We cannot change the value of the products name in any way, because it's marked *fixed* in the ontology. So we will have to be content with changing the price.

Of course this way of changing objects is quite simple for java-programmers. But we wanted to do JADL, so let's look for different ways of manipulation. The most simple way of manipulating objects in JADL is the *update*-statement. This statement is originally designed to change objects in the factbase. But as the formula for the statement is always evaluated, no matter if the object is in the factbase or not, we may also use it in this case. The syntax looks like this: `(update` *formula* `)`. So for our example we would use the statement:

```
(update (att price ?p 2.30))
```

The update statement tries to make the formula true by changing the stated object. Thus the price for the object is changed to *2.30*. Now, as the formula is true, the agent checks whether the object is in the factbase, and updates it if necessary. But as the object is not yet in the factbase, nothing more happens.

Now let's change our script, and see if it actually works. As the update-statement is the easiest way to change something, we will use this:

```
(script
  (var ?mp:Product ?s:string ?foo:bool)
  (seq
    (bind ?p (obj Product
                  (name "TBird")
                  (price 11.59)
             )
    )
    (bind ?foo (fun printString (fun productToString ?p)))

    (update (att price ?p 2.30))
    (bind ?foo (fun printString (fun productToString ?p)))
  )
)
```

6.3 Controlflow - conditions and branching

Now we get back to the original idea of our action: Fill a stock of products. For this we first need to change our object from a simple Product to a MultipleProduct. This is quite easy, and we will just show you the code for this, trusting that you will be able to understand it:

```
(script
  (var ?mp:MultipleProduct ?s:string ?foo:bool)
  (seq
    (bind ?mp (obj MultipleProduct
                    (name "TBird")
                    (price 11.59)
                    (amount 5)
              )
    )
  )
)
```

As you have learned in the last section, we can now alter the *amount* of our product by using an update-statement. Let's say we can store 20 pieces of every product. So we would change the amount of the product to 20, if we want to fill the stock. But what if a MultipleProduct does already have an amount of 20? Or if we have even more than 20 pieces in our stock? We obviously need a distinction of cases here.

For this, JADL has the *branch*-statement. This statement allows you to define a condition that is evaluated, and depending on the result of this condition, you can specify different courses of action. But unlike other programming languages, JADL allows you to define any number of different cases for the condition. We will have a look at the condition first, to make this clearer:

```
(cond checkAmount
  (var ?mprod:MultipleProduct ?amount:int)

  (att amount ?mprod ?amount) // amount equals int
  true                        // amount is different from int
)
```

This condition is defined outside of an action. It stands on the same level as actions, and can also be imported and used in other jadl-files. As you can see, the condition has two parameters (in the *var*-statement). The MultipleProduct and an integer that is stating the number to compare the amount with. Nothing special here. The body of the condition consists of two formulas: `(att amount ?product ?amount)` and `true`.

Now let's see how such a condition works. Once the condition is called, each formula is evaluated in sequential order. For this evaluation the declared variables are

considered. If one of the formulas evaluates to *true*, then the condition remembers the position of that formula. The other formulas are ignored. For our example this means that if the formula `(att amount ?product ?amount)` can be evaluated to *true*, then the condition would memorize that the first case shall be processed. If the formula evaluates to *false* or *unknown*, then the condition would try the second formula which is always *true*.

Note that you can do this with any number of formulas. The condition will try to evaluate all given formulas until one of them returns *true*. But what does that help us? Well the *branch*-statement takes a condition with a number of different formulas, evaluates them, and jumps to the course of action with a position corresponding to the first *true* formula of the condition. So we need two courses of action for our *branch*-statement, so we can match them to the formulas in our condition. In pseudocode it would look like this:

```
(branch (checkAmount (var ?mp 20))
  do nothing    // first case
  update stock  // second case
)
```

For a first test, we will simply put *printString*-statements into the cases of the branch, so we can see how it works. If you feel like testing it, you can alter the amount of the MultipleProduct, to see how JIAC processes the different branches. The complete script would look like this:

```
(script
  (var ?mp:MultipleProduct ?s:string ?foo:bool)
  (seq
    (bind ?mp (obj MultipleProduct
                   (name "TBird")
                   (price 11.59)
                   (amount 5)
              )
    )

    (branch (checkAmount (var ?mp 20))
      (seq // first case
        (bind ?foo (fun printString "first case"))
      )
      (seq // second case
        (bind ?foo (fun printString "second case"))
      )
    ) // end branch
  )
) // end script
```

Each case of the branch contains a *seq*-statement. This is necessary, because you can define a new execution-order for each single case if you want to. Alternatively you can also use the *alt* and *par*-execution orders here. Of course you have to include the cond-planelement into your .jadl-file, if you want to run this script.

But right now our script is only able to determine whether a product has exactly 20 pieces. We want to check for the amount of pieces and only do something if there are less than 20 pieces. To turn it around, we do nothing if we have more than 19 pieces. So we need to check for this. What we need is a comparison which is working on the amount of products we have. This looks pretty much like something that is specific for our ontology, so we write an ontology-function for this test.

There is a special kind of ontology-function for such a case. It's called *comparison*. This is basically a function with a fixed return-type. A comparison does always return a boolean value. Thus it's perfectly suited for a conditional planelement. But first things first, here's the comparison:

```
(comp amountIsOk MultipleProduct int)
  #begincode
    KObjectRef prod = (KObjectRef)a0;
    long value = a1;
    return(prod.getValue(MultipleProduct_amount)) >= value);
  #endcode
```

So a comparison is really quite the same as a function. The only differences are the keyword and the lack of a return-parameter (it's fixed anyway, so why write it down?). Now we can use this comparison inside our conditional planelement instead of the proposition. The new planelement looks as follows:

```
(cond checkAmount
  (var ?mprod:MultipleProduct ?a:int)

  (comp amountIsOk ?mprod ?a) // amount is at least a
  true                        // amount is less than a
)
```

As this obviously works better than the old version, we will assume that we use this condition for the rest of the tutorial. One final sidenote for this section: Of course you do not have to implement comparisons like this one for every application. If you want to do Integer-comparisons, you will find some valuable libraries in the jiac-release, that already implement these functions. We just showed you how to write them, because we feel that it is useful to know.

6.4 Working with lists

Now that we can alter a product depending on its state, we have to start working with a list of products. Most programmers would want to use a loop to do this. As

a matter of fact, JADL offers a *loop*-statement, that allows you to repeat actions. The syntax for this would be: `(loop` *(condition) execution-order**`)`

Digression: **Loop**

This allows you to define a number of cases similar to a *branch*-statement. The difference is that a *loop* continues to endlessly process one of these cases as long as no failure occurs. For the control of the loop, there exist two statements: *break* and *cont*. While the first exits the loop, and continues with the next statement after the loop, the *cont*-statement is used to return to the top of the loop and continue with the next iteration.
A typical loop would look like this:

```
(loop (isTrue(var ?b))
  (seq // this is the body of the loop
    ...
  )
  break // finish loop
)
```

isTrue is a conditional planelement, that has two possible effects. As long as the first effect is true, the loop is repeated. If the second effect becomes true (meaning that the first effect was false), the loop goes to the second case which terminates the loop.
There are two more things to mention: First of all, if you want to count something in the loop or use some index-variables, you have to administer them yourself. There is no such thing as `i++` or `iterator.next()` in JADL - this has to be done with ontology-functions. Secondly, if you use any local variables inside the loop, those variables are not cleared after one iteration.

If the loop goes to the next iteration, all variables carry their contents with them. So if you need any variables to be empty, you have to **unbind** *them explicitly.*

But wait. Do you remember how we defined our list of products? The ontology says something about `MultipleProduct[]`. This is a JADL-List. And as one might expect, JADL has some support for it's own lists. We don't want to handle indices and list-sizes - we just want to touch every element of the list. The JADL-statement that allows us to do so is *iseq*.

The *iseq*-statement stands for *iterated-sequential execution*. This means that each element of the provided list is given to the upcoming instructions. Analogous to the simple execution-orders, there also exist *ipar* and *ialt*-statements. The syntax is: `(iseq` *list* `(var` *variable*`)` *execution*`)`

As you can see, the statement takes the list of objects and a variable declaration. The variable has to be of the same type as the objects in the list. During each iteration the variable is bound to a new object of the list, thus allowing you to work with these objects. So you can easily process the list, without worrying about it's length.

The execution part can be anything of *seq*, *par* or *alt*. It is necessary, because the *iseq* statement only defines a sequential processing order for the elements of the list. It does not define anything for the execution of the instructions.

For our example, the header of the iteration looks as follows:

```
(iseq ?mpList (var ?item:MultipleProduct)
  (seq
    ...
  )
)
```

In this example, the variable `?mpList` is the list of products we want to work on. The `?item`-variable will be used to get each single element from the list. After we adapt our remaining code to this variable, we get this:

```
(script
  (var ?mpList:MultipleProduct[] ?foo:bool ?stock:Stock)
  (seq
    // get list of products
    (eval (att products ?stock ?mpList))

    (iseq ?mpList (var ?item:MultipleProduct)
      (seq
        (bind ?foo (fun printString (fun productToString ?item)))
        // --> check the amount
        (branch (checkAmount (var ?item 20))
          (seq // amount is at least 20
            (bind ?foo (fun printString "product.amount is 20 - ok"))
          )
          (seq // amount is less than 20
            (bind ?foo (fun printString "product.amount is filled"))
            (update (att amount ?item 20))
          )
        ) // --> end branch
      )
    ) // --> end iseq
  )
) // end script
```

You may have noticed that the statement `(eval (att products ?stock ?mpList))` at the top of the script is new. This is the opposite of an update-statement. It tries

to make the formula true, by binding empty variables with appropriate values. As the variable *?mpList* is empty at the start of the script, it is bound to the list of products in the *Stock*. You now have a statement, that allows you to read attributes from a category-object. Note that this statement is also considering objects in the factbase when trying to fullfil the formula. So you can use it to get an object from the factbase.

But wait, there is one problem with the *iseq*-statement. Because of the synchronization problems you would get if different scripts work on a list at the same time, any changes you make on the elements of the list are **not** written to the original list. Thus we cannot actually change the list by manipulating it's objects with this statement. But we can put all objects into a new list, and exchange those two lists after the iteration is complete. For this we first need a way to add objects to the list.

There is no generic way for this. As the lists in JADL have strong types, the signature of an add-function as we need it, cannot be determined generically. It is individual for each type of list. Thus we have to write an ontology-function, which realizes the add-function. This ontology-function would look like this:

```
(fun MultipleProduct[] addToList MultipleProduct[] MultipleProduct)
  #begincode
    KList inList = (KList)a0;
    KObjectRef inProd = (KObjectRef)a1;
    inList.add(inProd);
    return(inList);
  #endcode
```

This function simply takes a list of MultipleProducts and a single MultipleProduct and adds the product to the list. Finaly the new list is returned. We now can use this function to fill the new list with the contents of the old list, after they are updated. In pseudocode it would look like this:

```
(script
  (seq
    initialize newList
    getOldList
    (iterate over oldList
      (branch
        if element has at least 20 pieces do nothing
        if element has less than 20 pieces update element
      )
      addElement to newList
    ) // end iterate
     put newList into stock
  )
)
```

Now let's have a look at the actual JADL-Code for this. Obviously we can reuse most of the code we already had. We just need to add a few lines of code for the handling of the new list:

```
(script
  (var  ?mpList:MultipleProduct[]
        ?newList:?mpList:MultipleProduct[]
        ?foo:bool
        ?stock:Stock)

  (seq

    // get list of products
    (eval (att products ?stock ?mpList))
    // initialize newList
    (bind ?newList [MultipleProduct:])

    (iseq ?mpList (var ?item:MultipleProduct)
      (seq
        (bind ?foo (fun printString (fun productToString ?item)))

        // --> check the amount
        (branch (checkAmount (var ?item 20))

          (seq // amount is at least 20
            // --> amount is ok
            (bind ?foo (fun printString "product.amount is 20 - ok"))
          )
          (seq // amount is less than 20
            (bind ?foo (fun printString "product.amount is filled"))
            (update (att amount ?item 20))
          )
        ) // --> end branch

        // add item to newList
        (bind ?newList (fun addToList ?newList ?item))
      )
    ) // --> end iseq

    // update List in stock
    (update (att products ?stock ?newList))
  )
) // end script
```

Again, there are a couple of lines worth mentioning. First there is the line `(bind ?newList (fun addToList ?newList ?item))`. It is necessary to bind the return-value of the function to *?newList* again, because functions are not allowed to manipulate their parameters. JADL-functions use call-by-value, that is functions only get

copies of all their parameters. Thus we have to overwrite the *?newList* to actually change it.

Also we should mention the *update*-statement at the end of the script. We use this to change the contents of the *stock*-object. By this the list is replaced, and the stock holds the new productlist. As in this case the replacement of the list is an atomic action, there are no synchronization problems. It is simply the replacement of a reference and not a list of actions on single elements.

Furthermore there is the initialization of the list. Our *addToList*-function needs an existing list. It can't work with an unbound list-variable. Thus we have to create an empty list. This is done by the statement *[MultipleProduct:]*. This creates a constant empty list, which is bound to the variable *?newList*. Basically it's the same syntax as in the object definition, with the difference being, that now objects are defined inside the list.

Last but not least we need to achive the stated effect of our script happen. To do this, we simply add one last line to the script. The line is:

```
(update (att full ?stock true))
```

By now you should be able to guess what it does. It simply sets the attribute *full* of the stock to true, thus producing the stated effect.

7. Interaction & Services

This chapter gives you an overview over the communication-infrastructure of JIAC. It also shows you, how to use this infrastructure and how to get agents to communicate with each other. Unlike the other chapters, we start with some theory about how JIAC enables communication. But don't be worried about that. The sending and receiving of messages is actually quite easy. We just need the theory to understand how it works and how to use it correctly.

7.1 Communication in JIAC

Being an Agent- & Service-Framework, JIAC provides some very elaborate infrastructure for the interaction and communication that is necessary for services. The first and most important rule for communication in JIAC is:

> *'Communication' does always mean: 'Using a Service'.*

Thus, whenever you want to contact another agent, you have to use a service of that agent. This may seem like a lot of overhead in the first place, but once you start creating more complex interactions between agents, you will find that the service-concept makes your work a lot easier.

The first thing you will notice here, is that communication does always happen on a one-to-one basis in JIAC. There is always one *user* and one *provider* involved. Things like Multicasting, Blackboard-Communication or a change of communication-partners can not be implemented directly, but have to be modeled with this service-concept.

While the conversation as a whole is wrapped in a service, the single messages that are send between the agents, are called *Speechacts*[1]. For the moment, you may regard

[1]The original concept of speechacts comes from the FIPA (www.fipa.org) and is designed to allow for interoperability between different agent-systems.

a speechact as a simple message that has a sender, a receiver and a content. JIAC takes such speechacts, uses JAVA-Serialization to make them sendable, and delivers them to the appropriate agent. The good thing about this is: Even though you are working with a Multi-Agent-System, once the agents have agreed on a service-session, the sender and receiver of the speechacts are clear. You do not have to worry about addresses, domains, or the transport-protocols that are used. JIAC will take care of all those things. So you can concentrate on what happens inside the service.

7.1.1 Transportation

Depending on your scenario and the available hardware, you may have different possibilities of how to transport the messages. The most common way for this is TCP/IP. Other alternatives include SSL-communication, HTTP-transport or the very simple JVM-communication[2]. Depending on what you want, all you have to do is give your agents the appropriate components. The actual protocols, the transportation, and the resolving of addresses are handled by the platform. Thus the JIAC-Framework allows you to forget about the transportation-layer.

7.1.2 Services

A service in JIAC can be easily described as an action that is executed by another agent. Actually this description does fit quite well. The *user* of the service is looking for an action, that helps to fulfill a certain goal. If the agent cannot find such an action of it's own, it starts looking for services. Therefore a service is selected for execution in the same manner as a normal action. Once the service is selected, the agent starts to initiate the service. For this the *Meta-Protocol* is used.

Virtually the only difference between a service and a normal action is the fact, that the service has no execution-part of it's own. Instead it references a list of protocols which realize the execution. The interesting thing here is, that a service can have any number of protocols attached to it. It does not matter how the protocols reach the effect of the service. Once the service is selected, both agents - the user and the provider - start negotiating about which protocol to use. If they can find a protocol which they both support, they will use it and start the execution.

A good example of how this can be of advantage is user-authentification. You may for example have two protocols for a service, one of which simply realizes the functionality, while the other requires the user to authenticate in the beginning of the service. The reason for this would be, that the user-agent supports both protocols, thus using the simple version if possible, and the authentification-version if it is required. Different provider-agents would support only one of the protocols, depending on their individual requirements.

7.1.3 Protocols

A protocol for a Service consists of two protocol-roles. Each participating agent gets its own role. These protocol-sides reference the service they implement, and contain the actual execution-part. Note that the protocols inherit the variables declared

[2]Meaning that the messages are delivered via the JavaVirtualMachine. Thus the agents can only communicate, if they are running in the same virtual machine.

in the service. Thus you can regard the service-variables as some kind of global variables.

There is a special case, which will become clearer in the next section. Sometimes the user doesn't really have to do anything. In this case, you can specify that the user should not have a protocol, and that the protocol has only a provider side. Of course you can mix two-sided-protocols and provider-only-protocols in one service.

One good thing about these protocols is, that you don't have to care about the selection and the execution of the protocols. This runs totally automatical, if your agent is configured properly. The protcols are selected and each side starts the execution of its protocol. If one of the agents decides to send a speechact, it is automatically delivered to the service-partner. And if anything goes wrong, the agents complain and terminate the service-usage. This is all handled by:

7.1.4 The Service-Metaprotocol

This is both the most complex and the most useful thing in the interaction of JIAC-Agents. But fortunately, you don't have to worry about how the Meta-protocol works. All you need to know is what it does. Here is an overview:

- it wraps each single service usage
- it handles the negotiation that takes place if a service is initiated
- it takes care of the communication adresses of agents and the search for service providers
- if appropriate, it allows the participating agents to negotiate about service parameters
- if anything goes wrong, i.e. one of the agents crashes, the protocol is violated or the connection is broken, the Meta-protocol makes sure, that the agents know that the service has failed
- it assures sure, that the global variables of the service are send to the provider at the start, and back to the user in the end

Most of these points don't have to be explained in detail. They just make implementing the interaction between agents very convenient. But we have to say something about the last point. We already told you that the variables declared in a service are globaly known. This means, that they are known to the user and the provider. As the user is the one who wants the service to take place, he is responsible to set initial values for those variables. Of course the provider has to receive these values, to do anything useful. Because of this, the Meta-protocol automatically sends the first speechact of each protocol. This speechact simply updates the global variables on the provider-side.

Once the service is finished, the Meta-protocol makes sure that the provider sends the values of these variables back to the user. So the user always gets the results of the service.

> *The Meta-protocol always sends the first and the last speechact of each service-usage.*

You should remember that, if you design your protocols on paper and want to implement them. You always need to send two speechacts less than you have to design.

This is also the reason, why the *provider-only*-protocols mentioned in the last section can be pretty useful. If the user does nothing more than send some objects to the provider and wait for the results, then you don't have to implement a protocol for this. The Meta-protocol already does this for you, and you only have to implement the actions for the provider.

But now it's time to apply all this. We will start to implement a simple service that just goes to show how things are done. Considering our scenario, we would want to order some products from the producer. So we will write a service, that allows us to do so.

7.2 A Service

As we told you in the last section, there is only little difference between a service and an ordinary action. One thing they both have in common is, that you need a goal to start a service. We will specify this goal in advance, as this gives us a first idea of what our service should do.

We want to order products. But what, if the products we want are not available? Well we will just formulate the goal in such a way, that the service helps us to find out about this. We want to know if the goods are available. If we translate this to JADL it looks as follows:

```
(goal
  (var ?order:Order)
  (known (var ?b:bool)(att available ?order ?b))
)
```

The *Order* is a new category that we need for this service. It is very simple and looks like this:

```
(cat Order
  (products MultipleProduct[] needed)
  (available bool)
)
```

Of course this category belongs into our Trading-ontology. But you would have guessed that.

For the goal to work properly, we need to provide an *Order* that can be processed. We will do this directly in the goal. The final goal looks like this:

```
(goal
  (known
    (var ?b:bool)
    (att available (obj Order
                        (products
                            [MultipleProduct:
                              (obj MultipleProduct
                                    (name "foo")
                                    (amount 5)
                              )
                              (obj MultipleProduct
                                    (name "small")
                                    (amount 2)
                              )
                            ]
                        )
                )
    ?b)
  )
)
```

Ok. This is our goal. Now we need a way to reach that goal. Let's open a new JADL-file and write a service for this.

7.2.1 Servicedeclaration

Of course the effect of the service must match the goal. For the precondition, we just assume that it's true right now. The file with the basic service declaration looks as follows:

```
(package knowledge.trading.ProductServices

  (ont trading.ontology.Trading:Version_0)
  (ont de.dailab.kit.ontology.Service:DAI_1)

  (act GetProductsService
    (var ?order:Order ?b:bool)
    (pre true)
    (eff (known (att available ?order ?b)))
    (service ...)
  )
)
```

Note that we imported `de.dailab.kit.ontology.Service:DAI_1` in addition to our own ontology. This ontology comes from the standard JIAC-infrastructure, and defines a category for services. We need to instantiate this category, because the agent has to know about the service.

7.2.2 Details about the services

The important attributes for a service are *name*, *protocols* and *ontologies*. These are all needed for the category. Of course you may have a look at the other attributes if you want to, but that would be too much of a digression here. The name is of course a non-empty string. You can use anything you want here, but please make it meaningful. Ontologies is a set of strings, describing ontology-names. This is rather for humans who want to know which ontologies are used in this service - the attribute is not yet evaluated by the architecture.

The final attribute is *protocols*. This attribute must contain a list of protocols that can be used for the service. Such a protocol is a category of its own. The declaration for this category is in the same ontology as the service-category. But first things first, let's write down the service:

```
(act GetProductsService
  (var ?order:Order ?b:bool)
  (pre true)
  (eff (known (att available ?order ?b)))
  (service
    (obj Service
        (name "GetProductsService")
        (protocols ...)
        (ontologies
          {string: "trading.ontology.Trading:Version_0"}
        )
    )
  )
)
```

So we simply put the object-declaration at the end of the action-header. As there is no execution-part in a Service, we have nothing else to do. Except the protocols of course. So let's start defining a protocol. To keep an overview on the names, we will call the protocol *GetProductsProtocol*. For the moment, we need nothing more than the name of the protocol. So we can define the protocol-object for our service:

```
(obj Protocol
      (name "GetProductsProtocol")
    // (provider true)
)
```

The second attribute *provider* is used, to state whether the protocol has a user-side or not. This can be read as *provider-only*. So if you set it to *true*, the user-agent

does not search for a protocol when the service is started. As we do not yet know, if we have a user-side or not, we keep this attribute in a comment.

The service expects a list of protocols (you remember: we can have multiple protocols for each service), so we have to do a little more coding:

```
(act GetProductsService
  (var ?order:Order ?b:bool)
  (pre true)
  (eff (known (att available ?order ?b)))
  (service
    (obj Service
        (name "GetProductsService")
        (protocols
            [Protocol:
              (obj Protocol
                    (name "GetProductsProtocol")
                    // (provider true)
              )
            ]
        )
        (ontologies
          {string: "trading.ontology.Trading:Version_0"}
        )
    )
  )
)
```

7.3 Protocols

Now we can start implementing our protocol. It is obvious, that every protocol has two protocol-roles. The *user* and the *provider*. As each side must have it's own version of the protocol, we will put them into two separate files. To make this obvious to everybody, we will call the files *ProductsUser.jadl* and *ProductsProvider.jadl*.

Of course, for a real protocol it would be helpful to know what messages are sent and how synchronization is realized, in advance. But we will skip the design part for this example, and just go for the implementation. But please notice that all messages between agents are asynchronous. That means, that you cannot know, when a message will arrive at its receiver, and when the answer will return. At this point, the `par` and `alt`-execution orders for scripts come in very handy. You may look them up in the Programmers Guide.

But now we'll start with the provider.

7.3.1 Provider

The skeleton of the `ProductsProvider.jadl`-file looks as usual. Package-declaration, imports and ontologies. You should notice though, that we imported the ProductServices-file. This is necessary, because the protocols need to reference the service[3].

```
(package trading.knowledge.ProductProvider

  (ont trading.ontology.Trading:Version_0)

  (import trading.knowledge.ProductServices GetProductsService)

  ...
)
```

The Provider-protocol itself is again an action-declaration. As protocols are only selected by an agent in the context of a corresponding service, the protocol needs no precondition or effect. So we can set both to *true*. Of course, therefore the action must be declared to be a protocol. For this we use the following line:

```
(prot GetProductsProtocol GetProductsService provider)
```

This line contains all information that is necessary to bind the protocol to the service. The syntax is: `(prot` *protocol-name service-name protocol-role*`)`, where the protocol-name is the name we used in the service-object. *service-name* is the name of the service-declaration (not the name of the file). And the protocol-role is either `user` or `provider`.

Finally the protocol has an execution-part. This is usually a script. But you may also decide to go directly for a call into a component (See chapter: ...). The complete code looks like this:

```
(act GetProductsProvider
  // (var ?order:Order ?b:bool)
  (prot GetProductsService GetProductsProtocol provider)
  (pre true)
  (eff true)
  (script
    (seq ...)
  )
)
```

Note that we put the variable declaration of the service as a comment into the act. Of course you could also define local variables at this point. But the comment

[3]The service references the protocols only by their names, so it doesn't really need them. But the protocols use the variables of the service, therefore they can only be compiled **after** the service.

is there to remind you that the variables from the service are handed over to the protocol. This is done by the Meta-protocol. So you can use those variables here, as if you had declared them in this action.

Variables, that were declared for the Service, are always known in both protocol-roles

7.3.2 User

The user-side of a protocol is implemented in pretty much the same way as the provider side. You just should make sure, that you use `user` in the *prot*-declaration, and that you put the protocol in a different file. The latter is necessary, because you want to give the protocol-roles to different agents - at least usually.

So here's the code:

```
(act GetProductsUser
  // (var ?order:Order ?b:bool)
  (prot GetProductsService GetProductsProtocol user)
  (pre true)
  (eff true)
  (script
    (seq ...)
  )
)
```

Again we would like to remind you, that you do not necessarily need a user-role for a protocol. If the exchange of information can be handled by the Meta-protocol, i.e. 'send it to the provider' at the start and 'return it to the user' at the end, then you can skip the user-role. In this case you have to set the `(provider true)` line, in the protocol-declaration of the service.

7.4 Sending Speechacts

Now that we have our basic protocols, can share the objects and write some scripts to work with the objects, you will probably wonder how you can communicate while the protocol is running. As we already mentioned, JIAC-agents use the FIPA-Speechact concept for communication. This concept defines a format for speechacts, and allows different content-languages to be used inside the messages. The good thing is, you don't have to worry about most of this.

Usually, you just have to define what you want to send, and what *type* the speechact has. FIPA defines a whole lot of different speechact-types that can be used. But as JIAC-services have such an elaborate protocol-concept, you only need the *inform*-speechacts most of the time. So this is the only type of speechact we will introduce here.

7.4.1 send...

Sending a speechact is an element of JADL. So we introduce a new language-element. This is used to define the sending of a speechact, together with its contents. This definition can afterwards be used as often as you like, even in other planelement-files, if you import it.

The syntax looks like this: `(send` *name variable-declaration type*`)`. The name is anything you choose. It is used to call the send-statement later. The variable-declaration is used to define the types of objects you want to send with the speechact - as usual this is optional. And the *type* usually looks like this: `(inform (data` *variables* `))`. Note that the data-part is optional, and you can skip it, if you just want to use the speechact for synchronization.

So if we want to define a new send-action it would look like this:

```
(send sendProduct
  (var ?p:Product)
  (inform (data ?p))
)
```

That's it for the definition part. Now we want to try and send an object. This is done simply by calling the send-action from a script. Here it is:

```
(script
  (var ?prod:Product)
  (seq
    (bind ?prod (obj Product
                      (name "foo")
                      (price 2.30)
                )
    )
    (sendProduct (var ?prod))
  )
)
```

And that's all. The agent will bind the product to the send-action, and send it to the service-partner as the data-part of an *inform-speechact*. So there's nothing special about this. The *send*-statement is nearly always successful in a script. Only if the sending agent itself cannot process the speechact for some reason (for example, if the CommunicationBean was just removed), the statement would fail. But this hardly ever happens.

If the receiver of the speechact has any problems, he will send a *not-understood*-speechact. If you do not catch it yourself, this will be caught by the Meta-protocol, and the service will fail. If you do want an acknowledgement for your message, you have to send it yourself from the other side of the protocol.

Of course you may also set things like *time-outs* or *reply-tags* for a speechact, but this would lead to far here.

7.4.2 ... and receive

On the other side of the protocol, you have to receive something. This has to be done explicitly, because the agent must know, what to receive and when. The basic *receive*-action is pretty similar to the send-action. Here is the code:

```
(receive receiveProduct
  (var ?p:Product)
  (inform (data ?p))
)
```

So it's just symmetrical to the send-action. At this point it might be useful to explain how the receive-action works. The inclusion into your code is pretty simple:

```
(script
  (var ?rec:Product ?foo:bool)
  (seq
    (receiveProduct (var ?rec))
    (bind ?foo (fun printString (fun productToString ?rec)))
  )
)
```

Once the agent encounters the call for the receiveProduct-action, it stops executing the script. The CommunicationBean is notified, that the script is waiting for a speechact, and the agent starts doing other things. Once a speechact arrives at the agent, the CommunicationBean tries to find out, to which service-usage the speechact belongs. If the speechact is assigned to this service, the receive-statement is evaluated, and the script is reactivated. Only then will the *'(bind ?foo...'*-statement become active.

At this point you can see, why it can be useful to use the parallel execution-order. In a parallel execution, only the part with the receive-statement would stop, and the other part would continue.

We also have to point out, that the receive-action is only successful, if it receives the correct speechact, with the correct contents. If you would send another object or another type of speechact, it would be compared with the declared type and contents of the receive-action. If these don't match, the receive-statement in the script would fail. At this point, the *alt* execution-order comes in handy, if you want to make your protocols error-resistant.

7.4.3 Receiving different contents

There is one more feature of the receive-statement that can be very useful. You can define multiple speechacts you are ready to receive. This works similar to other

plan-elements with multiple effects. You simply define all the speechacts you want to accept - together with the appropriate variables - and then use the receive-action inside a branch-statement. The code would look like this:

```
(receive receiveProduct
  (var ?p:Product ?s:string)
  (inform (data ?p))
  (inform (data ?s))
)
```

With the following script:

```
(script
  (var ?rec:Product ?str:string ?foo:bool)
  (seq
    (branch (receiveProduct (var ?rec ?str))
      (seq // receive product
        (bind ?foo (fun printString (fun productToString ?rec)))
      )
      (seq // receive string
        (bind ?foo (fun printString ?str))
      )
    ) // end branch
  )
)
```

And now you do know, why the branch takes such a strange argument. It is actually able to process different kinds of planelements, as long as they have the appropriate number of cases. Depending on what kind of speechact the receive-action gets, it will return with the case for that, and thus allow the branch-statement to jump to the corresponding actions. If the receive-action gets an illegal speechact, it will of course still fail, thus letting the branch-statement fail.

8. AgentBeans - beyond the agent

Now that you have some knowledge of JADL and Services, it's time to explain how everything works. In this chapter we will introduce you to the standard architecture of a JIAC-Agent. Furthermore we will explain how you can implement AgentBeans for JIAC and what you can do with them. These Beans are the agents' way of interacting with things that are not JIAC-Agents, e.g. human users, databases, etc.

8.1 How does my agent work

By now you will probably wonder how a JIAC-Agent can select the appropriate actions for a goal, process those actions, and interpret the JADL-code that makes up a script. Also there are things like services and communication, which the agent has handled almost on its own. But how is all this realized?

Well, we already mentioned in chapter ... that JIAC-Agents are made up of different components. These components include Ontologies, Planelements and AgentBeans. You have already learned how to exchange the first two of these. Either you choose to use predefined ontologies and planelements for your agent, or you create implementations of your own.

It's quite the same with the AgentsBeans. One important part of the JIAC-Framework is the standard-architecture for agents. This consists mainly of a couple of planelements, that realize the management and infrastructure functionalities, as well as a list of AgentBeans, that make the agent work. We already introduced that list in section ... . So now let's have a look inside.

8.1.1 The component-system

The component-system was created to allow as much flexibility and independence between the components as possible. Thus the system is not actually specific to JIAC, but it might also be used with other applications or frameworks. The framework allows you to combine an open number of different components, let them communicate through messages, and even exchange the components at runtime.

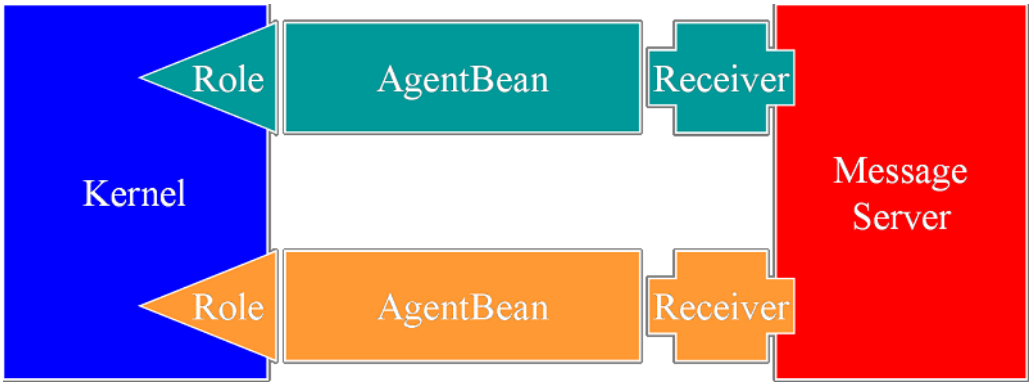
We will not give a detailed introduction into the internals of the component-system here. But we will give you a short overview of what's happening, so you get a better understanding of JIAC-agents.

There are three fundamental parts, that control and maintain the components. These parts are:

- the **KernelBean** is a component that administers the other components, controls the life-cycle state and takes care of the general configuration of the system.
- the **ControlCycle** is in charge of the agents *main-thread*. This thread provides computing time to components that do not have a thread of their own. The ControlCycle decides which component gets to work next.
- the **MessageServer** stores and delivers the messages that are send between components. These messages are the usual means of interaction between components and thus are an integral feature of the component-system.

If you want to integrate a new AgentBean into this system, your bean needs to fulfill certain requirements. Of course the bean must be registered at the *KernelBean*. Otherwise it could not be configured or activated. For this, all AgentBeans need to have an interface, called the *ComponentRole*. This role-interface is loaded and registered at the KernelBean, and allows the other AgentBeans to address your bean with messages. These interfaces play a very central role in the component-system, because the system accesses your bean through this role-interface almost all the time. Also this is how the beans can be exchanged at runtime. As the role-interface stays, all other parts of the system have still access to the interface, and only the access from the interface to the bean has to be replaced.

The other thing you need is a *receiver-interface* for messages. If the *MessageServer* wants to deliver a message to your bean, it tries to call a method inside the message. This method calls a receiver-interface, that hands the message over to your bean. So you have to know what kinds of messages you want to receive, and implement the appropriate interfaces.



Again in simple words - your bean needs to implement a role-interface to allow access from the KernelBean, and a list of receiver-interfaces to allow the delivery of messages from the MessageServer. You do not need to care about the ControlCycle, because it only interacts with the MessageServer by deciding which Message will be delivered next.

8.1.2 Interfaces - Roles and Receivers

How do these interfaces look in detail? Well if you are new to JIAC, you probably want to take the simple solution first. If you want to create a new role-interface, it will likely be for a bean that is specific for your application. The JIAC-release provides a default-solution for this case. The *ApplicationRole* together with the *DefaultApplicationBean* provides an interface that you can easily use for your first steps.

If you let your bean implement the ApplicationRole, you don't have to do anything else to put it into an agent. The code would look something like this:

```
public class MyBean extends DefaultApplicationBean
                            implements ApplicationRole {
  ...
}
```

And that's it. The only problem with this is, that each role-interface may only be used once in an agent. Do you remember that the role-interface is used for addressing the messages? So you can imagine what would happen, if a role-interface is used twice. To avoid this problem, you can write your own role, and let it extend a group of role-interfaces. For beginners this would be the *ApplicationGroup*. A group of role-interfaces is simply another interface, that gives certain default-rights and properties to a role-interface. Thus your first own role could look like this.

```
public interface MyRole extends ApplicationGroup {
  // nothing more to do here for now
}
```

And this actually works. The role-interface extends the ApplicationGroup, thus giving it the same rights as the normal ApplicationRole. You can create any number of different role-interfaces this way, and put them all into the same agent. Note: The naming-conventions for role-interfaces recomemnd that the name of your interfaces ends with **Role**. This makes it easier for other programmers to recognize its purpose.

Concerning the receiver-interfaces, you don't have to do anything right now. The *ApplicationGroup* already provides you with the necessery receivers, thus you can skip that part. But just in case you stumble upon such a receiver-interface one day, we will show to you how it would look like:

```
public interface MyRole extends ApplicationGroup,
                                ChangeGoalReceiver {
  // nothing more to do here for now
}
```

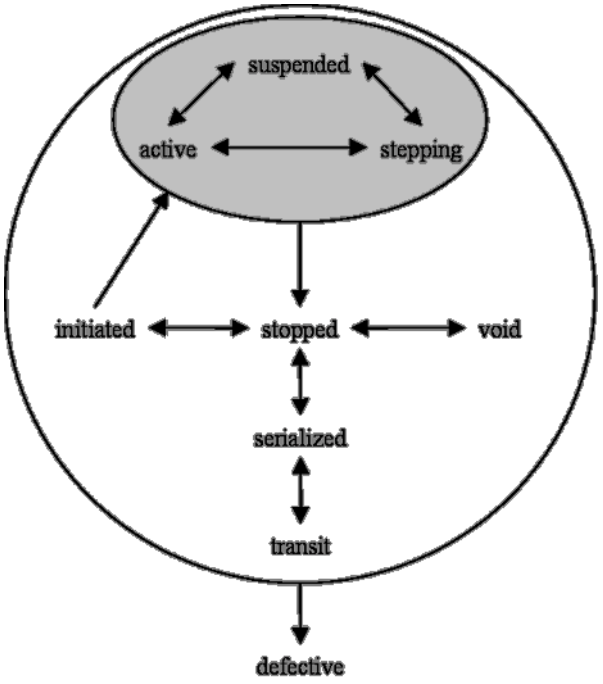
And the corresponding Bean:

```
public class MyBean implements MyRole {

    // Methods for ChangeGoalReceiver
    public void addGoal(Message reply, Goal goal) {...}

    public void removeGoal(Message reply, Goal goal) {...}

    public void activateGoal(Message reply, Goal goal,
                             boolean active) {...}
}
```

This would allow your bean to receive messages for new goals. The methods we sketched in this class are all defined in the *ChangeGoalReceiver*-interface, and thus must be implemented by our bean. This doesn't make much sense in a real application, because the *ChangeGoalMessages* are all handled by the *GoalBean* of an agent. But it gives you an idea about how messages are handled.

8.1.3 Lifecycle

The lifecycle of an agent is controlled by the *KernelBean*. The current state of the agent determines what the components can or cannot do. For example, messages between components are not delivered, unless the agent is *active*. They can be send though. An overview on the lifecycles can be seen here:



Upon creation, an agent starts with state *void*. The first change is made to the state *stopped*, which is the basis for all other changes an agent can undergo. For

example if an agent shall be serialized and saved to a harddisk, it must be set to the stopped-state first.

If the agent is started, it will proceed through the state *initiated*. This state is used for initializing beanspecific data and starting all components. Once this state is completed, the agent will change to one of the *active* states. Only in these states is the componentsystem actually going to work. This is because the messages actually delivered and executed here (the MessageServer will already accept messages in the *stopped*-state). The difference between the states *active*, *stepping* and *suspended* is the speed at which the agent is working. The maximum performance can be reached with the *active*-state. The *stepping*-state expects a user-confirmation for each step of the control-cycle. And in the *suspended*-state the control-cycle will take a break (Note that components with own threads may continue to work, depending on their implementation).

8.2 The standard architecture

8.2.1 Agent physiology

Now we will give you a brief introduction into how a JIAC-agent works. It is made up by a list of core-AgentBeans, which control and enable the agent. The list of the components is here:

- **FactBase**: The factbase is the memory of the agent. All (JADL)-Objects the agent knows about are stored here. For convenience, every component has a *FactAccess*, thus it can directly work with the factbase, without any message being necessary. [1]
- **GoalBean**: This component is responsible for the management of the goals. All goals for the agent must be send to this bean, and afterwards the bean selects wich goal shall be worked on next.
- **SelectionBean**: This bean gets a goal, and tries to find an appropriate action that fulfills the goal. It first tries to search through the own actions of an agent, and if that was not successful, the bean tries to find a suitable service. This search may inlcude requests to the *Directory Facilitator*.
- **SchedulerBean**: This component realizes a simple scheduling algorithm, that tries to keep the number of executing scripts in a reasonable dimension. For the moment you can ignore this bean and simply assume that it hands the selected actions over to the ExecutionBean.
- **ExecutionBean**: Does the actual JADL-Execution. It works on a JADL-script, and executes the instructions step by step. If necessary, this bean creates new goals or uses the functionalities of the CommunicationBean.
- **CommunicationBean**: Realizes the Meta-Protocol and does the analyzing and classifying of the speechacts. For each running service, this bean memorizes a session, that allows it to recognize an incomming speechact. Different TransportationBeans are used for actual transmission (TCP/IP, JVM, ...).

[1] This is one of the reasons why all agents must have a FactBean.

- **CommunityBean**: This bean is used to access services from the AMS and DF. It looks for agent-addresses, sends requests for service-lists and in general does everything that has to do with reaching other agents.

8.2.2 An Agent in action

Now we will give you a short overview of what happens, if an agent gets a new goal. First of all, the goal has to be send to the GoalBean with a *ChangeGoalMessage*. This message is executed by the bean and the goal is added to the goal stack. Once the goal bean is about to select a new goal, it takes the goal and sends it to the SelectionBean inside a *ReachGoalMessage*.

The SelectionBean selects an appropriate action or service, wraps it into an *Intention* together with the goal, and sends a *ChangeIntentionMessage* to the SchedulerBean. This bean creates an *ExecIntentionMessage* that is processed by the ExecutionBean.

If the ExecutionBean needs to interact with other agents, it sends either a *SendSpeechActMessage* or a *ChangeServiceMessage* to the CommunicationBean.

All Components are allowed to send ChangeGoalMessages to the GoalBean, thus everybody can set up new goals.

8.3 Implementing a new agentbean

After all this theory, we will give you a simple example of how to implement a new agentbean by using the DefaultApplicationBean. We will also show to you how you can access this bean from a JADL-script.

For our example we will assume, that we implement a script that requests an order from the user. It is obvious, that we have to implement a GUI for that. As this GUI is interacting with something that is not an agent, a bean would be appropriate. For this bean, the first thing we need is:

8.3.1 The OrderRole

This is quite simple. We already gave you some introductions to the ComponentRoles in section ??. So all we do is just give you the code for the role-interface:

```
package trading.role;

import de.dailab.control.role.ApplicationGroup;

public interface OrderRole extends ApplicationGroup {

  /* call for getOrder */
  public final static int GETORDER = 1;

}
```

All this role does, is to extend the ApplicationGroup. This will make it an role-interface for the component-system and gives us all necessary ReceiverInterfaces. Also we defined a constant *GETORDER* wich we will use later. It is considered best practice to define constants like this in the role-interface, because they can be found and understood easier this way.

8.3.2 The JADL-Script

As our main-method of doing something inside an agent is JADL, we now need a way to access the agentbean from JADL. For this, there is a generic way how the ExecutionBean can be triggered to send a special message to another bean.

JADL has so called *primitive* planelements, which cause an ExecIntentionMessage to be send to a given role-interface. All beans that extend the ApplicationGroup are allowed to receive those messages.

But how does such a primitive planelement look like? Well here is an example:

```
(act getOrder
  (var ?ord:Order)
  (pre true)
  (eff true)
  (call trading.role.OrderRole 1) // call to OrderRole.GETORDER
) : (meta (active false))
```

So all we have to do is put the *call* statement there instead of the script. The statement takes the fully qualified name of the role-interface to which the call shall be delegated and an integer, that denotes the precised function of that role-interface. This allows you to have multiple operations in the same bean, each of which is identified by the parameter. And if you have not yet noticed: The integer in the JADL-call is the same as in the constant we defined in the role-interface.

Also we need to mention the last line of the JADL-code. After the closing bracket of the act, we put `:(meta (active false))`. This set's the *active*-MetaAttribute for this action to false. This MetaAttribute determines, whether an action can be selected for reaching a goal. Our getOrder-action will now be ignored by the SelectionBean. That's why we could set the effect to *true*. The only way of executing this planelement is by a direct call from a JADL-script. This can be done very easily:

```
(act createOrder
  (var ?ord:Order)
  (pre true)
  (eff (known (var ?b:bool) (att available ?ord ?b)))
  (script
    (var ?foo:bool)
    (seq
      (bind ?foo (fun printString "##### calling fillOrder..."))

      (fillOrder (var ?ord)) // call to (act getOrder)

      (bind ?foo (fun printString "##### order Filled..."))
    )
  )
)
```

The only really interesting line in this script is the `(fillOrder (var ?ord)))` statement. This calls the primitive planelement. The variable `?ord` is hereby handed over to the primitive planelement, as if it was a method-call with a parameter.

8.3.3 The OrderBean

Now we should implement the actual component. First of all, have a look at the JIAC-DOC for JIAC-API. There you will find a class called *DefaultApplicationBean*. We are going to extend that class. During implementation you are free to overwrite all the methods defined in this class. That is mainly for the statechanging-methods. You **have** to overwrite the following methods, if your component should do something useful:

- protected int execAction(int id, VarMap vars, Message reply)
- protected void goalFailure(int reference, Object context)
- protected void goalSuccess(int reference, Object context, VarMap vars)
- protected boolean testAction(int id)

But let's look at the obvious things first. The skeleton of our bean would look as follows:

```
package trading.component;

import trading.role.OrderRole;
import de.dailab.control.component.DefaultApplicationBean;
import de.dailab.cat.message.Message;
import de.dailab.kit.term.VarMap;


public class OrderBean
              extends DefaultApplicationBean
              implements OrderRole {

  public OrderBean() {
    super(OrderRole.class);
  }
}
```

The imports are pretty much standard, and we are going to need all of this classes sooner or later. Of course, we need to implement our role-interface, because that's why we created it. And finally you have to call the method `super(`*role-interface*`)` because your bean needs to register with the correct role-interface. If you do not call this method, the *DefaultApplicationBean* would register your bean as ApplicationRole. And that's not what we want.

Now let's get to the real business. First of all, we are going to have a look at the statechange-methods (changeToStopped, changeToActive, etc.). These are for the state-changes, we described a few sections earlier. Each component needs a *changeState*-method, that realizes the state-changes. This method is somewhat complex, so we made it easier for you. The DefaultApplicationBean implements the method, and calls the appropriate *changeTo???*-method. So if you want to do something in a special state - e.g. your bean changeing to the *active*) state - then you have to overwrite the *changeToActive*-method.

Note that the method has to return a boolean. `true` means that the state-change was successful, while `false` means, that an error occured. If one of the components cannot change its state correctly, the whole agent will refuse to change its state. So be cautious if you return false here. And that's all for the state-changing.

Now we will have a look at the other methods. We will start with the *testAction*-method. This method is called by the ExecutionBean before the actual call from the primitive planelement is executed. The agent tries to test, whether the action denoted in the call is legal or not. For this, the testAction-method should return true, for all call-parameters that are legal, and false for all other. In our case, the only legal call-parameter is our constant from the role: `GETORDER`. So the *testAction*-method may look like this:

```
protected boolean testAction(int id) {
  switch(id) {
    case GETORDER:
      return(true);

    default :
      return(false);
  }
}
```

If we didn't overwrite this method, then our call could never be successfully executed.

So keep this in your mind, when implementing a new component.

The most important thing though is the *execAction*-method. Originally the messages send to your component are delivered at the *execIntention*-method. But we decided to make things easier for you, so that you don't have to worry about the form of the messages. The execAction-method gives you the id of the requested action and the appropriate variable mapping as parameters. You also get a reference to the original message - but you will only need this, if you cannot answer directly.

A typical execAction-method would look like this:

```
protected int execAction(int id, VarMap vars, Message reply) {
  switch(id) {
    case GETORDER:
      KObjectRef ord = vars.evalValue("ord");
      System.out.println(">>>>> call in execAction with GETORDER");
      var.setValue("ord",ord); // set a new Value for the variable
      // do something
      return(RESULT_SUCCESS);
    default :
      return(RESULT_FAILURE);
  }
}
```

As you can see, you can get access to the variables in the VarMap of the JADL-action by calling the evalValue method on the VarMap. The argument for this method is the name of the variable without the *?*. Also you should always return `RESULT_SUCCESS` or `RESULT_FAILURE` at the end of this method, because the ExecutionBean wants to know whether the call was successful or not. RESULT_SUCCESS

and RESULT_FAILURE are constants that are defined in the ApplicationBean, and that are used for the results.

We hope, that you know how to create the GUI and get the user input. For the manipulation of the JADL-Objects we recomend the JIAC-DOC that describes the API. One thing we should explain though, is how to create goals in JAVA. There are two possible ways. You can either write down the goal in JADL, and let the JADL-parser do the work, or you can create the goal in JAVA-Objects and pass those objects to the agent. In each case you have to call the *addGoal*-method that is supplied by the ApplicationBean. This method takes either a StateGoal (see the API for details), or a String that is send to the JADL-parser. The signature looks like this: `addGoal(StateGoal goal,int ref, Serializable context)`. *ref* is an integer that will help you later to identify which goal you send away. It's the same with the context, but here you have the option of sending a whole JAVA-Object (must be Serializable) with the goal, and the object will be returned to you later - unchanged.

Whatever you do, sooner or later the agent will have processed your goal and have a result. This result will reach your component [2] either at the *goalSuccess* or at the *goalFailure*-method. In each case you will get the reference and the context back. If your goal was successful, you will also receive the new VarMap. We will not go into the details of what you can do in these last two methods. But you can surely think of something, e.g. tell the user at the GUI what happened with the goal.

[2]The result is always delivered to the issuer of a goal.

9. Appendix: Code examples and templates

9.1 templates

9.1.1 An agent or platform configuration file

Go back to the tutorial: 3.3

9.2 Concrete code examples

Go back to the tutorial: 3.3

9.2.1 An agent configuration file

```
# General Settings

  de.dailab.jiac.agent.permissions=stationary
  de.dailab.jiac.agent.permission.monitor

  # the state of the agent when starting
  de.dailab.jiac.agent.state=active

# Core Components

  de.dailab.jiac.agent.beans=\
    de.dailab.control.component.FactBean \
    de.dailab.control.component.GoalBean \
    de.dailab.control.component.SelectionBean \
    de.dailab.control.component.SchedulerBean \
    de.dailab.control.component.ExecutionBean \
```

```
    de.dailab.control.component.CommunicationBean \
    de.dailab.control.component.TimerBean \
    de.dailab.control.transport.component.TCPIPCommunicationBean \
    de.dailab.control.transport.component.JVMCommunicationBean \
    de.dailab.jiac.component.AMSUserBean \
    de.dailab.jiac.component.CommunityBean \
    de.dailab.control.debugging.component.DebuggerBean

# Plans

  de.dailab.control.role.PlanLibraryRole.plans=\
    de/dailab/jiac/knowledge/APService.know \
    de/dailab/jiac/knowledge/APUser.know \
    de/dailab/jiac/knowledge/AMSService.know \
    de/dailab/jiac/knowledge/AMSUser.know \
    de/dailab/jiac/knowledge/DFService.know \
    de/dailab/jiac/knowledge/DFUser.know \

# Debugging Settings

  de.dailab.cat.components.logOut
  de.dailab.cat.components.logLevel=f
```

9.2.2 A platform configuration file

```
# General Settings

  # the state of the agent when starting
  de.dailab.jiac.agent.state=active

  # address of this agent
  de.dailab.jiac.agent.guid=ams@tcpip://localhost:5555

  # access to all functionalities for this agent
  de.dailab.jiac.agent.permissions=all

  de.dailab.jiac.component.AMSProviderBean.name=trading
  de.dailab.jiac.component.AMSProviderBean.dynamicRegistration=false
  de.dailab.jiac.component.AMSProviderBean.mobility=false
  de.dailab.jiac.component.AMSProviderBean.systemExit=true
  de.dailab.jiac.component.AMSProviderBean.gui=true

# Core Components

  de.dailab.jiac.agent.beans=\
    de.dailab.control.component.FactBean \
    de.dailab.control.component.GoalBean \
    de.dailab.control.component.SelectionBean \
    de.dailab.control.component.SchedulerBean \
```

```
    de.dailab.control.component.ExecutionBean \
    de.dailab.control.component.CommunicationBean \
    de.dailab.control.component.TimerBean \
    de.dailab.control.transport.component.TCPIPCommunicationBean \
    de.dailab.control.transport.component.JVMCommunicationBean \
    de.dailab.jiac.component.AMSProviderBean \
    de.dailab.jiac.component.AMSUserBean \
    de.dailab.jiac.component.DFProviderBean \
    de.dailab.jiac.component.CommunityBean \
    de.dailab.control.debugging.component.DebuggerBean

# Plans

  de.dailab.control.role.PlanLibraryRole.plans=\
    de/dailab/jiac/knowledge/APService.know \
    de/dailab/jiac/knowledge/APProvider.know \
    de/dailab/jiac/knowledge/APUser.know \
    de/dailab/jiac/knowledge/AMSService.know \
    de/dailab/jiac/knowledge/AMSProvider.know \
    de/dailab/jiac/knowledge/AMSUser.know \
    de/dailab/jiac/knowledge/DFService.know \
    de/dailab/jiac/knowledge/DFProvider.know \
    de/dailab/jiac/knowledge/DFUser.know \
    de/dailab/kit/knowledge/Basics.know

  de.dailab.control.role.ServiceLibraryRole.services=\
    de/dailab/jiac/knowledge/APService.know \
    de/dailab/jiac/knowledge/AMSService.know \
    de/dailab/jiac/knowledge/DFService.know
# Debugging Settings
  de.dailab.cat.components.logOut
  de.dailab.cat.components.logLevel=f

# agents added to platform while initialization
  de.dailab.jiac.component.AMSProviderBean.agents=\
    Producer
  de.dailab.jiac.component.AMSProviderBean.agentsPropFile.Producer=\
    producer.agent

# socket settings for tcp/ip communication
  de.dailab.control.transport.component.TCPIPCommunicationBean.socketPort=5555

# DF settings
  de.dailab.jiac.component.DFProviderBean.storage=hashtable
```

9.2.3 Our ontology

Go back to the tutorial: 4.3

```
(package ontology

// This ontology describes a domain for trading products.
(ont Trading:Version_0

  // ********* Categories *********

  (cat Product

    (name string defined needed fixed)

    (price real needed)

  )

  (cat MultipleProduct

    (ext Product)

    (amount int needed)

  )

  (cat Offer

    (products Product[] needed)

  )

  (cat Stock

    (products MultipleProduct[])

    (full bool)

  )

  // ********* Functions and Comparisons *********

  (fun real getPrice Product)

  #begincode

    // getting the parameter
    KObject product = (KObject)a0.getValue();

    // extracting the product's price
    double price = product.getValue(Product_price);
```

```
    return price;

  #endcode

  (comp isCheaper Product Product)

  #begincode

    // getting the first parameter
    KObject product1 = (KObject)a0.getValue();

    // getting the second parameter
    KObject product2 = (KObject)a1.getValue();

    // return the comparisons result
    return product1.getValue(Product_price) < product2.getValue(Product_price);

  #endcode

)

)
```

9.2.4 An object file

Go back to the tutorial: 4.3

```
(package knowledge.startingKnowledge

  (ont ontology.Trading:Version_0)

    (obj notebook Product

      (name "IBM Thinkpad X30")

      (price 2500.00)

    )

    (obj sellersOffer Offer

      (products

        [Product:

          (obj Product

            (name "Apple IBook")
```

```
          (price 2000.00)
        )
        (obj Product
          (name "Acer Travelmate")
          (price 2400.00)
        )
      ]
    )
  )
  (obj Stocks Stock
    (products
      [MultipleProduct:
        (obj MultipleProduct
          (name "foo")
          (price 1.11)
          (amount 10)
        )
        (obj MultipleProduct
          (name "bar")
          (price 8.89)
          (amount 5)
      )
    ]
    )
    (full false)
```

```
    )
)
```

9.2.5 A goal file

Go back to the tutorial: 5.2.1

```
(ont ontology.Trading:Version_0)

(objref Stocks Stock)

(goal

    (att full Stocks true)

)
```

Index