

Welcome to the Jadex 3.0.0-RC16 Documentation

For API Documentation visit the [Active Components Website](#).

To quickly get an overview about what Jadex is about you can have a look at the [Quickstart Tutorial](#).

To see some example programs in action, you can try out some of the online [Java Webstart applications](#) or the [web applications](#).

Tutorials

[Quickstart Tutorial](#)

[AC Tutorial](#)

[BDI V3 Tutorial](#)

[BPMN Tutorial](#)

[BDI Tutorial](#)

Guides

[AC User Guide](#)

[BDI User Guide](#)

[Env User Guide](#)

[Android User Guide](#)

[AC Tool Guide](#)

Introduction

This tutorial provides a quick start for using Jadex. It is intended for people who just like to “jump in” and quickly getting things to run. Furthermore this tutorial provides many pointers to other documentation pages that you can follow if you want to learn a bit more about a certain topic.

Purpose of the Example Application

In this tutorial, a small distributed system will be built in which time clients can subscribe to time servers for continuously receiving the current time of the server. Therefore time user components will search for available time provider components and subscribe to all providers they can find (see Fig.1a). The time providers remember the subscribed time user components. They then periodically (e.g. every 5 seconds) send their current time to all subscribed time users components (see Fig.1b).

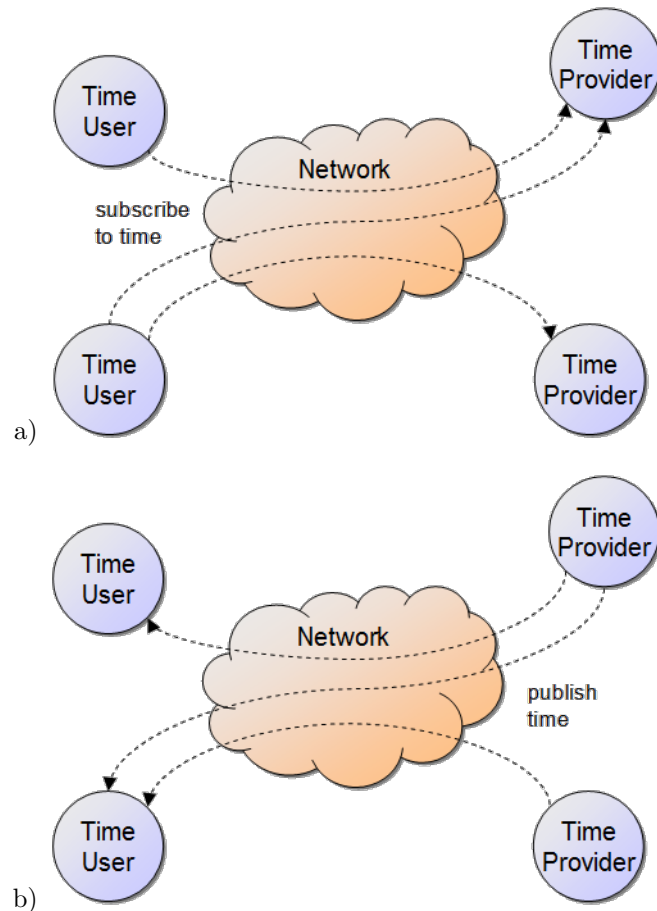


Figure 1: a) Time users subscribing to time providers, b) time providers publishing their current time to time users

Although this application is quite simple, it contains several common challenges regarding programming distributed systems. In the following it will be shortly sketched, how the Jadex Active Components middleware helps coping with these challenges:

Challenge 1: Discovery of distributed components

The time user components need to discover the available time provider components. In Jadex discovery is supported by a combination of two mechanisms. First, platform awareness automatically discovers all available Jadex platforms in local networks and potentially across the whole internet. Second, the service search potentially traverses all known platforms looking for the desired service and thus potentially finds any matching service available somewhere on the Internet.

Challenge 2: Components with internal behavior

The time providers need to periodically send out time values. In Jadex, components may have internal behavior ranging from purely reactive components to simple task-oriented or even complex intelligent agents. E.g. the time provider component has a periodic task for sending out time values.

Challenge 3: Designing communication protocols

The communication between time user and time provider needs to be defined. Similar to discovery, in Jadex, communication is dealt with on the platform and on the component level. A set of message transports assures that platforms can communicate in local networks as well as across the Internet. On the component level, interaction typically uses object-oriented interfaces, e.g. the use of services through remote method invocation (RMI). Furthermore, Jadex comes with ready to use implementations for commonly used interaction patterns like publish/subscribe.

Challenge 4: Handling partial failures

Due to node or network failures, time user components may not always be able to correctly unsubscribe at the time providers. Thus time providers should automatically unsubscribe clients, which are no longer responding. Otherwise time providers would accumulate broken clients and quickly run into memory leaks. Using the available interaction pattern for publish/subscribe, Jadex will automatically detect failed clients and inform the time provider to remove the client.

Challenge 5: Security

Providing and accessing services across the Internet involves many security issues. In Jadex, by default only trusted platform may invoke services of each other, therefore running a Jadex platform is safe by default. More fine-grained treatment of security issues is supported by security annotations. These annotations can be placed alongside the component code and allows a clean separation between component functionality and non-functional aspects like security.

Application Architecture

The architecture of the system is shown in Fig.2 as a UML class diagram. The time service interface is the central aspect of the design. The interface is used by the time user agent and it is implemented by the time provider agent. Time user and time provider do not know each other directly and only communicate through the time service interface. In the following three chapters each of the three elements of the architecture will be explained in detail.

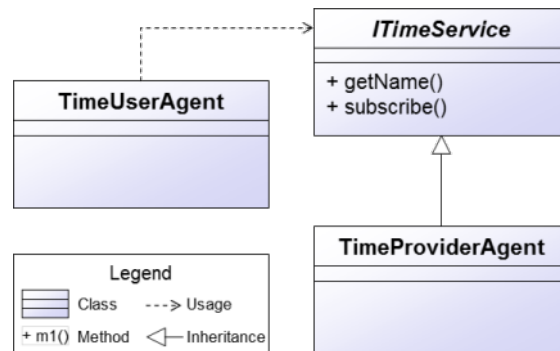


Figure 1: 01 Introduction@timearch.png

Figure 2: UML class diagram of the time user / time provider system architecture

Time Service Interface

This chapter sketches the Java project setup and explains how to define a Jadex service interface.

Prerequisites

This quickstart tutorial assumes that you are familiar with Java and your favourite IDE. Therefore, you should know how to set up a project to start developing Java applications, which use Jadex. If you need additional guidance for this, please have a look at the tutorial.

For your Java project please make sure **not** to include the jadex-application-xyz jars as they already contain the quickstart tutorial classes. If you include these, Java might ignore your own classes and only use the ones from the jar. If you set up your project using the maven-based Jadex example project, the appropriate jars will be automatically included and excluded as needed.

The Time Service Interface

Create Java file *ITimeService.java* in the package *jadex.micro.quickstart* and paste the contents as shown below. The relevant aspects of this file are explained in the following subsections.

```
package jadex.micro.quickstart;

import jadex.bridge.service.annotation.Security;
import jadex.commons.future.ISubscriptionIntermediateFuture;

import java.util.Date;

/**
 * Simple service to publish the local system time.
 */
@Security(Security.UNRESTRICTED)
public interface ITimeService
{
    /**
     * Get the name of the platform, where the time service runs.
     */
    public String getName();

    /**
     * Subscribe to the time service.
     */
    public ISubscriptionIntermediateFuture<Date> subscribe();
}
```

Figure 1: Time service interface in Java

The Name and Package of the Interface

In Jadex, the fully qualified name of a service interface is used for service discovery. Therefore when you implement a time user component to search for your *ITimeService*, Jadex will discover all components worldwide that offer a service of type *jadex.micro.quickstart.ITimeService*. Therefore, if you make sure to use the interface and package name as shown, you might be able to find other peoples time provider components. E.g. for testing purposes there should be a time provider running in our infrastructure.

The *getName()* Method

Normally, all service methods are potentially remote calls. Therefore it is always a good idea to use future types as return values. Futures avoid that the caller is blocked during the service processing and add further support e.g. for dealing with timeouts. One notable exception is when service operations only provide constant values. In the time service example, the name is considered a value that does not change during the lifetime of a service instance. This means that each separate instance of the time service may have a different name, but this name is constant with respect to each instance. Therefore the name can be transmitted as part of the service reference and can be provided without an additional remote call. To indicate that the name is constant the *getName()* method is declared with a String return value.

The *subscribe()* Method

The *subscribe()* method signature captures the interaction between time provider and time user. It uses the subscription intermediate future as a return type, which denotes a subscription semantics for the interaction as shown in the diagram below.

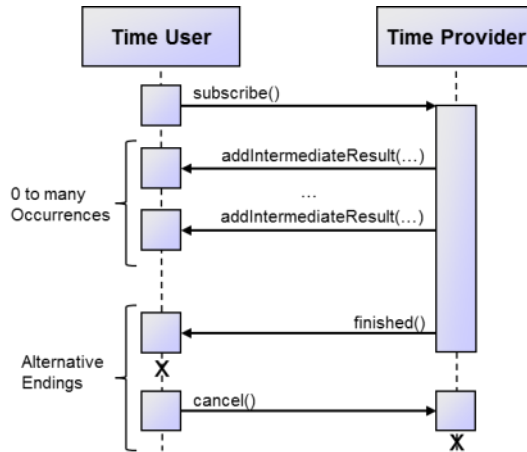


Figure 2: Interaction diagram of a subscription future

The interaction starts by the user calling the *subscribe()* method on the service of the provider. While the interaction is active, the time provider may post time values in arbitrary intervals. These time values are transmitted as intermediate results of the future and immediately become available to the time user. At any point in time, either of both sides may decide to end the interaction. The time provider may do this by setting the future to *finished* to indicate that no more results will be published. The time user on the other hand may decide that it

no longer wants to receive time values from the time provider and thus may call the `cancel()` method on the future.

As you can see, using subscription futures allows capturing a complex interaction semantics in a single method signature. Using generics, the shown method signature further demands that the intermediate results must be of type `Date`.

Security Issues

Jadex platforms can potentially detect any other Jadex platforms on the internet using several awareness mechanisms. This means that a time service hosted on your platform may be found and invoked by some other Jadex platform that you are not aware of. To avoid security issues due to being visible and accessible to other platforms, in Jadex all services are restricted by default. This would mean that only components on your platform can invoke your time service. This behavior is advantageous for getting used to Jadex, because you can just start and implement your own services without having to consider security issues. All your services are available for local testing but shielded from outside platforms.

For our time service example, we consider that the service does not pose any security threats, because the service does not change the local system and provides no sensitive information. Therefore, we want the service to be accessible to outside platforms as well. For example you should be able to test your user agent with one of our time provider components hosted in our infrastructure. To enable unrestricted access to the service, the `@Security` annotation is specified. More information about security and restricted and unrestricted services can be found in the Tutorial and the user guide.

The Time User Agent

This chapter shows how to discover and use the time service.

Agent Implementation

Create Java file *TimeUserAgent.java* in the package *jadex.micro.quickstart* and paste the contents as shown below.

```
package jadex.micro.quickstart;

import jadex.commons.future.IIntermediateFuture;
import jadex.commons.future.ISubscriptionIntermediateFuture;
import jadex.commons.future.IntermediateDefaultResultListener;
import jadex.micro.annotation.Agent;
```

```

import jadex.micro.annotation.AgentBody;
import jadex.micro.annotation.AgentService;
import jadex.micro.annotation.Binding;
import jadex.micro.annotation.RequiredService;
import jadex.micro.annotation.RequiredServices;

import java.util.Date;

/**
 * Simple agent that uses globally available time services.
 */
@Agent
@RequiredServices(@RequiredService(name="timeservices", type=ITimeService.class, multiple=true))
public class TimeUserAgent
{
    //----- attributes -----

    /** The time services are searched and set at agent startup. */
    @AgentService
    private IIntermediateFuture<ITimeService> timeservices;

    //----- methods -----

    /**
     * This method is called after agent startup.
     */
    @AgentBody
    public void body()
    {
        // Subscribe to all found time services.
        timeservices.addResultListener(new IntermediateDefaultResultListener<ITimeService>()
        {
            public void intermediateResultAvailable(final ITimeService timeservice)
            {
                ISubscriptionIntermediateFuture<Date> subscription = timeservice.subscribe();
                subscription.addResultListener(new IntermediateDefaultResultListener<Date>()
                {
                    /**
                     * This method gets called for each received time submission.
                     */
                    public void intermediateResultAvailable(Date time)
                    {
                        System.out.println("New time received from "+timeservice.getName()+" : "+time);
                    }
                });
            }
        });
    }
}

```



```
    });  
  }  
}
```

Execute the Agent

Start the Jadex platform with its main class `jadex.base.Starter`. Add the directory containing the compiled classes of your agent to the Jadex Control Center (JCC). Select the `TimeUserAgent.class` and click *Start*.

In case there are any time services online, you should see their time printed to the console in periodic updates. If no time service is found after 30 seconds, a `jadex.bridge.service.search.ServiceNotFoundException` is printed.

See AC Tutorial.Chapter 02: Installation for details on setting up a launch configuration and starting agents in the JCC.

The details of the agent are explained in the following subsections.

Class Name and @Agent Annotation

To identify the class as an agent that can be started on the platform, the `@Agent` annotation is used. Furthermore, to speed up the scanning for startable agents, all agent classes must end with 'Agent' by convention. Before the 'Agent' part, any valid Java identifier can be used.

The Required Service Declaration and Injection

A Jadex agent may use services provided by other agents. An agent might search for arbitrary services dynamically, but it also can declare required services as part of its public interface. A declaration of a required service is advantageous, because it makes the dependencies of an agent more explicit. Furthermore, the declarative specification of a required service allows separating details, such as service binding, from the agent implementation.

The time user agent declares the usage of the `ITimeService` by the `@RequiredService` annotation. The annotation here states, that the agent is interested in multiple instances of the service at once (`multiple=true`) and that all platforms world wide should be searched for available services (`binding=@Binding(scope=Binding.SCOPE_GLOBAL)`).

The required service declaration is given the name `timeservices` and is used for the corresponding field of the class. The `@AgentService` annotation for the field states that at startup, the field should be injected with the found required services as given in the required service declaration with the same name. Here a

type of `IntermediateFuture` is used to receive each service immediately when found as shown in the agent body.

The Agent Body

The agent body is executed after the agent is started. Using the `@AgentBody` annotation a method can be designated as agent body.

The body first adds a listener to the `timeservices` field, to receive updates whenever a `timeservice` is found. For each found service, the `intermediateResultAvailable()` method is called. In this method, the agent subscribes to the found time service by adding another listener. This listener is informed about each new time notified by the specific time service.

Time Provider

Summary

how challenges are met in concrete example

Introduction

The Active Components project aims at providing programming and execution facilities for distributed and concurrent systems. The general idea is to consider systems to be composed of components acting as service providers and consumers. Hence, it is very similar to the Service Component Architecture (SCA) approach and extends it in the direction of agents. In contrast to SCA, components are always active entities, i.e. they possess autonomy with respect to what they do and when they perform actions making them akin to agents. In contrast to agents communication is preferably done using service invocations.

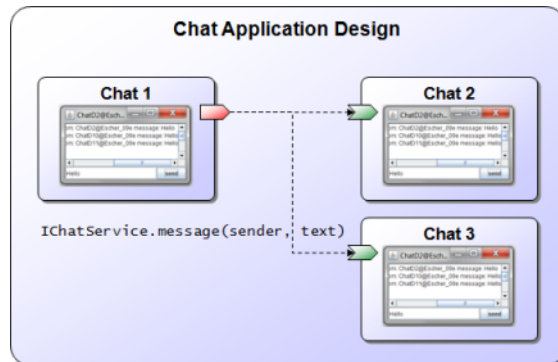
This tutorial provides step-by-step instructions to learn how to use the Jadex active components infrastructure. You will learn how to install and use the infrastructure (i.e. the Jadex platform) to execute components and how to compose systems from components. In particular, the following topics are covered in the upcoming chapters:

- Chapter 02 Installation describes the steps necessary to install and run the Jadex Active Components runtime.
- Chapter 03 Active Components illustrates how to program simple components.

- Chapter 04 Required Services describes how to fetch and use services of other components.
- Chapter 05 Provided Services explains how to equip a component with services.
- Chapter 06 Composition describes how to compose a component from subcomponents.
- Chapter 07 External Access describes how to attach tightly coupled functionality, e.g. for GUIs.

Application Context

In this tutorial a simple chat application will be implemented. The chat application can be used to send messages to other users. This base functionality will be extended in the different exercises, but it is not our goal to build up a solution that combines all the extensions, because this would lead to difficulties concerning the complexity of the application. Instead this tutorial will concentrate on setting up simple components that explain the Jadex concepts step by step.



Conceptual design of the Chat application

The figure above shows the conceptual design of the chat application. On different computers, so called 'Chat' components are running, each of which provides a graphical interface to a local user. When a user enters a new chat message (e.g. in 'Chat 1'), the message gets forwarded to all chat components in the network (e.g. 'Chat 2' and 'Chat 3').

We will come back to this design in 05 Provided Services, where we put all the pieces together that allow us building an initial working version of this chat application.

Installation

In this chapter, you will learn how to install eclipse for developing with Jadex. Therefore, you will find instructions on setting up a proper eclipse working environment for programming and executing Jadex applications. If you use a different IDE the instructions and screenshots below should still be helpful for setting up your IDE accordingly.

Note that this tutorial is not a replacement for the existing eclipse documentation. If you have questions about using eclipse try also looking for an answer at the <http://www.eclipse.org/documentation/> site.

Prerequisites

- Download and install a recent Java environment from <http://java.sun.com/javase/downloads> (if not already present).
- Download and install a suitable eclipse distribution (≥ 3.5) from <http://www.eclipse.org/downloads/> (if not already present). The following has been tested with the ‘Eclipse IDE for Java EE Developers’ package.
- (*Standard setup: A1-A3*) Download the latest Jadex distribution .zip from <https://www.activecomponents.org/bin/view/Download/Distributions> and unpack it to a place of your choice.
- (*Alternative setup: A4*) Download the `jadex-example-project.zip` from <https://activecomponents.org/bin/view/Download/Distributions> and unpack it to a place of your choice.

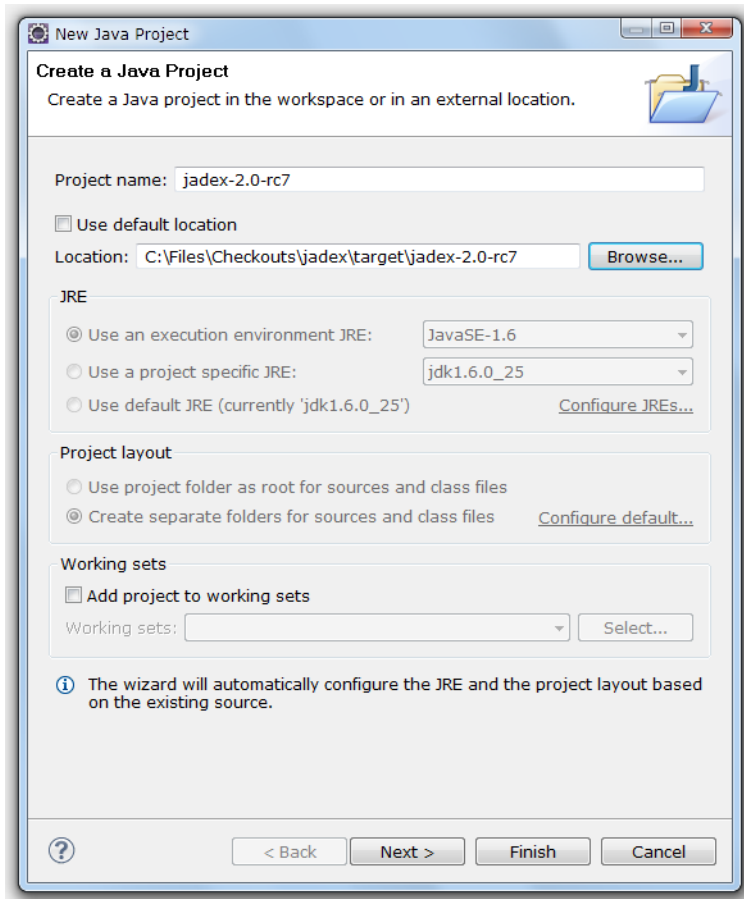
Before starting with the exercises, you need to decide if you’d like to use the standard setup as described in exercises A1-A3 or the alternative setup with the Maven build tool as described in exercise A4. For inexperienced users, it is recommended to follow the standard setup, were most steps are performed manually and are thus explained in the exercises.

Exercise A1 - Eclipse Environment Setup

In this lesson you will set up an initial eclipse environment that will be used in the following lessons. Please follow the instructions carefully and compare your setup to the screenshots to verify that everything went fine. The steps below describe how to develop Jadex applications with a basic eclipse installation. For advanced users it may be more convenient to use eclipse (or another IDE) in conjunction with Maven . In the Jadex distribution, there is a ‘`jadex-example-project.zip`’ that conatins a basic Maven project for Jadex including a ‘`readme.txt`’ with the alternative setup instructions.

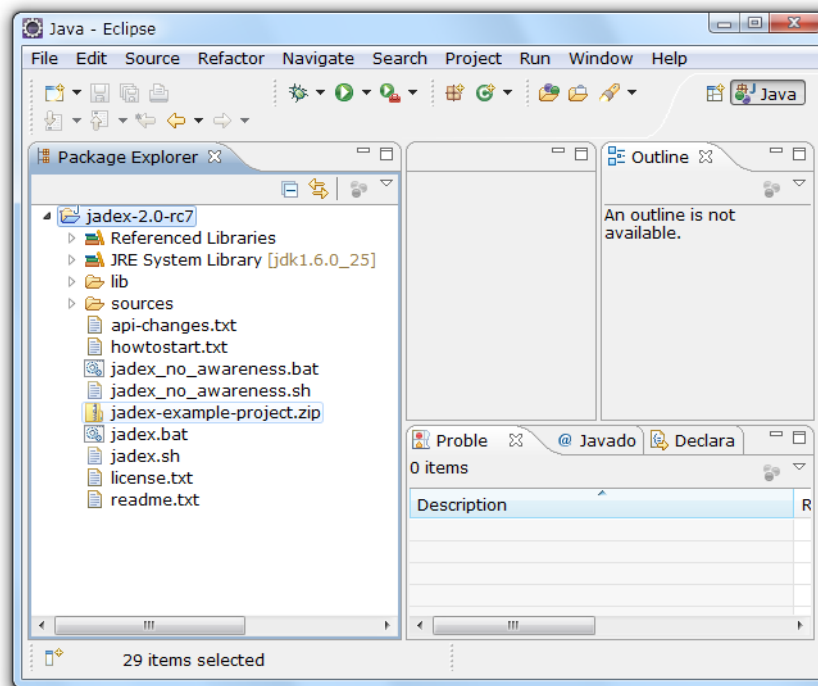
Start eclipse. Start the ‘New Java Project’ wizard, disable the ‘Use default location’ checkbox and browse to the directory, where you unpacked the Jadex

distribution. Note that the name of the directory might slightly differ due to changing Jadex version numbers.



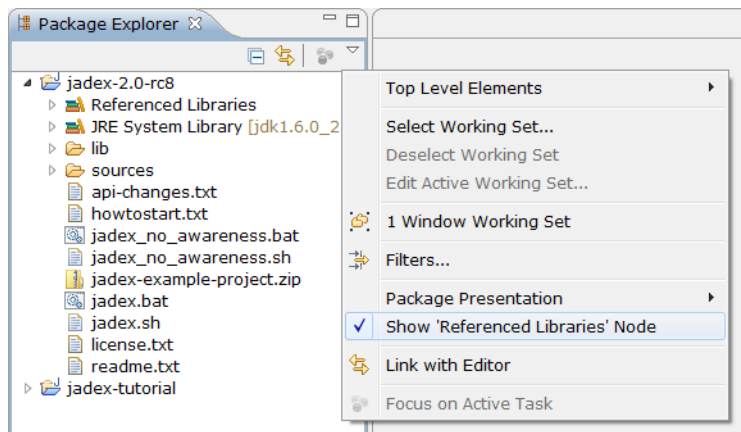
Create Java project in eclipse

Click 'Finish' - the project will be created. Your project should now look like the picture below.



Final Jadex eclipse project layout

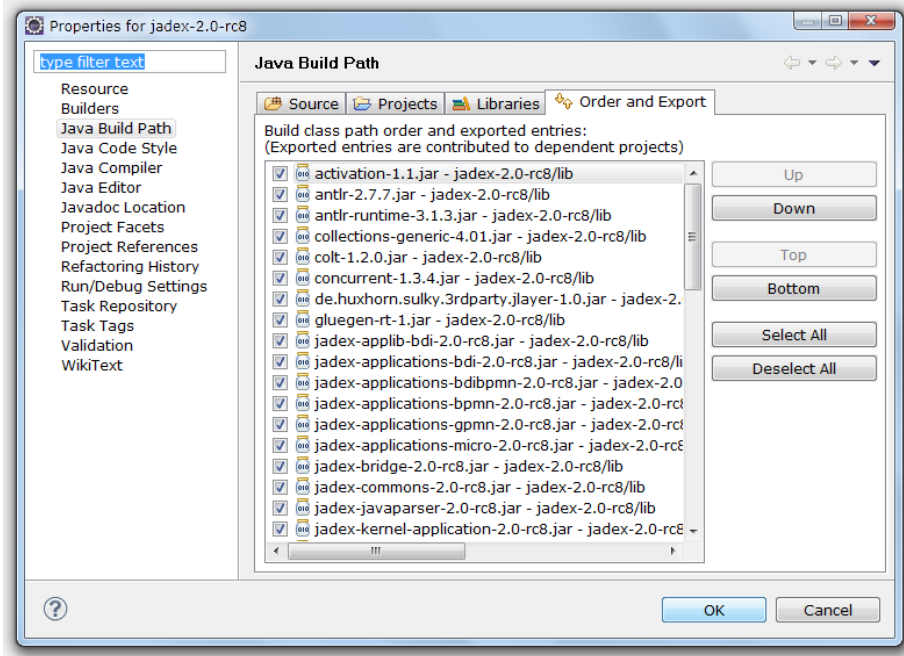
Note that depending on your eclipse configuration, the ‘Referenced Libraries’ node might be hidden and instead the jars from the lib folder will be displayed as direct child of the project. If you wish, you can change the appearance in the explorer menu by (de)selecting the “Show ‘Referenced Libraries’ Node” checkbox.



Show/hide the ‘Referenced Libraries’ node

This project will be used as a basis for your own development projects. To make the Jadex libraries accessible to other projects it is necessary in eclipse to

export them. Right-click on the project, choose 'Build Path -> Configure Build Path...'. Go to the 'Order and Export' tab, choose 'Select All' and hit 'OK'.



Export jars from build path

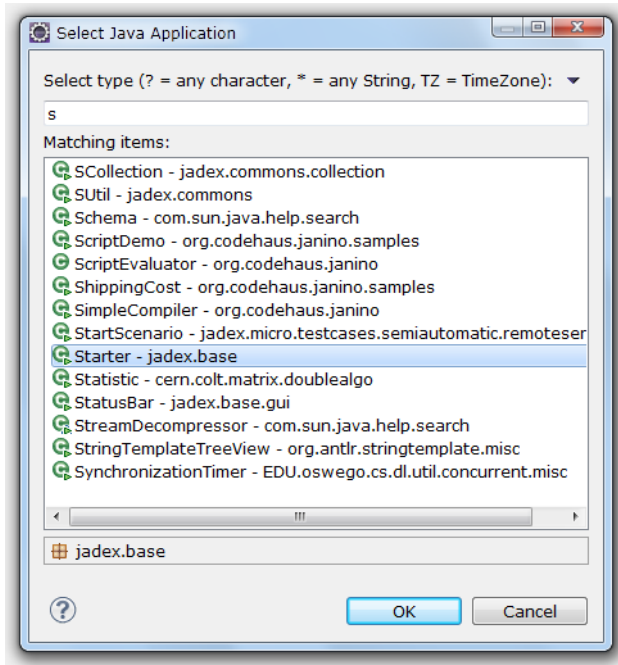
For further simplifying later development, you should attach the sources to the Jadex jars, as this will enable eclipse to provide better context sensitive help on method signatures, etc. Open the 'Referenced Libraries' node (if shown) and the 'jadex-applications-micro-2.0-rc7.jar'. In the 'jadex.micro.examples.helloworld' package double click on the 'HelloWorldAgent.class' file. The file will be opened, showing the reverse engineered byte code. Click the 'Attach Source...' button, choose 'Workspace...' and select the 'sources.zip' file contained in the Jadex project. Click 'OK'. The Java source of the hello world agent should now be displayed. Repeat these steps for the other Jadex jars. It is recommended for this tutorial to add the sources at least to the 'jadex-bridge', 'jadex-commons', 'jadex-kernel-micro' jars.

Exercise A2 - Running Example Applications

In this lesson we will create a launch configuration to start the Jadex platform. To see that everything works, we will execute some example applications that are distributed with the Jadex package.

Eclipse Launch Configuration

As we imported the Jadex distribution in lesson A1, we are almost ready to go. Right-click on the Jadex eclipse project and choose 'Run As' > 'Java Application' from the popup menu. Eclipse will search for main types, i.e. startable Java classes (with a *main()* method). Select *Starter* from the package *jadex.base* in the appearing dialog (just entering an 's' will probably be enough to find the starter class easily in the list).



Select main class for starting Jadex

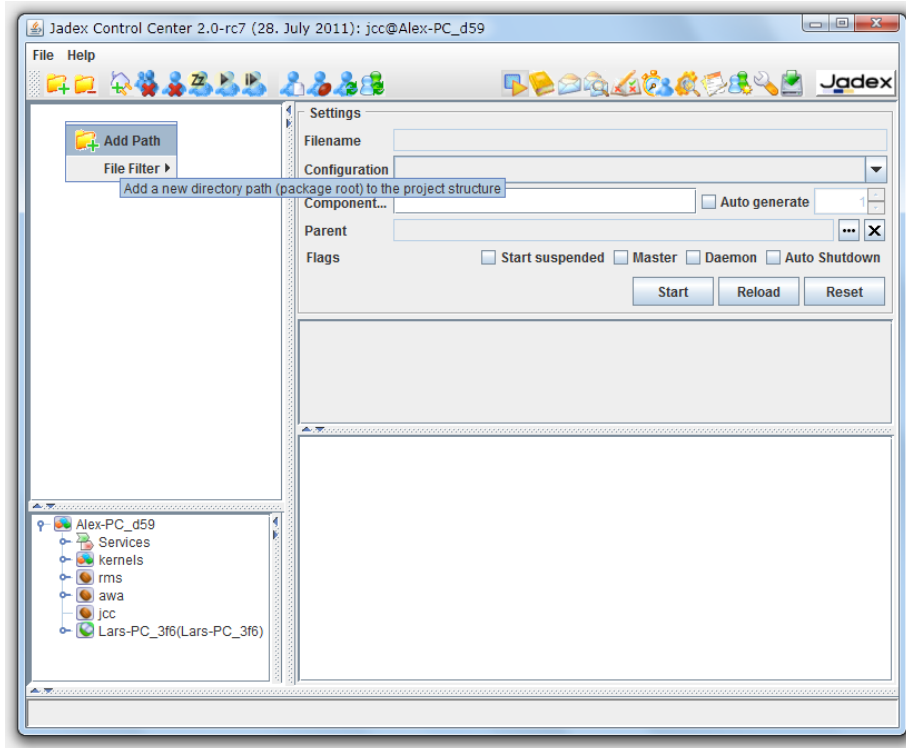
Hit 'Ok' to start the Jadex platform. The next time you want to start the platform, you do not have to repeat the above steps. Just choose the 'Starter' entry from the run history, which eclipse generates automatically.

Selecting and Starting a Component

If you managed to successfully start the Jadex platform, the Jadex control center (JCC) window will appear (see below). The JCC is a management and debugging interface for the Jadex platform and the components that run on it. The JCC has its own way (distinct from eclipse) of loading and saving settings. The reason for this separation is to allow using Jadex without being bound to a specific IDE (like eclipse).

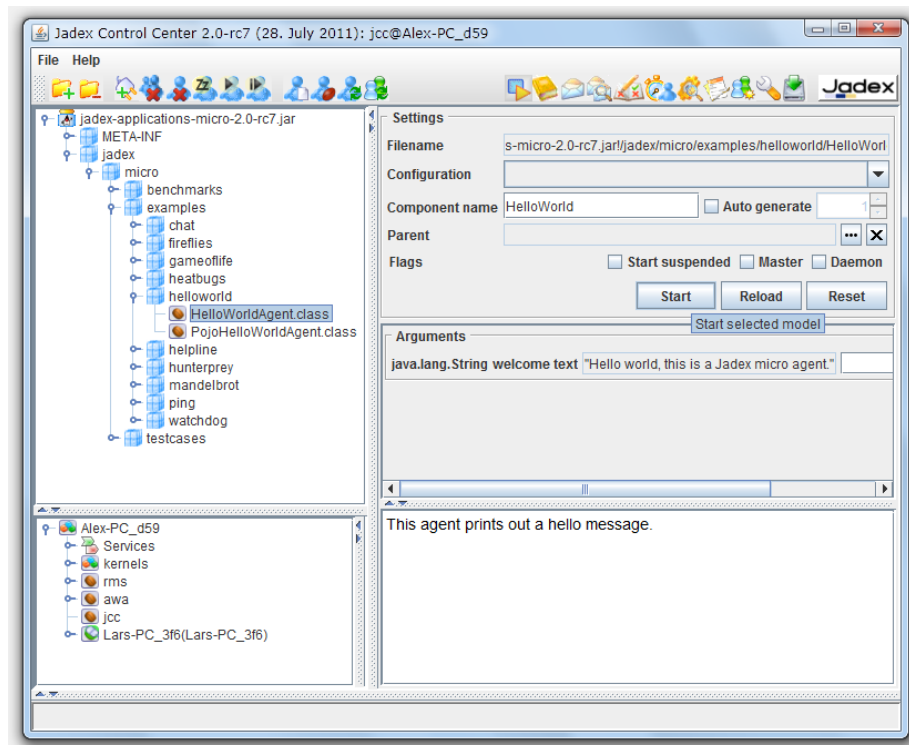
To execute any applications you need to add the corresponding path to the JCC

project. We will now set up the platform for starting some examples. Right-click in the upper left area (called the model explorer, as it is used to browse for models of e.g. processes) and choose ‘Add Path’.



Add path in JCC

A file requester appears that should initially present the directory, where you unpacked the Jadex distribution. Open the *lib* directory and select the file *jadex-applications-micro-2.0-rc7.jar*. Note that depending on your Jadex version the ‘2.0-rc7’ part might slightly differ in your setting. You can now unfold the contents of the jar file and browse to the helloworld example in the *jadex/examples* package. After you selected the *HelloWorldAgent.class* in the tree, you can start the process by clicking ‘Start’.



Start a component

The component will be executed, thereby printing some messages to the (eclipse) console.

Saving JCC and Platform Settings

As you probably do not want to add the jar file again, each time you start the Jadex platform, you should save the current settings. From the 'File' menu choose 'Save Settings'. The settings will be stored in two files in the current directory. The 'jcc.settings.xml' contains GUI settings like the window position. Another '*.settings.xml' file will be created named after the host name. It contains the platform settings (e.g. included jar files). The platform and JCC settings will automatically be loaded when the platform is started the next time.

Execute Example Applications

Execute some other examples, e.g. 'heatbugs' or 'mandelbrot'. Many examples involve more than one component and are typically launched by selecting and starting the '*.application.xml' component, which automatically starts all components of the application.

You can also load the other 'jadex-applications-*-2.0-rc7.jar' files (e.g. BDI or BPMN) and try the examples included there. These types of components are described in separate tutorials. This tutorial limits itself to XML components and Java-based Micro agents for simplicity.

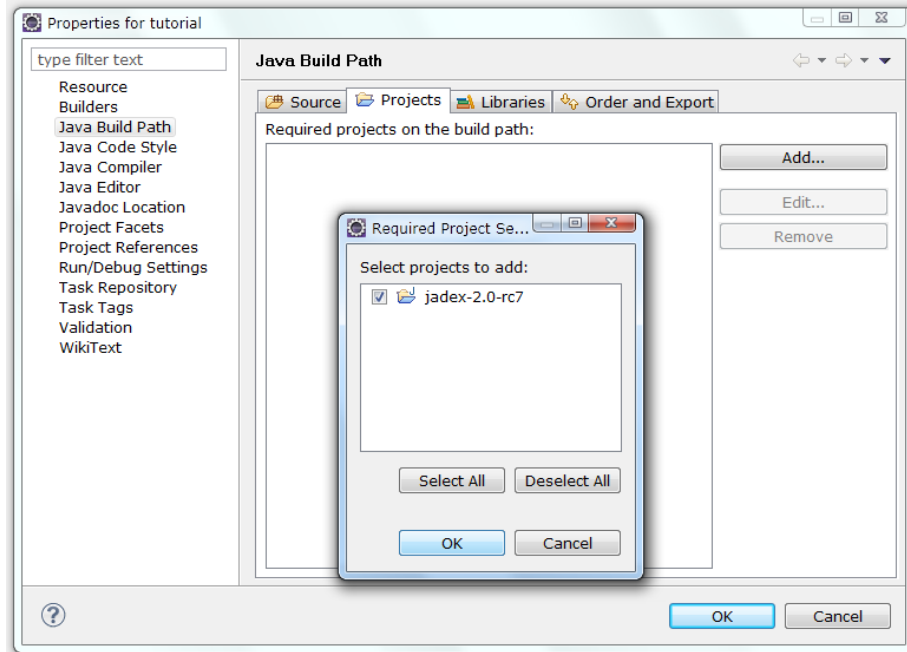
Exercise A3 - Development Project Setup

In this lesson you will set up your own an eclipse project for all the files that you create in this tutorial.

Eclipse Project Setup

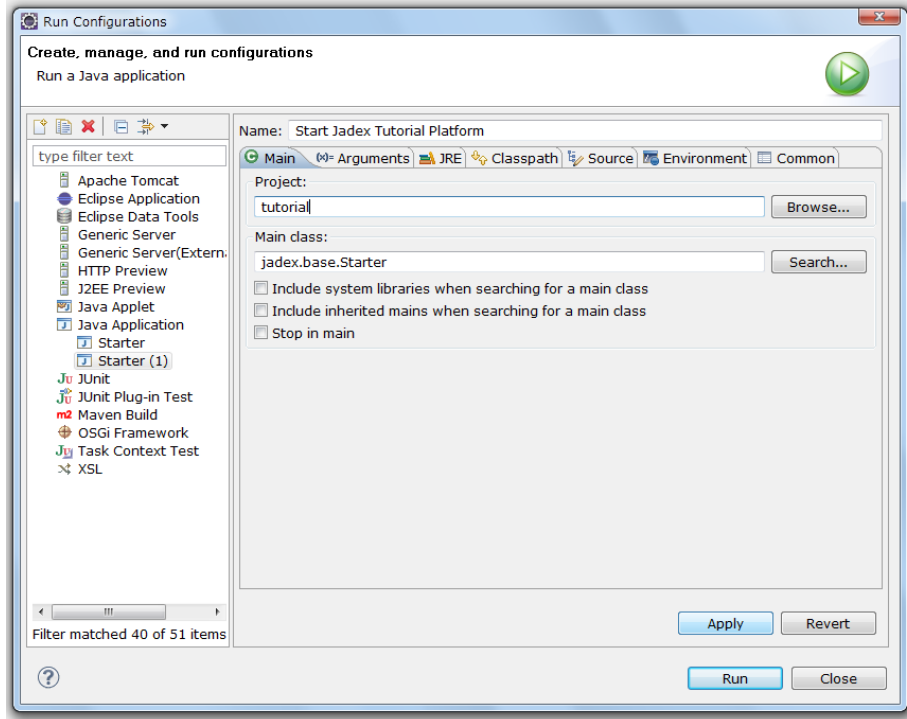
Create a new Java project in eclipse. Choose a name as you like (e.g. 'tutorial'). Create a Java package in this project. The following examples use 'tutorial' as a package name. You can find sample solution files in the Jadex distribution in the *jadex.micro.tutorial* package.

Add a reference to the project with the Jadex distribution. To do so, right-click on the tutorial project and select 'Build Path -> Configure Build Path...'. In the 'Projects' tab click 'Add...', check the 'jadex-2.0-rc7' checkbox and hit 'OK'. Close the build path window by hitting 'OK' again.



Add project dependency

Now change the launch configuration to start from the newly created project. Therefore, choose the 'Run -> Run Configurations... ' menu. Right-click the 'Starter' configuration and choose 'Duplicate'. Change 'Starter (1)' to a more telling name of your choice (e.g. 'Start Jadex Tutorial Platform'). In the 'Main' tab select 'Browse...' besides the project textfield and choose your 'tutorial' project. Select 'Run' to save the launch configuration and start the platform.



Create Jadex launch configuration in eclipse

Jadex Setup

When starting the Jadex platform using your tutorial launch configuration, the JCC window will appear with the settings belonging to the tutorial project (i.e. nothing yet). Right-click in the model explorer and choose 'Add Path'. Browse to your eclipse workspace and select the 'bin' folder from the eclipse project that you created in the beginning of this lesson. When you unfold the contents, you should find the package(s) that you created. Save the settings. When you refresh the project content in eclipse, you should see the settings files appearing in the package explorer. The following screenshot shows how your setup should look like in eclipse (left) and in the JCC (right).

Layout of Jadex tutorial project

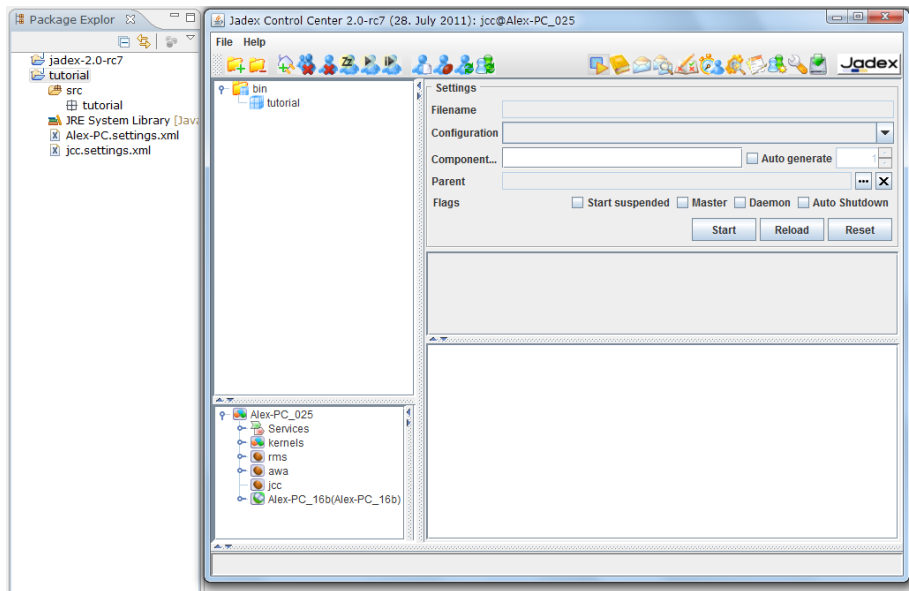


Figure 2: 02 Installation@eclipsejctutorialproject.png

You are now ready to implement and execute your own components in Jadex. The following exercise represents an alternative installation process using Maven, which is optional and can be skipped. The next chapter will introduce the active components concept in more detail and show how simple components are built in XML or Java.

Exercise A4 - Alternative Project Setup Using Maven

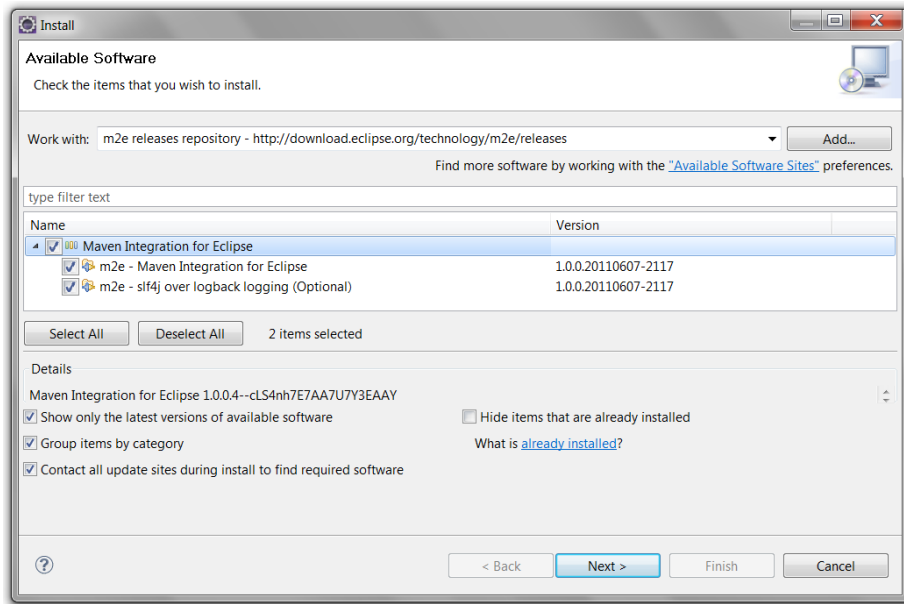
This exercise describes an alternative way of setting up a Jadex project for development in eclipse using the Maven build tool. The exercise replaces all steps from the previous exercises (A1-A3). It does not require prior knowledge of Maven for initial setup, but you should be willing to learn about using Maven later, when you need to do custom changes to the project setup. Furthermore, if you just started using Jadex then you should read A2 and A3 as well, as these exercises contain helpful usage information.

Prerequisites

- Download the `jadex-example-project.zip` from <http://jadex-agents.informatik.uni-hamburg.de/xwiki/bin/view/Download/Available+Packages> and unpack it to a place of your choice.

Installing the M2Eclipse plugin

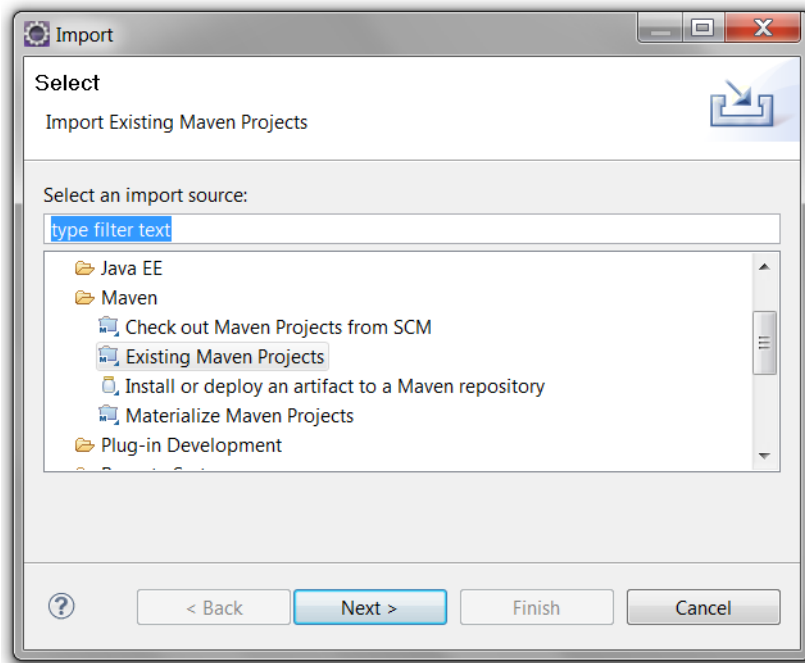
- Select “Help -> Install New Software...” from the menu.
- Enter the plugin URL <http://download.eclipse.org/technology/m2e/releases> (for newer eclipse versions, the Maven plugin is available from the default, e.g. Juno, eclipse update site)
- Select “Maven Integration for Eclipse” and install the plugin.



Install the m2eclipse plugin

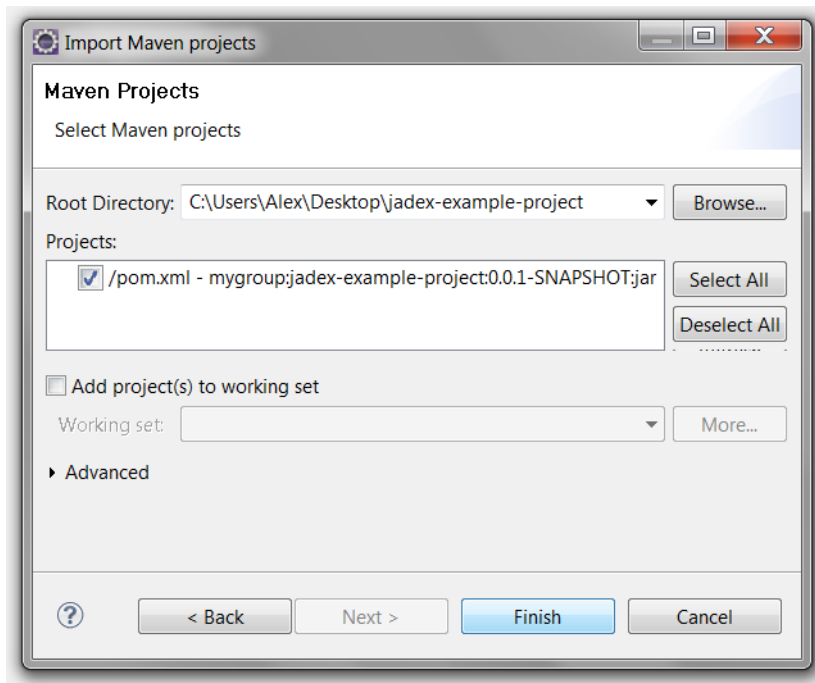
Importing the Jadex example project

- Use “File -> Import -> Maven / Existing Maven Projects” and choose “Next”.



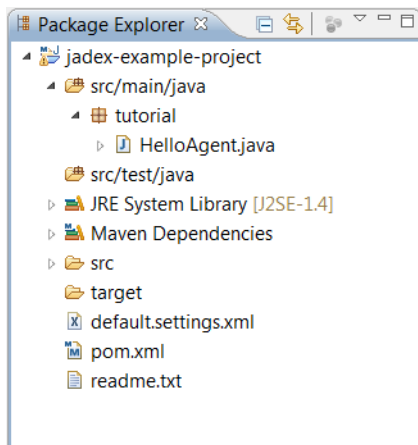
Import maven project (1)

- “Browse...” to the directory where you unzipped the jadex-example-project.zip. Maven will detect the project, which is described in the ‘pom.xml’ file.



Import maven project (2)

- Click “Finish”. Maven will import the project and start the build process thereby download the necessary Jadex libraries from the web.



Imported example project in eclipse

Starting the Jadex platform

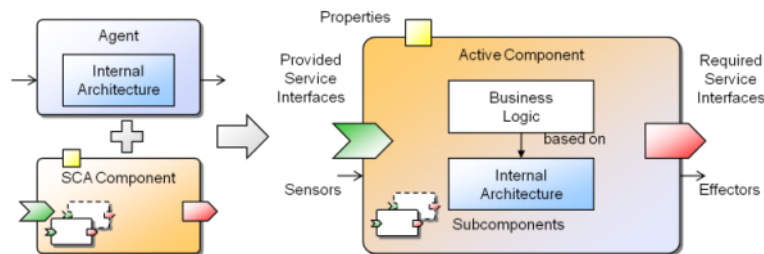
- Right-click on the imported project and choose “Run As” -> “Java Application”. Select the ‘Starter’ class from package ‘jadex.base’. Click “Run” and the Jadex platform should start.
- In the JCC, the ‘HelloAgent’ should be already selected. Click “Start” to start the agent and check the console for the output “Hello world!”. (cf. Exercise A2)
- Eclipse remembers the launch configuration. Therefore in the future, you can simply select the “Starter” configuration from the run history to start the platform.

You are now ready to continue with the tutorial. If you later wish to setup your own project you can unzip the `jadex-example-project.zip` again to a different location. Change the following line by replacing ‘`jadex-example-project`’ with a project name of your choice. Afterwards import the project in eclipse as described above.

```
<artifactId>jadex-example-project</artifactId>
```

Active Components

The active component approach brings together agents, services and components in order to build a worldview that is helpful for modelling and programming various classes of distributed systems. Recently, with the service component architecture (SCA), a new software engineering approach has been proposed by several major industry vendors including IBM, Oracle and TIBCO. SCA combines in a natural way the service oriented architecture (SOA) with component orientation by introducing SCA components communicating via services. Active components build on SCA and extend it in the direction of software agents. The general idea is to transform passive SCA components into autonomously acting service providers and consumers in order to better reflect real world scenarios which are composed of various active stakeholders. In the figure below an overview of the synthesis of SCA and agents to active components is shown.



Active Component Structure

The figure presents on the right hand side the structure of an active component. It yields from conceptually merging an agent with an SCA component (shown at the left hand side). An agent is considered here as an autonomous entity that is perceiving its environment using sensors and can influence it by its effectors. The behavior of the agent depends on its internal reasoning capabilities ranging from rather simple reflex to intelligent goal-directed decision procedures. The underlying reasoning mechanism of an agent is described as an agent architecture and determines also the way an agent is programmed. On the other side an SCA component is a passive entity that has clearly defined dependencies with its environment. Similar to other component models these dependencies are described using required and provided services, i.e. services that a component needs to consume from other components for its functioning and services that it provides to others. Furthermore, the SCA component model is hierarchical meaning that a component can be composed of an arbitrary number of subcomponents. Connections between subcomponents and a parent component are established by service relationships, i.e. connection their required and provided service ports. Configuration of SCA components is done using so called properties, which allow values being provided at startup of components for predefined component attributes. The synthesis of both conceptual approaches is done by keeping all of the aforementioned key characteristics of agents and SCA components. On the one hand, from an agent-oriented point of view the new SCA properties lead to enhanced software engineering capabilities as hierarchical agent composition and service based interactions become possible. On the other hand, from an SCA perspective internal agent architectures enhance the way how component functionality can be described and allow reactive as well as proactive behavior.

Exercise B1 - XML Component Definition

The figure above has highlighted the important properties of an active component from a black-box perspective, i.e. for now we do not care about its internal behavior definition. Instead, in this lecture we will learn how a component can be defined and started in Jadex. For this purpose we resort to a component xml description, which follows the XML-schema definition for active components. It basically contains the elements shown above, but as first step we will only use the most basic aspects:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Chat component. -->
<componenttype xmlns="http://jadex.sourceforge.net/jadex"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://jadex.sourceforge.net/jadex
    http://jadex.sourceforge.net/jadex-component-2.1.xsd"
  name="ChatB1" package="tutorial">
```

`</componenttype>`

Basic chat component definition

Open a source code editor or an IDE of your choice and create a new component definition file called *ChatB1.component.xml* (cf. figure above). We recommend using eclipse for Java EE for editing files or some other advanced XML-Editor. In this file all important startup properties are defined in a way that complies to the Jadex schema specification. First property of the component is its type name which must be the same as the file name (similar to Java class files), in this case it is set to *ChatB1.component.xml*. Additionally you can specify a package attribute, which has a similar meaning as in Java programs and serves for grouping purposes only (you will need to alter the package name with respect to your actually used directory structure). All Java classes from the component's package are automatically known to it and need not to be imported via an import tag.

Start your first active component

Start the Jadex platform. In the Jadex Control Center (JCC) use the “Add Path” button explained above to add the root directory of your example package (typically named bin or classes). Then open the folder until you can see your file “ChatB1.component.xml”. The effect of selecting the input file is that the component model is loaded. The filename extension *component.xml* is used by the Jadex platform to determine which kind of component it is loading. When the model contains no errors, the description of the model, taken from the XML comment above the componenttype tag, is shown in the description view. In case there are errors in the model, correct the errors shown in the description view and press reload. Below the file name, the component name and its default configuration are shown. After pressing the start button the new component should appear in the component tree (at the bottom left). It is also possible to start a component simply by double-clicking it in the model tree. *Please note that when you use a double-click on the model name in the left tree view to start a component, the settings on the right will be ignored.*

Exercise B2 - Java Component Definition

There are various Jadex active component types such as applications, BPMN workflows, micro and BDI agents. Most of these component types are XML-based. These types all share the same XML descriptor format introduced above and extend it in certain directions. In contrast, micro agents are defined using Java only. In order to equip such Java based components with active component specifics the *Java annotation mechanism* can be used. Annotations are meta-information placed in to Java source code starting with `'*@*`. The

meta-information can be used by tools or frameworks (like Jadex) for dealing with the Java objects in a special way.

```
package tutorial;
import ...

/**
 * Chat micro agent.
 */
@Description("This agent offers a chat service.")
@Agent
public class ChatB2Agent
{
}
```

Basic chat micro agent definition

Use your IDE to create a Java class called *ChatB2Agent.java* and add the annotations as shown in the figure above. In this case the component only posses a description that will be displayed in the JCC and the marker annotation *@Agent*. The Java comment cannot be used directly as Jadex operates on class files, in which the Java comments are not retained. Please further note that it is also required to follow a naming convention which requires that all micro agent files end with *Agent.java*. Start the micro agent following the same steps as explained in excercise B1 and verify that it appears at the component tree at the lower left panel in the Starter. You can kill a component by rightclicking on it to activate a popup menu and choosing *Kill component*. # Required Services

In this chapter we will cover the mechanisms that can be used to obtain services and use them programmatically. In active components (as in SCA) services are part of components and consist of an interface specification and a service implementation. Service interfaces are defined as Java interfaces and implementations as corresponding Java classes. Required services allow specifying in a declarative way how the service we need should look like and how the systems can find such a service. These aspects of how to find a suitable service are defined in the *service binding*. Such bindings can either be static, i.e. once the required service is bound it will not change from call to call, or dynamic meaning that it will be rebound each time the required service is accessed.

Exercise C1 - Declaring a Predefined Service

In this exercise we will fetch the predefined clock service. The clock service is already provided by the platform component itself so that it is sufficient to define that the service interface is named *jadex.bridge.service.clock.IClockService* and that it can be found within the *platform* scope.

Defining the Component

Create a Java class called *ChatC1Agent.java* and use the following:

- Add the *@Agent* annotation to state that this java file is an agent
- Add a *@Description* annotation with an illustrative example explanation such as *@Description("This agent uses the clock service.")*
- Add a required service description that defines the name of the service as *clockservice*, its type as *IClockService* and its scope as *RequiredService-Info.SCOPE_PLATFORM*.

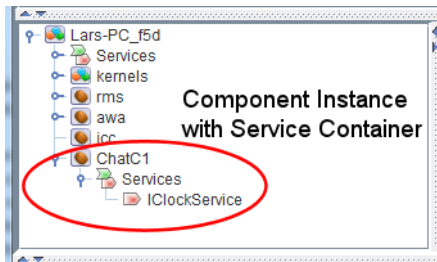
The resulting definition should look like the following:

```
package tutorial;
import ...

@Description("This agent declares a required clock service.")
@Agent
@RequiredServices(@RequiredService(name="clockservice", type=IClockService.class,
    binding=@Binding(scope=RequiredServiceInfo.SCOPE_PLATFORM)))
public class ChatC1Agent
{
}
}
```

Verify the Component Behavior

To understand the code it is necessary to explain the underlying concepts of the component service container. Each active component (regardless of its type) contains a service container that basically fulfills three aspects. It allows for *fetching required services*, *searching services*, and *providing services*. In the Starter of the JCC the each component is displayed with its service container. Within the service container node the provided and required services can be seen. In the screenshot below the ChatC1 agent instance with its service container the required clock service is shown.



Component tree in Starter

Exercise C2 - Invoking a Predefined Service

In this lecture we will use the clock service to print out the current platform time.

Defining the Component

- Create a Java class called *ChatC2Agent.java* and copy the content from the last lecture.
- Insert a field of type *MicroAgent* and name it *agent*. Add an *@Agent* annotation above the field declaration. Jadex will notice the *@Agent* annotation and automatically inject the micro agent object to the agent. The injected micro agent can be used to access the agent programming interface, e.g. for fetching its service container.
- Declare a public void method called *executeBody()* and add a *@AgentBody* annotation above the method. This method is one of three lifecycle methods of a micro agents (*agentCreated()*, *executeBody()* and *agentKilled()*) which will contain the functional agent code and is called once after the agent is born.
- What is still missing is the usage of the declared required service. This will be done in the *executeBody()* method with the following code:

```
IFuture<IClockService> fut = agent.getServiceContainer().getRequiredService("clockservice");
fut.addListener(new DefaultResultListener<IClockService>()
{
    public void resultAvailable(IClockService cs)
    {
        System.out.println("Time for a chat, buddy: "+new Date(cs.getTime()));
    }
});
```

Verify the Component Behavior

The injected micro agent is used to get the service container and then the declared service is looked up via its name using the *getRequiredService("clockservice")* method. Fetching a required service makes Jadex inspect the binding definition of that service and possibly initiates an implicit service search. As resolving the binding may involve complex lookups it is realized as asynchronous call using a *future* return value (also nearly all services are defined this way). This means that the call immediately returns the future that represents a holder for the 'real' return value. The future allows the caller to be decoupled from the action of the callee because the method call can return without waiting for the callee to finish processing and the result is set by the callee whenever it is ready. (For further details on futures e.g. refer to Wikipedia) The caller can add a

listener on the future to be informed when the ‘real’ result is ready. Here a *DefaultResultListener* is used, which requires only one method to be overridden called *resultAvailable()*. The result is of type *IClockService*, on which calling the *getTime()* method delivers the current time in millis. This time is printed out to the console. (In contrast to *System.currentTimeMillis()* the platform clock could also return other times, e.g. when Jadex is used in simulation mode).

Exercise C3 - Invoking a Futurized Service

In this exercise we will use another service of the platform and invoke a method on it. In contrast to the clock service, which offers only normal synchronous method signatures the component management service used here has an asynchronous interface.

Defining the Component

- Create a Java class called *ChatC3Agent.java* and copy the content from the last lecture.
- Change the required service specification to look for the *IComponentManagementService.class* and change the name to *cms*. The scope can be kept the same as this service is also made available by the platform component itself.
- Adapt the *getRequiredService()* call to use the new service name *cms* and type *IComponentManagementService*.
- Change the parameter type of the *resultAvailable* method to *IComponentManagementService*.
- Invoke the *getComponentDescriptions()* method on the service to get an array of all components in the platform. Note, that the call returns a future so that you will have to add a result listener again for result processing.
- Iterate through the retrieved *IComponentDescription[]* array and print out the results one by one.

Verify the intended behavior

Start the platform and the agent and check if the component descriptions are printed out. The output should look similar to the console snapshot shown below. Please note that the component management service is one of the central services of the Jadex platform. It can e.g. be used to *create*, *kill*, *suspend* and resume components. Feel free to inspect the *IComponentManagementService* interface to learn more about these functionalities.

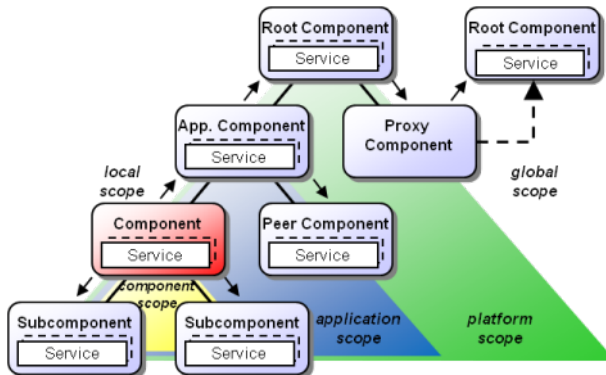
```

Progress @ Javadoc Problems Search Properties Console X
Jadex Standalone [Java Application] C:\Program Files (x86)\Java\jre6\bin\javaw.exe (29.07.2011 09:58:35)
Lars-PC_355 platform startup time: 2383 ms.
Found: CMSComponentDescription(name=Lars-PC_355, state=active, ownership=null)
Found: CMSComponentDescription(name=kernel_micro@kernels.Lars-PC_355, state=active, ownership=null)
Found: CMSComponentDescription(name=kernels@Lars-PC_355, state=active, ownership=null)
Found: CMSComponentDescription(name=ChatC3@Lars-PC_355, state=active, ownership=null)
Found: CMSComponentDescription(name=kernel_component@kernels.Lars-PC_355, state=active, ownership=null)
Found: CMSComponentDescription(name=jcc@Lars-PC_355, state=active, ownership=null)
Found: CMSComponentDescription(name=rms@Lars-PC_355, state=active, ownership=null)
Found: CMSComponentDescription(name=awa@Lars-PC_355, state=active, ownership=null)
Found: CMSComponentDescription(name=broadcastdis@awa.Lars-PC_355, state=active, ownership=null)

```

Console snaphsot

Please note that, besides the interface type itself, the most important factor of searches and required service specifications is the search scope. It defines the area of the search and is per default set to *application*. This means that only components within the started application are considered within the search. Knowing this it becomes clear why we had to change the scope to *platform* in all lectures so far. Otherwise the search would have stopped at the application component and the platform services would not have been found. In the figure below a visual representation of search scopes is given.



Component search scopes

Exercise C4 - Searching services

In general, the services used by a component should be made explicit by declaring required services in the component model. By default the required services are assumed to be static and therefore search a service only once and subsequently returns the cached value. (This default behavior can be changed by setting *dynamic=true* in the binding). In rare cases one might want to search for services directly. This can be done by fetching the service container and using the *searchService* methods. An alternative that can be used if really fine-grained search control is necessary is using the *jadex.bridge.service.SServiceProvider* class, which provides many static methods for searching services. In this lecture we will search for the micro agent factory service of the platform. This is not directly possible using a

required service definition, because the platform has several component factories with the same interface *IComponentFactory*. (As an alternative one could use a required service binding for all services of type *IComponentFactory* and select from those). So what we need is a possibility to further restrict the search results. This can be done using a *jadex.bridge.service.IResultSelector*. For component factories there is already a ready to use selector called *jadex.bridge.service.component.ComponentFactorySelector*.

Defining the Component

- Create a Java class called *ChatC4Agent.java* and copy the content from the last lecture.
- Delete the required service annotation.
- Use the following code to search for the factory:

```
IFuture<IComponentFactory> factory = SServiceProvider.getService(agent.getServiceContainer()  
    new ComponentFactorySelector(MicroAgentFactory.FILETYPE_MICROAGENT));  
factory.addListener(agent.createResultListener(new DefaultResultListener<IComponentFactory>  
{  
    public void resultAvailable(IComponentFactory result)  
    {  
        System.out.println("Found: "+result);  
    }  
}));
```

Verify the intended behavior

After starting the component on the console the output should indicate that a factory was found that is capable to load micro agents. Depending on the platform configuration this could either be the *MultiFactory* or the *MicroAgentFactory*.

Exercise C5 - Declaring a Predefined Service in XML

This lecture will explain how to define a required service within an XML component specification. The lecture is functionally equivalent to the lecture C1.

Defining the Component

- Create a component file called *ChatC5.component.xml* and copy the content from lecture B1.
- Adjust the filename comments.
- Add an imports section and add an import for the class *jadex.bridge.service.types.clock.IClockService*.

- Add a services section and add a required service definition for the clock service. Its class has to be set to *IClockService*. Within the required service definition add a binding specification with scope set to *platform*.

The result should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<componenttype xmlns="http://jadex.sourceforge.net/jadex"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://jadex.sourceforge.net/jadex
    http://jadex.sourceforge.net/jadex-component-2.1.xsd"
  name="ChatC5" package="jadex.micro.tutorial">

  <imports>
    <import>jadex.bridge.service.types.clock.IClockService</import>
  </imports>

  <services>
    <requiredservice name="clockservice" class="IClockService">
      <binding scope="platform"/>
    </requiredservice>
  </services>
</componenttype>
```

Verify the intended behavior

Start the component and check that it has a service container with the required clock service inside.

Exercise C6 - Invoking a Predefined Service in XML

This lecture will explain how to use a required service within an XML component specification. The lecture is functionally equivalent to the lecture C2. It has to be stated that typically plain XML components do not possess own behavior but are rather used for defining applications and composites, i.e. components with subcomponents. Despite this typical use case it is also possible to attach some behavior to XML components. For this purpose so called steps can be used which have to be implemented as Java classes.

Defining the Component

- Create a component file called *ChatC6.component.xml* and copy the content from the last lecture.

- Add a component step in the XML description by inserting the following snippet after the services section.

```
<configurations>
  <configuration name="first">
    <steps>
      <initialstep class="PrintTimeStep"/>
    </steps>
  </configuration>
</configurations>
```

- Create a new Java class called *PrintTimeStep* and let it implement the *jadex.bridge.IComponentStep* interface. The class will have to implement one method called *execute(IInternalAccess ia)* in which the code to fetch and invoke the clock service (the same as in lecture C2) has to be placed. You can use the internal access argument here to get the service container of the component. For simplicity, just return *IFuture.DONE* as a result of the step. The usage of the return value of the component step will be discussed in a later chapter.

```
public class PrintTimeStep implements IComponentStep<Void>
{
  public IFuture<Void> execute(IInternalAccess ia)
  {
    ...
    return IFuture.DONE;
  }
}
```

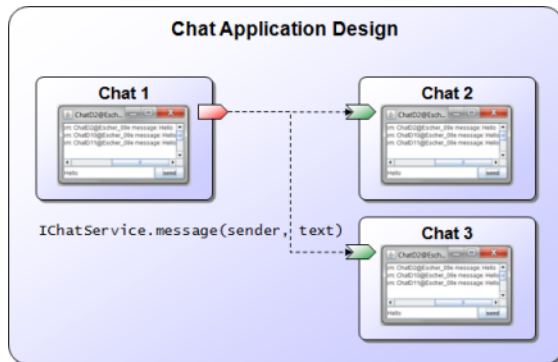
Verify the intended behavior

Assure that after starting the component it will print out the current time on the console out. # Provided Services

This chapter will deal with aspects of how to implement and provide component services. In general, a component service consists of two aspects: a service interface and a service implementation.

Learning how to build provided services allows building an initial version of the chat application as described in 01 Introduction. The design figure is repeated below for completeness. Now that the concepts of active components and services has been introduced, you should be able to grasp some more details of this figure. First, each chat instance is represented by an active component. Second, the chat components communicate using a provided and required service called

IChatService. Third, this service provides a method *message()* that is called on all running components, whenever a chat component wishes to send a message.



Conceptual design of the chat application

Exercise D1 - Defining a service

In this exercise we will create a basic chat service, attach it to the chat component and invoke it for testing purposes.

Defining the chat service interface

- Create a Java interface file called *IChatService.java* and add one method called *message* with two input parameters of type *String* called *sender* and *text*.

Defining the chat service implementation

- Create a Java class file called *ChatServiceD1.java* and let it implement the *IChatService* interface.
- Add the *@Service* annotation above the class definition.
- Add a field of type *InternalAccess* and name it *agent*. Also add a *@ServiceComponent* annotation above this field. Jadex will automatically inject the agent to the service so that the service can access functionalities of the agent.
- Add a field of type *IClockService* and name it *clock*.
- Add a field of type *DateFormat* and name it *format*.
- Add the *message* method from the *IChatService* interface. The method will be called to let the chat service know about a new message. In the message body this new message should be printed out to the console. Concretely the output should look like: <receiver component name> received at

<time> from: <sender> message: <text>. You can access the receiver's component name using `agent.getComponentIdentifier().getLocalName()`.

- Add a method called `startService` with no parameters and an `IFuture<Void>` return value. Place the `@ServiceStart` annotation above the method signature. This method is called once after the service is created and will be used to init the service. In this case the method should assign the `format` with `new SimpleDateFormat("hh:mm:ss")` and the `clock` by fetching the clock service in the same way as in earlier lectures. Please note that it is important that the method should return a future indicating when the init has been finished. As fetching the clock service is done asynchronously init has finished when the clock service has been assigned. To ensure that the caller of the start method is notified also in case an error occurs and the service could not be found a `DelegationResultListener` can be used. It will forward exception to the future that is passed to it. This init code for the clock should look like this:

```
final Future ret = new Future();
...
IFuture<IClockService> fut = agent.getServiceContainer().getRequiredService("clockservice");
fut.addListener(new DelegationResultListener<IClockService>(ret)
{
    public void customResultAvailable(IClockService result)
    {
        clock = result;
        super.customResultAvailable(null);
    }
});
return ret;
```

Defining the chat service component

- Create a Java class file called `ChatD1Agent.java` and copy its content from the `ChatC2Agent`.
- Add a second required service definition to the agent. For that purpose you have to change the required services specification to look like `@RequiredServices({rs1, rs2})`, with `rs1`, `rs2` representing the respective service definitions. The first clock service specification can be kept and the second we will name `chatservices`. The type has to be declared as `IChatService` and as we wish to retrieve all chat services we need to set `multiple` to true. In the binding of the service we use scope `platform` again and additionally define it as `dynamic`. Making it dynamic disallows caching former search results and will always deliver the currently available chat services to the caller.
- Furthermore, we need to specify the provided service. We use the `@Provid-`

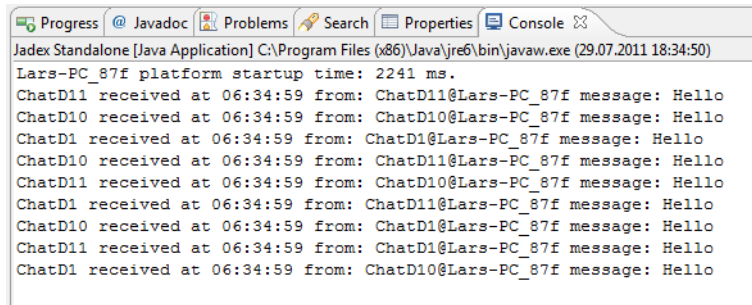
edServices annotation and add one *@ProvidedService* annotation inside. For a provided service we have to define its interface using the type attribute and set it to *IChatService* in this case. In addition, a provided service should have an implementation which is defined using the *@Implementation* annotation. Here, we just directly specify the implementation class to *ChatServiceD1*. It should look like the following:

```
@ProvidedServices(@ProvidedService(type=IChatService.class, implementation=@Implementation(C
```

- The code of the agent body should be changed to fetch the chat services using the call *agent.getServiceContainer().getRequiredServices("chatservices")*. As result you will retrieve a *java.util.Collection* of the available chat services (at least the one our agent is offering itself). Iterate through this collection and invoke the *message* method on each service with your own component name as sender *getComponentIdentifier().getName()* and some arbitrary text as message content.

Verify the Component Behavior

Start the component and observe if it prints out the message text to the console. Then try out to start another chat agent with another name. If you didn't change the name the platform will complain that it cannot start the agent due to a naming conflict. You can activate the *Auto Generate* option in the Starter to ensure that the platform automatically creates a new component instance name for each started component. Whenever you start a new component you should see in the output that it sends a message to all existing agents. Below you can see the output that has been produced from three chat agents.



```
Jadex Standalone [Java Application] C:\Program Files (x86)\Java\jre6\bin\javaw.exe (29.07.2011 18:34:50)
Lars-PC_87f platform startup time: 2241 ms.
ChatD11 received at 06:34:59 from: ChatD11@Lars-PC_87f message: Hello
ChatD10 received at 06:34:59 from: ChatD10@Lars-PC_87f message: Hello
ChatD1 received at 06:34:59 from: ChatD1@Lars-PC_87f message: Hello
ChatD10 received at 06:34:59 from: ChatD11@Lars-PC_87f message: Hello
ChatD11 received at 06:34:59 from: ChatD10@Lars-PC_87f message: Hello
ChatD1 received at 06:34:59 from: ChatD11@Lars-PC_87f message: Hello
ChatD10 received at 06:34:59 from: ChatD1@Lars-PC_87f message: Hello
ChatD11 received at 06:34:59 from: ChatD1@Lars-PC_87f message: Hello
ChatD1 received at 06:34:59 from: ChatD10@Lars-PC_87f message: Hello
```

Exercise D2 - Chat User Interface

In this lecture we will add a small graphical chat user interface. The interface will display chat messages of other chat users and allow us to send manually entered chat messages.

Defining the chat user interface

- Create a new Java class called *ChatGuiD2* that extends *JFrame*.
- Add a field of type *JTextArea* and name it *received*. It will be used to display the received messages.
- Add a method *addMessage* that adds new chat messages to the content of the text area.
- Create a constructor with one parameter *ChatGuiD2(final IExternalAccess agent)*. The external access allows the user interface to work on the agent. In the constructor the following needs to be done:
 - Set the title of the chat window to the component name by calling *super(agent.getComponentIdentifier().getName())*
 - Create the user interface components, the *received* text area, a *JTextField* called *message* for the user to enter a message text and a *JButton* called *send* for sending messages.
 - Layout the gui components using some *LayoutManager*, e.g. using a *BorderLayout*
 - Show the gui by calling *pack()* and *setVisible(true)*
 - Add an *ActionListener* to the send button to notify the available chat services of the new message. The code should look like the following:

```
send.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        final String text = message.getText();
        agent.scheduleStep(new IComponentStep<Void>()
        {
            public IFuture<Void> execute(IInternalAccess ia)
            {
                IFuture<Collection<IChatService>> chatservices = ia.getServiceContainer().getRequiredServices();
                chatservices.addListener(new DefaultResultListener<Collection<IChatService>>()
                {
                    public void resultAvailable(Collection<IChatService> result)
                    {
                        for(Iterator<IChatService> it=result.iterator(); it.hasNext(); )
                        {
                            IChatService cs = it.next();
                            cs.message(agent.getComponentIdentifier().getName(), text);
                        }
                    }
                });
                return IFuture.DONE;
            }
        });
    }
});
```

```
    }  
  });
```

- Also in the constructor, add code to terminate the chat component when the window is closed:

```
addWindowListener(new WindowAdapter()  
{  
    public void windowClosing(WindowEvent e)  
    {  
        agent.killComponent();  
    }  
});
```

Defining the chat service implementation

- Create a Java class file called *ChatServiceD2.java* and copy its content from the last lecture.
- In contrast to the previous lecture we will use the user interface to output the received messages. Therefore, in the *startService* method we need to create the gui and then use it in the *message* method. The creation of the gui has to be done on the Swing thread. Hence, we will change the result listener to a *SwingDelegationResultListener* which ensures that the *customResultAvailable()* method is called on the Swing thread. The user interface can be created by calling *new ChatGuiD2(exta)*, letting *exta* being the external access of the agent. The init code should then look like the following:

```
final IExternalAccess exta = agent.getExternalAccess();  
IFuture<IClockService> fut = agent.getServiceContainer().getRequiredService("clockservice");  
fut.addListener(new SwingDelegationResultListener<IClockService>(ret)  
{  
    public void customResultAvailable(IClockService result)  
    {  
        clock = result;  
        gui = createGui(exta);  
        super.customResultAvailable(null);  
    }  
});
```

- Add the *protected ChatGuiD2 createGui(IExternalAccess agent)* method and implement it by simply returning *new ChatGuiD2(agent)*.
- In order to dispose the user interface when the component is killed, a new method called *shutdownService()* needs to be created. Equip the method

with the `@ServiceShutdown` annotation to let Jadex call the method when the service is terminated. Within the method you should call `gui.dispose()` to close the user interface. You should call dispose from the Swing thread, which can be achieved by using `SwingUtilities.invokeLater()` and placing the dispose within the `Runnable` that has to be passed as parameter.

- Finally the `message` method has to be adjusted that it does not print out the message but redirects it to user interface using the previously defined gui method `addMessage`

Defining the chat service component

- Create a Java class file called `ChatD2Agent.java` and copy its content from the last lecture.
- Remove the `executeBody` method.
- Adapt the implementation class of the provided service to `ChatServiceD2`.

Verify the Component Behavior

After starting several chat agents you should be able to enter text messages and send them using the send button. If your implementation works correctly you should see the sent messages appearing the all other chat windows. Also verify that the chat window is close when you kill the agent from the component tree. Below the screenshot shows what can be expected.

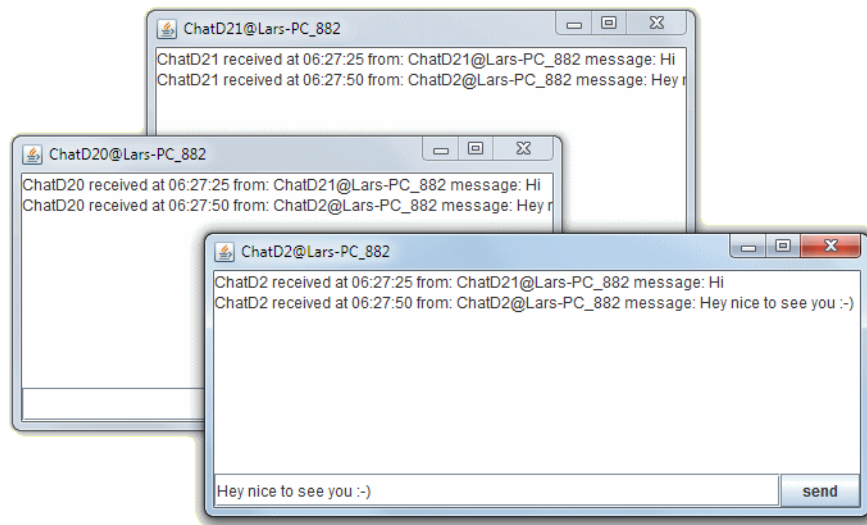
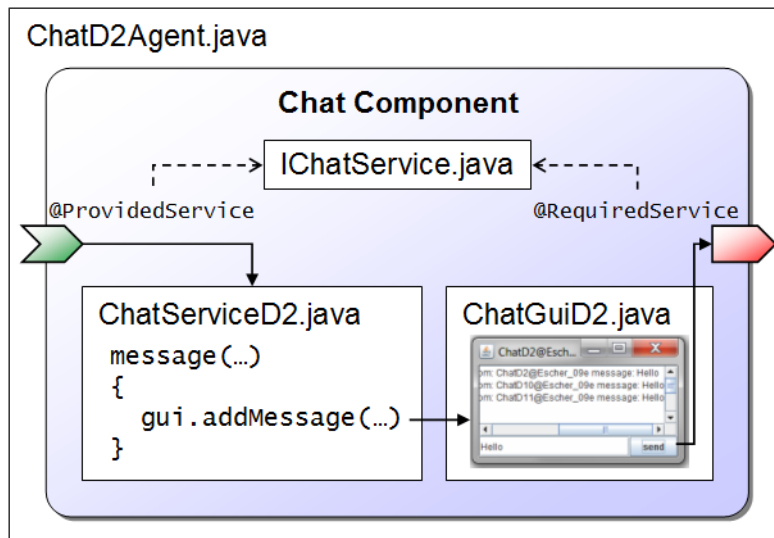


Figure 3: 05 Provided Services@chatter.png

Understanding the Chat Implementation

The figure below shows the interplay between the implemented Java classes when executing a chat component. The component is defined in the file *ChatD2Agent.java*, where the provided and required services are specified. Both specifications refer to the service interface defined in the file *IChatService.java*. The provided service annotation further specifies the service implementation *ChatServiceD2.java*. On startup, the service implementation creates a user interface as defined in *ChatGuiD2.java*. Whenever a chat message is received through the provided service interface, the *message()* method of the chat service implementation gets executed. It calls the *addMessage()* method of the user interface, such that the chat message is displayed. Moreover, when the user hits the “send” button in the gui, the gui will fetch the available chat service as specified in the required services and call the *message()* method on the remote chat services.



Interplay between implemented Java classes inside a chat component

Exercise D3 - Service Interfaces

So far we have used a very simple chat service interface. In this lecture we will extend the interface with another method that has a non-void return value. In general, all methods with return values should return future objects. In this way methods can be processed asynchronously by the called active component and the caller does not have to wait for the answer but instead is notified once it has been made available by the callee. With this purely asynchronously call model Jadex achieves a loose coupling between caller and callee and additionally avoids technical thread deadlocks simply because there are never waiting threads in

components. Regarding the rule of future return values there is one exception. Methods without parameters that return constant values can be used because their result values can be cached. In this lecture we will add a method that returns the user profile of a chatter on request. This will allow us to see details of our potential chat partners. As a user may change its profile at any time it cannot be considered constant.

Defining the chat service interface

- Create a new Java interface file called *IExtendedChatService.java* and let it extend *IChatService*
- Add a method *getUserProfile()* with no in parameters and *IFuture<UserProfileD3>* as return value.
- Create a new Java class file *UserProfileD3* and add fields as well as public getter and setter methods for *String name*, *int age*, *boolean gender*, and *String description*.
- Add an empty constructor and a field based constructor with all fields to the *UserProfileD3* class.
- Override the *toString()* method and return a descriptive representation of the profile using its attributes.

Defining the chat service implementation

- Create a Java class file called *ChatServiceD3.java*, let it inherit from *ChatServiceD2* and implement the new *IExtendedChatService*.
- Add a field of type *UserProfileD3* with name *profile* and assign it a random generated user profile already as part of the variable declaration. (Hint: you could e.g. use a statically created list of predefined user profiles from which you select an entry using *Math.random()*num_of_profiles*).
- Implement the *getUserProfile()* method by returning *new Future<UserProfileD3>(profile)*, i.e. a future with the user profile.
- Override the *createGui()* method and return a new gui version using *new ChatGuiD3(agent)*.

Defining the chat user interface

- Create a Java class file called *ChatGuiD3.java* and let it extend *ChatGuiD2.java*.
- In the constructor create a new *JButton* called *profiles* and add it to the frame's content pane by using *getContentPane()*. Depending on your layout of the content pane you can add the new button with different constraints, e.g. if you chose *BorderLayout* and north has not been used you could call *getContentPane().add(profiles, BorderLayout.NORTH)*.

- You also need to add an action listener on the new *profiles* button that is in charge of printing the user profiles to the gui message area (using *addMessage()*). The code of the action listener should start with:

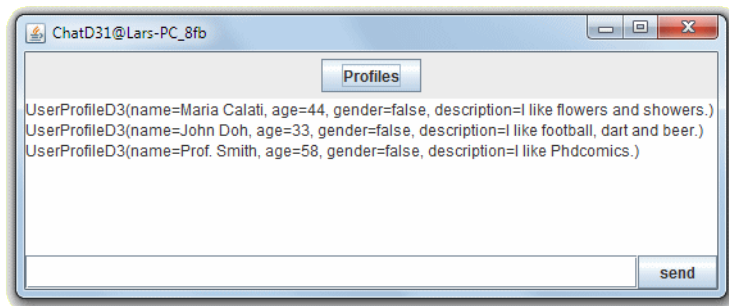
```
agent.scheduleStep(new IComponentStep<Void>()
{
    public IFuture<Void> execute(IInternalAccess ia)
    {
        IFuture<Collection<IChatService>> chatservices = ia.getServiceContainer().getRequiredServices(IChatService.class);
        chatservices.addResultListener(new DefaultResultListener<Collection<IChatService>>()
        {
            ...
        });
    }
});
```

Defining the chat service component

- Create a Java class file called *ChatD3Agent.java* and copy its content from the last lecture.
- Replace occurrences of *IChatService* with *IExtendedChatService*.
- Replace *ChatServiceD2* with *ChatServiceD3*.

Verify the Component Behavior

Use the JCC to start several chat agents. Check if the new *Profiles* button is present in the chat user interface and if hitting the button causes the profiles of currently online users to be printed out. Below a screenshot of the solution with imaginary profiles of three chatters are shown.



Exercise D4 - Service Interceptor

In this lecture we will deal with service interceptors, which are a means for interrupting a method call before and/or after it is executed and executing

arbitrary code for pre- or postprocessing purposes. Using service interceptors it is possible to easily implement crosscutting concerns which would be scattered in many methods otherwise. In this way the interceptor concept is very similar to aspect oriented programming, in which point cuts are used to intercept calls. Good example use cases are authorization checks or logging. In our example we will use an interceptor to prevent spam messages from being displayed at the user's message display. In this case we make the simplified assumption that spam messages can be identified by the sender's name. Concretely, we will block all messages coming from chatters with 'Bot' in their name.

Defining the chat service interceptor

- Create a new Java class file called *SpamInterceptorD4.java* and let it implement *IServiceInvocationInterceptor*.
- Add method implementations for the defined method signatures in the interface.
- The *isApplicable* method is used to check if the interceptor should be called for the current call. Here, we only want to intercept method calls to the *message* method. So in the method body you should use *context.getMethod().getName().equals("message")* to determine the result.
- In the *execute* method the functionality of an intercepted call should be placed. In order to check if the sender's name contains 'Bot' we need first to fetch the argument with the sender's name. This can be achieved using *(String)context.getArgumentArray()[0]*. The check itself can be performed by a substring containment operation using *sender.indexOf("Bot")!=-1*. In case this check is true we will not call the message method but return an exception immediately by returning *new Future((new RuntimeException("No spammers allowed.")))*. Additionally, you should print out to the console that a spammer call was blocked and the sender's name and message content. If the name is ok, we let the call pass by returning *context.invoke()*.

Defining the chat service component

- Create a Java class file called *ChatD4Agent.java* and copy its content from the last lecture.
- Change the provided service definition to contain the interceptor definition. The chat implementation needs not to be modified. This should look like the following:

```
@ProvidedServices(@ProvidedService(type=IExtendedChatService.class,
implementation=@Implementation(value=ChatServiceD3.class,
interceptors=@Value(clazz=SpamInterceptorD4.class))))
```

Verify the Component Behavior

Start several chat agents including an agent that contains ‘Bot’ in its name. The name of a component can be entered in the *Component name* field of the Starter. If the field is uneditable you should deselected the checkbox named *Auto generate*, which will generate names automatically. Enter messages in normal chatter and the spam bot and test if messages from the spam bot are blocked. You should see a printout on the console as shown in the screenshot below.

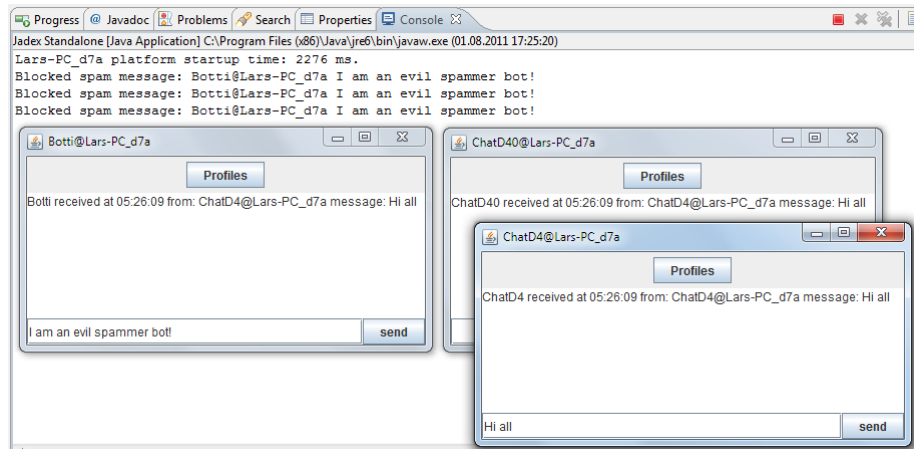


Figure 4: 05 Provided Services@spammers.png

Exercise D5 - Remote Services

Until now we have used chat agents only from one platform. In this lecture we will change our chat agent to efficiently communicate also with agents from other platforms that may reside on different network nodes. Basically, the only important change that has to be performed consists in changing the search scope of the required service for chat agents from *SCOPE_PLATFORM* to *SCOPE_GLOBAL*. One problem with this solution is that we currently search for chat services in a way that delivers the results altogether, i.e. we have to wait till all known platforms have answered or a timeout has occurred. To avoid waiting for all results an *IntermediateResultListener* can be used. This listener is notified separately for each result as soon as it becomes available. Each time the *intermediateResultAvailable* method is called with one result value. After the last result has been made available the *finished* method is called. In case the called method does not support intermediate results it behaves exactly as the already known *IResultListener*, i.e. *resultAvailable* is called when all results are available or *exceptionOccurred* is invoked if an exception occurred during processing.

Defining the chat user interface

- Create a Java class file called *ChatGuiD5.java* and copy its content from *ChatGuiD2.java*
- Change the listener implementation that is added to search for the chat services in a step by step manner. Implement the *intermediateResultAvailable* method in the same way as before by calling the *message* method on the found service. Please note that you will get a single service as parameter, not a collection of services.

```
ia.getServiceContainer().getRequiredServices("chatservices")
    .addResultListener(new IIntermediateResultListener()...
```

Defining the chat service implementation

- Create a Java class file called *ChatServiceD5.java* and copy its content from *ChatServiceD2.java*.
- Change the created gui by making the *createGui* method return *new ChatGuiD5(agent)*.

Defining the chat service component

- Create a Java class file called *ChatD5Agent.java* and copy its content from *ChatD4Agent.java*.
- Change the scope of *chatservices* to *RequiredServiceInfo.SCOPE_GLOBAL*.
- Change the implementation of the provided chat service to *ChatServiceD5*.

Verify the Component Behavior

Start two or more Jadex platforms and on each at least one chat agent. Send a chat message via a chat agent and observe if the message arrives also at the remote chatters. Simulate a network breakdown by right clicking on the rms component and selecting suspend (the rms is the remote management service component, which is responsible for remote service communication between platforms). Then choose a chat agent from another platform and send a message again. Ensure that the still available chatters get the message immediately and do not suffer from waiting for a timeout from the disconnected remote platform.

Composition

In this chapter we will introduce composite components, i.e. components that are composed of subcomponents. As these subcomponents may also be primitive

components or composites actually a hierarchy of components emerges. One important reason for creating components out of others is that one often wants to create *self contained* components, i.e. components with few outbound required services. Self contained components can provide most of their functionality out of the box in varying application contexts.

Exercise E1 - Composite components and configurations

In the first lecture of this chapter we will create a composite component that has several subcomponents. The lecture will show how subcomponents can be defined and how configurations can be used to specify different component setups. As the composite component itself will only used as component container and does not contain functionalities by itself we will use an XML component descriptor (of course we could use a micro agent as well). A configuration is a specific setting of a component that has a name. At startup of a component the configuration can be selected. This allows to define different interesting setups within one component. In this example we will create a component with two configurations. In the first only one chat subcomponent is created whereas in the second ten chat agents will be instantiated.

Defining the Component

- Create an XML file called *ChatE1.component.xml* and copy the contents from *ChatB1.component.xml*
- Add a *componenttypes* section and insert one *componenttype* definition inside.
- The *componenttype* should be equipped with a *name* and a *filename* attribute.
- The *name* is used as internal identifier within the component.xml to refer to the defined kind of agent. Here is should be set to *chatagent*.
- The *filename* serves as file location and could be set to *tutorial.ChatD5Agent.class*.
- Create a *configurations* section below the *componenttypes* section. In the configurations section place two *configuration* definitions. The first should be named *One chatter* the second *Ten chatters* using the *name* attribute of the *configuration* definition.
- In each of the *configuration* section add a *components* section and inside of it a *component* definition. The component should have the attribute *type* set to *chatagent* (or formerly defined logical agent type). In the component definition of the second configuration also introduced an attribute *number* and set it to *10* (all attributes must be set in quotation marks).

<configurations>


```
<configuration name="One chatter">
```

```
...
```

Definition of a configuration

Verify the component behavior

Select the new component model in the JCC and look at the configuration choice box at the right upper panel. Ensure that you can see both configuration names. Start the component once with each of the configurations selected and look inside the component in the component. You should see one and in the second case ten subcomponents inside of it.

Exercise E2 - Agent arguments

In addition to configurations a component can be customized at startup by using *arguments*. Furthermore, a component can declare *results* which are assumed to be available after component termination. Taken together, arguments and results can be used for a function oriented view on components, if a component can use a subcomponent in a function oriented way by starting it with specific arguments and waiting it to terminate for fetching the results. In this lecture we will change the component description from the last exercise to include an argument with which we can determine the number of subcomponents that should be created.

Defining the Component

- Create an XML file called *ChatE2.component.xml* and copy the contents from the previous lecture.
- Add an *arguments* section at the beginning of the componenttype definition. Within this section define an *argument* with name set to *chatters* and *class* set to *int*. In between the opening and closing tag of the argument its default value can be specified as Java expression. Here we just set it to *2*.
- Add a configuration called *Argument number of chatters* and define it in the same way as the others. In contrast to the others we will set the number attribute of the *component* description in this case to “*\$args.chatters*”.

Verify the component behavior

- After selecting the component model in the JCC you should see an input field for the argument value in the Starter panel as shown below in the screenshot. Try out starting without and with entering a number in

the argument input field. You should check that the started composite component has as many components as you entered. Be sure to have started the component in the new configuration.

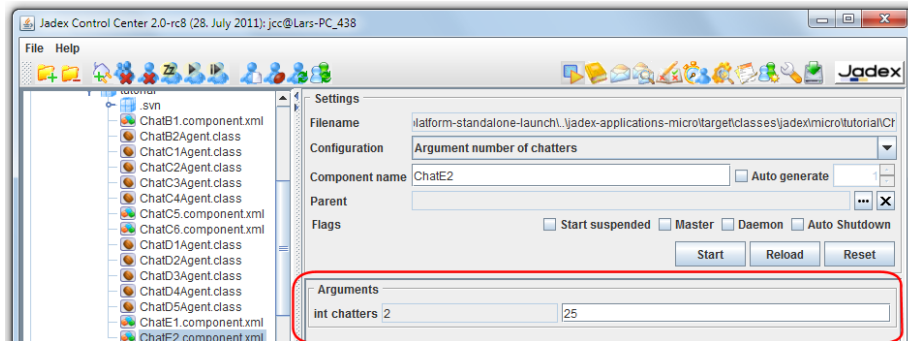


Figure 5: AC Tutorial.06 Composition@arguments.png

Exercise E3 - Static component binding

This lecture will show how services of components can be directly connected. In this way typically normal SCA components are bound to each other. The use case in this exercise is that we introduce a registry component at which the chatters announce their real and nickname identity. In an application there is one predefined registry component that is made directly known to the chatter agents at startup.

Defining the registry service interface

- Create a new interface called *IRegistryServiceE3.java*.
- Add a void method for registering a chatter with the following name and parameters *register(IComponentIdentifier cid, String nickname)*
- Add a method for getting the chatter identities named *getChatters()* and *IFuture* as return value.

Defining the registry service implementation

- Create a new class called *RegistryServiceE3.java* that implements the interface *IRegistryServiceE3*
- Add a class attribute called *entries* of type *Map* and initialize it to a *new HashMap*.
- Add both methods of the interface.

- In the *register* method use *put* to store the nickname as key and the component identifier in the map.
- In the *getChatters* method simply return the entries by *new Future(entries)*.

Defining the registry agent

- Create a new agent class called *RegistryE3Agent.java*.
- Mark the class as an agent by adding the *@Agent* annotation above the class definition.
- Add the registry service by adding a *@ProvidedServices* annotation. Inside of this annotation create a *@ProvidedService* annotation with type set to *IRegistryServiceE3*. Inside of the provided service annotation add an *@Implementation* annotation and assign it to *RegistryServiceE3.class*.

Defining the chat agent

- Create a new agent class called *ChatE3Agent.java* and copy its content from *ChatD5Agent*.
- You can keep the service implementation of D5 as we will not introduce new chat functionalities here.
- Add a *@Arguments* annotation. Inside create a *@Argument* annotation and give it the *name* nickname, the *clazz* String and the *defaultvalue* “Willi”. The backslash is used because Jadex expects a parseable Java expression and for a String the quotation marks necessary. The defaultvalue is used in case no other value is supplied during startup of the component (e.g. from the JCC or programmatically).
- Also add a field *nickname* of type String in the agent class. Above this field add the *@AgentArgument* annotation. This automatically injects the argument value in the corresponding field.
- Add a *@RequiredService* annotation with name *regservice* and type *IRegistryServiceE3.class*. Do not add a *@Binding* annotation inside. Instead we will assign the binding explicitly in the composite component.
- Delete the content of the *executeBody* method. We will use it to fetch the registry service. On the registry service we will first call the register method with the component’s identifier and some chooseable nickname. Thereafter, we will let the agent wait for 10 seconds and then use the registry service again to get all registered chat agents. This should look like the following:

```
agent.getServiceContainer().getRequiredService("regservice")
    .addListener(new DefaultResultListener()
{
    public void resultAvailable(Object result)
    {
```

```

final IRegistryServiceE3 rs = (IRegistryServiceE3)result;
rs.register(agent.getComponentIdentifier(), "my_nick");

agent.waitFor(10000, new IComponentStep()
{
    public IFuture<Void> execute(IInternalAccess ia)
    {
        rs.getChatters().addResultListener(new DefaultResultListener()
        {
            public void resultAvailable(Object result)
            {
                System.out.println("The current chatters: "+result);
            }
        });
        return IFuture.DONE;
    }
});
});
});

```

Defining the chat composite component

- Create a composite file called *ChatE3.component.xml* and copy its content from *ChatE2.component.xml*.
- Delete the *arguments* section.
- In the *componenttypes* section change the *chatagent* component filename to the new *ChatE3Agent.class*.
- Add a second *component* definition with name *registry* and filename *tutorial.RegistryE3Agent.class*.
- Delete all configurations except the first one and name it *Two chatters*.
- In this configuration add a component definition with type *registry* and name *reg*. It is important to set the name of this component as we will use it in the following binding definitions.
- Change the chat component definition to include the arguments required service binding of the registry. Here we use the instance name of the registry *reg* to assign it as target component of the required service in the chat component. This is a deployment time wiring of components. It should look like the following:

```

<component name="chatter1" type="chatagent">
  <arguments>
    <argument name="nickname">"Hans"</argument>
  </arguments>
  <requiredservices>

```

```

    <binding name="regservice" componentname="reg" scope="parent"/>
  </requiredservices>
</component>

```

- Create a second component definition by copying the first one and change the component *name* to *chatter2* as well as the *nickname* to Franz.

Verify the component behavior

In the JCC start the chat application (`ChatE3.component.xml`) and check if it has three subcomponents (one registry and two chat agents). After ten seconds you should see that the chat agents print out the information from the registry (component id and nicknames). The nicknames of the components should correspond to the argument values that have been entered in the component xml, namely Franz and Hans. In case you comment out the argument value for a component it will use the *defaultvalue* as specified in the chat component itself (here Willi). The expected output is also shown below in the figure.

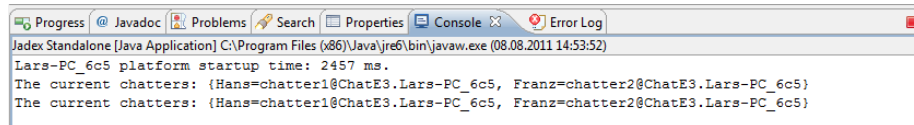


Figure 6: AC Tutorial.06 Composition@consolee3.png

Exercise E4 - On demand component creation

In this small lecture we will create the registry component on demand, i.e. when a chat component tries to access the registry service for the first time and no service provider could be found, it will create one on its own. This kind of behavior is not enabled per default and we need to turn it on by setting the *create* flag to true in the required service binding specification.

Defining the chat composite component

- Create the component xml file called *ChatE4.component.xml* by copying the content from *ChatE3.component.xml*.
- Delete the registry component definition, i.e. the following line:

```
<component name="reg" type="registry"/>
```

- We change the required service binding of both chat component instance definitions to look like the following:

```
<binding name="regservice" componentname="reg" scope="parent" create="true">
  <creationinfo name="reg" type="registry"/>
</binding>
```

Verify the component behavior

After startup you can see that the same components are started as in lecture E3. Looking inside the chat application reveals that it has one registry and two chat components. This time the registry is created when a chat agent resolves its required registry service. It is important to note that in this case the *name* attribute *reg* is also used for component creation, i.e. the registry gets this name. To determine the model of the component to start the *type* attribute is used, which is set to *registry*.

Exercise E5 - Using services of specific components

In this exercise we will change the system behavior so that the chat agents use the registry to look up the component identifier of a chatpartner's nickname. Using the component identifier we will fetch the chat service of this chat partner and send him a private message.

Defining the chat agent

- Create a chat agent Java class called *ChatE5Agent.java* by copying it from *ChatE3Agent.java*.
- Add an *@Argument* annotation in the *@Arguments* area and name it *partner*. The type should be set to *String* and no default value needs to be specified.
- Add a field called *partner* of type *String* in the chat agent class. Above the field declaration add an *@AgentArgument* annotation.
- The existing behavior in the *executeBody* method can be kept. In addition to printing out the available chatters from the registry we insert code to talk with our specified chat partner. For this purpose we first need to fetch the component identifier of the partner and can then retrieve the chat service from the component with that id using our service container. The code looks like the following:

```
...
Map chatters = (Map)result;
System.out.println("The current chatters: "+result);
final IComponentIdentifier cid = (IComponentIdentifier)chatters.get(partner);
if(cid==null)
```

```

{
    System.out.println("Could not find chat partner named: "+partner);
}
else
{
    agent.getServiceContainer().getService(IChatService.class, cid).addResultListener(new Defa
    {
        public void resultAvailable(Object result)
        {
            IChatService cs = (IChatService)result;
            cs.message(agent.getComponentIdentifier().toString(), "Private hello from: "+nickname);
        }
    });
}
}

```

Defining the chat composite component

- Create the component xml file called *ChatE5.component.xml* by copying the content from *ChatE3.component.xml*.
- Change the componenttype definition of the chatagent to *tutorial.ChatE5Agent.class*
- Add an argument in the component instance argument section for the *partner* attribute. For the first chat agent with name *Hans* the partner should be *Franz* and for the second chat agent vice versa.

Verify the component behavior

Starting the composite component should lead to the following behavior. Both chat agents first register at the registry component. After ten seconds both agents use the registry to determine the component identifier of their chat partner using its partner's nickname (called partner). In the chat windows a private message of the other chat agent should appear as shown below.

External Access

The previous lessons have introduced how to build the internals of components (e.g. service implementations) and how to build loosely coupled interactions based on explicitly defined required and provided services. There are some occasions where a more tight coupling is desired, e.g. when building a user interface for a specific component implementation. For this reason, Jadex provides mechanisms for accessing and manipulating internals of components.

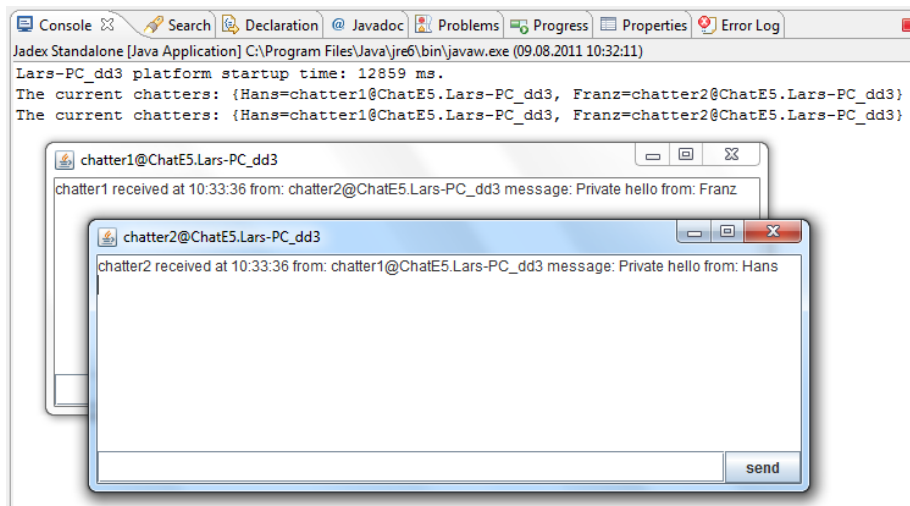


Figure 7: AC Tutorial.06 Composition@console5.png

Exercise F1 - Chat Bot

As a basis for the subsequent exercises in this chapter a new agent is used: the chat bot. It has the purpose to monitor chat messages and issue a reply message, whenever a chat messages contains a given keyword.

Defining the chat bot component

- Create a Java class file called `ChatBotF1Agent.java` and copy its content from the `ChatD2Agent`.
- Remove the required service definition of the 'clockservice' as it is not needed.
- Add two fields of type *String* named 'keyword' and 'reply'. These fields will control the behavior of the bot, i.e. to which messages it reacts (keyword) and which message it will issue in response (reply).
- We want these fields to be automatically injected from the component arguments, therefore add an '@AgentArgument' annotation to each.
- Add a corresponding get and set method for each field.
- Add an '@Arguments' annotation above the class definition as shown below. The arguments annotation defines which kind of arguments can be specified when starting the component. This information is e.g. used by the JCC allowing a user to enter argument values before starting a component.

```

@Arguments({
    @Argument(name="keyword", clazz=String.class, defaultvalue="\nerd", description="The k

```



```
@Argument(name="reply", clazz=String.class, defaultvalue="\Watch your language\"", descri  
})
```

Defining the chat service of the chat bot

- Create a Java class file called ChatServiceF1.java implementing the *IChatService* interface.
- Change the chat service implementation class of the ChatBotF1Agent to refer to the new service.
- Add a field of type *IInternalAccess* called 'agent' and add a '@ServiceComponent' annotation for injecting the component (c.f., e.g. Exercise D2).
- Implement the *message()* method by checking if the received message contains the keyword as specified in the agent. If the keyword is contained, send the reply message concatenated with the name of the sender to all chat services (e.g. adapting the code from ChatGuiD2). You can access the keyword in the agent and check for containment as follows:

```
// Reply if the message contains the keyword.  
ChatBotF1Agent chatbot = (ChatBotF1Agent)((IPojoMicroAgent)agent).getPojoAgent();  
if(text.toLowerCase().indexOf(chatbot.getKeyword().toLowerCase())!=-1)  
{  
    ...  
}
```

Verify the Component Behavior

Start the chat bot and another chat agent (e.g. ChatD2). Enter and send a chat message containing the keyword. Observe that the chat bot will automatically respond to the message. Send another message without the keyword and observe, if the chat bot stays quiet. The conversation might look like shown below:

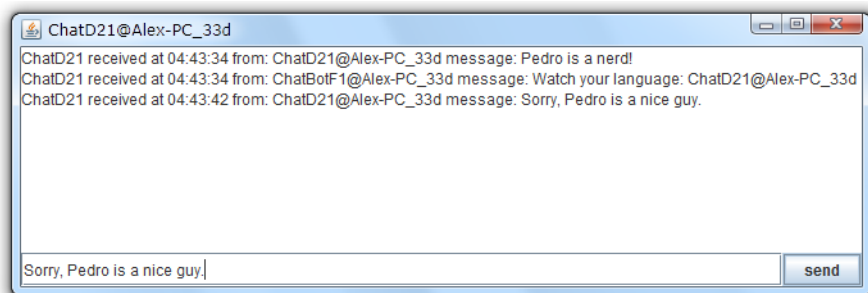


Figure 8: 07 External Access@chatbotreply.png

Exercise F2 - Component Viewer

In this exercise, we will create a user interface for the chat bot that can be accessed from the JCC.

Defining a Chat Bot User Interface

- Create a new Java class named *BotGuiF2* and let it extend *AbstractComponentViewerPanel* from package *jadex.base.gui.componentviewer*. The component viewer panel is a mechanism used in Jadex to add user interfaces to components. These user interfaces can be accessed using the ‘Component Viewer’ tool of the JCC. The advantage of the mechanism is that it allows administering components on remote platforms, too.
- Implement the *getComponent()* method as required by the abstract superclass. This method should return a swing component (e.g. a *JPanel*) that represents the components user interface. The chat bot user interface should contain two text fields (*JTextField*) for the keyword and the reply.

Defining a Chat Bot Component

- Copy the contents of the *ChatBotF1Agent.java* to a new *ChatBotF2Agent.java* and change the service implementation class to *ChatServiceF2*.
- Copy the contents of the *ChatServiceF1.java* to a new *ChatServiceF2.java* and change the occurrences of *ChatBotF1* accordingly.
- Add a *GuiClass* annotation to the agent class pointing to the user interface implementation as follows:

```
@GuiClass(BotGuiF2.class)
```

Verify the User Interface

- Launch the Jadex platform and start the chat bot (F2).
- Switch to the component viewer tool and double-click on the ChatBotF2 component in the tree on the left (see below).
- The user interface will be displayed allowing you to enter keyword and reply.

The actual look of the user interface largely depends on the Swing components and layout managers that you have used. E.g. in the screenshot a *GridBagLayout* and two labels in front of the text field were used, in addition to a titled border around the panel itself. Currently the user interface does nothing, when new values are entered. We will change that in the next exercise.

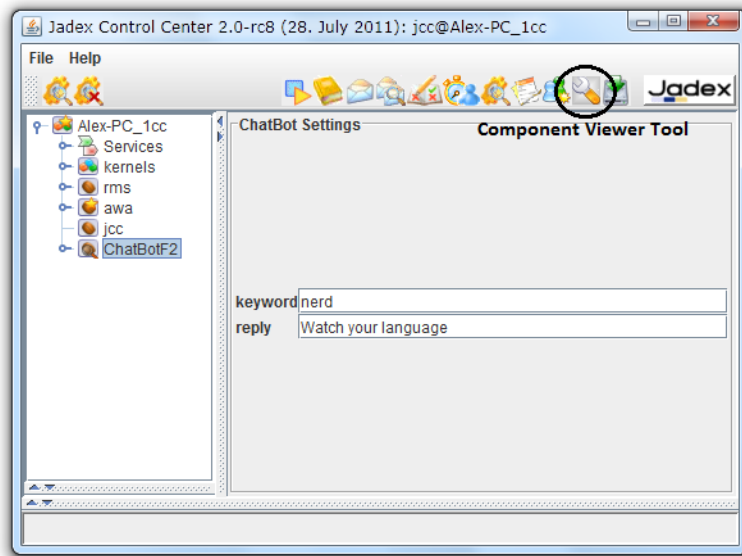


Figure 9: 07 External Access@jcccomponentviewer.png

Exercise F3 - Scheduling Steps on Components

In Jadex, each component owns its own conceptual thread (provided by the platform whenever needed). The single-thread execution model of a component assures state consistency as the internal data structures inside a component are never accessed by two threads at once. For the chat bot user interface to work properly, it requires access to the keyword and reply values of the component. The user interface runs on the swing thread and therefore cannot access the component internals directly, as this would cause consistency issues due to concurrent access. This exercise shows how to schedule a step on the components thread and safely access the component internals.

Defining the User Interface

- Copy the *BotGuiF2* contents to a new *BotGuiF3.java* file.
- Towards the end of the *getComponent* method, fetch the values of the reply and keyword properties from the component. Use the *getActiveComponent()* method already provided by the abstract component viewer panel to get hold of the *IExternalAccess* of the chat bot component. Schedule a step on the component and use the *IInternalAccess* to obtain the chat bot object. Get the reply and keyword values from the chat bot using the get methods and return a *String* array containing bot values. This result value is provided in the future return value of the *scheduleStep()* method.

```

getActiveComponent().scheduleStep(new IComponentStep<String[]>()
{
    public IFuture<String[]> execute(IInternalAccess ia)
    {
        ChatBotF3Agent chatbot = (ChatBotF3Agent)((IPojoMicroAgent)ia).getPojoAgent();
        return new Future<String[]>(new String[]{chatbot.getKeyword(), chatbot.getReply()});
    }
})

```

- Add a result listener to the future returned by the *scheduleStep()* method call and set the obtained values in the GUI. Use a *SwingDefaultResultListener* to have the code executed on the Swing thread to avoid GUI inconsistencies.

```

...addResultListener(new SwingDefaultResultListener<String[]>()
{
    public void customResultAvailable(String[] values)
    {
        tfkeyword.setText(values[0]);
        tfreply.setText(values[1]);
    }
});

```

- For reacting to changes add action listeners to the two text fields as shown below. In each action listener first extract the value from the text field and afterwards schedule a step on the component for setting the changed value by using the set method of the chat bot object.

```

tfkeyword.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        final String keyword = tfkeyword.getText();
        ...
    }
});

```

Defining the Component

- Copy and edit the chat bot and chat service to create F3 versions and don't forget to change the *GuiClass* annotation in the chat bot to point to the new *BotGuiF3*.

Verify the User Interface

- Launch the Jadex platform and start the chat bot (F3).
- Open the user interface in the component viewer.
- Check if the initial values of the reply and keyword properties are correctly displayed.
- Change the reply and keyword properties by entering a new value and hitting return in each text field.
- Start a chat component and send some messages to verify that the chat bot settings have actually changed.

Exercise F4 - Scheduling Steps on Remote Components

Another advantage of the schedule step approach is that the developer can differentiate between code executed on the local platform and code that is executed on a potentially remote platform. Consider a situation where the user interface of the chat bot is on a different computer than the chat bot itself. With schedule step a developer specified a piece of code that may be transferred across the network and executed remotely.

When looking at the code from the last exercise (shown again below), you should be aware that the two inner lines are potentially executed in a different Java machine on a different computer. Therefore inside the `execute()` method you are not allowed to access fields or methods of the enclosing object, because it is not available at the remote side. One exception are final variables, which are automatically transferred over the network by the Jadex infrastructure.

```
getActiveComponent().scheduleStep(new IComponentStep<String[]>()
{
    public IFuture<String[]> execute(IInternalAccess ia)
    {
        // This code is executed on a potentially remote component.
        ChatBotF3Agent chatbot = (ChatBotF3Agent)((IPojoMicroAgent)ia).getPojoAgent();
        return new Future<String[]>(new String[]{chatbot.getKeyword(), chatbot.getReply()});
    }
})
```

Accessing the chat bot from a remote platform

- Launch two Jadex platforms and start the chat bot (F3) on one of them.
- Go to the component viewer of the **other** platform (not running the chat bot). This platform should automatically connect to the platform running the chat bot.

- Unfold the remote platform node and double-click on the remote chat bot to open its GUI (you may have to wait a little after opening the component viewer tool or the platform node due to inter-process communication). The GUI should behave exactly the same as in the local case.

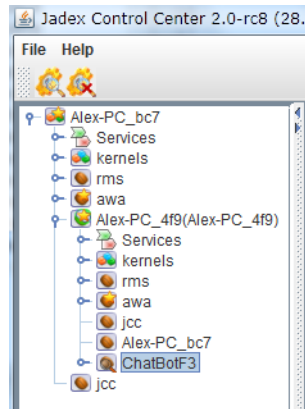


Figure 10: 07 External Access@jccremoteviewer.png

- Check, if you can alter the keyword or reply property. If an exception occurs, you probably access part of the GUI (running on one platform) from the scheduled step (executed on the other platform). Make sure that you only access final variables inside the steps as follows:

```
keyword_textfield.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        // Access GUI on local platform.
        final String keyword = keyword_textfield.getText();
        getActiveComponent().scheduleStep(new IComponentStep<Void>()
        {
            public IFuture<Void> execute(IInternalAccess ia)
            {
                // Use final keyword value on remote platform.
                ...

                return IFuture.DONE;
            }
        });
    }
});
```

Making the remote execution safe for different builds

When accessing the GUI remote you may have noticed the following message being printed to the console: *Warning: Anonymous class without XML class name property (XML_CLASSNAME) / annotation (@XMLClassname): tutorial.BotGuiF3\$1.*

This warning indicates a potential problem due to the Java language specification not describing a naming scheme for anonymous inner classes. Each java compiler decides for itself how to name an inner class (typically OuterClass\$1, OuterClass\$2, ...). This can cause incompatibilities when two platforms communicate that have been compiled using a different compiler (e.g. javac vs. eclipse). To allow proper mapping of inner classes you can specify an additional identifier using the @XMLClassname annotation:

- Copy the F3 files into new F4 files, changing the all occurrences of F3 to F4 accordingly.
- For each inner *IComponentStep* class, add an *@XMLClassname("some_identifier")* annotation. Of course you should use different identifiers for each occurrence (three in total).
- Access the GUI remotely and check if the warnings have vanished.

Security

Thanks to mechanisms for global awareness and connectivity, in principle, any Jadex platform around the world may find and invoke any services of components on any other Jadex platform. In practice, of course, access to platforms and provided services needs to be restricted to appropriate groups of users. In this chapter you will learn how to specify or relax the default security restrictions of Jadex services in your application code. Furthermore you will learn how to configure your platforms to enable restricted access to other platforms.

Exercise G1 - Making the Chat Publicly Available

The security level of services and their methods can be adjusted by the *@Security* annotation (package *jadex.bridge.service.annotation*). Currently, two levels are supported (other more fine-grained levels may be added later). The default level for all services is *PASSWORD* and allows access only to platforms, which have some sort of security credentials for the invoked platform. The other level is *UNRESTRICTED* and allows access to any platform.

Starting two different Platforms

Currently, your chat service has the default security level *PASSWORD* and therefore cannot be accessed from other platforms. To verify this behavior, start two platforms with different names. E.g. in eclipse, duplicate your launch configuration (cf. Installation) and add the following in the programm arguments section: `-platformname platform2_`. Start the chat component on each platform (e.g. *ChatD2*) and check that chat messages are not sent between the platforms.

Changing the security level of the service

Edit the *IChatService.java* and add a corresponding security annotation.

```
@Security(Security.UNRESTRICTED)
public interface IChatService
{
    ...
}
```

Restart the two platform and verify that that messages are exchanged. Note, that the annotation can be added to the service as a whole, but also seperately to the service methods. The annotation of a method takes precedence over the annotation of the service as a whole. E.g. when the service itself is unrestricted but the method has an additional annotation with level *PASSWORD*, the service can be found but the method can not be accessed from outside platforms.

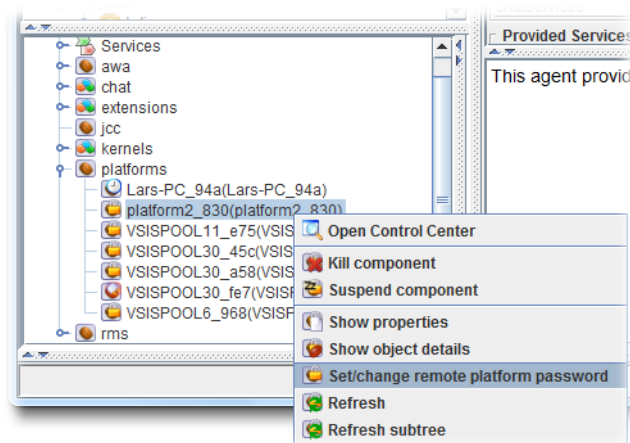
Exercise G2 - Accessing Restricted Services

In the Jadex Control Center, you can edit the security settings of a platform. This allows e.g. to set the password of the platform, but also to add known passwords of other platforms. Furthermore, you can setup trusted network zones. Using these security settings you can enable service calls between components, even when the services are restricted.

For this exercise, remove the security annotations from the chat service interface. Start two platforms as above and verify that the platforms do not communicate. In the following, different techniques are described to enable the restricted access.

Platform passwords

At startup, each Jadex platform prints out the platform as stored in its *.settings.xml*. Find the password of the second platform in the console. Go to the starter panel in the JCC of the first platform, unfold the *platforms* node and right-click on the node of the second platform (see below).



Entering a password for a remote platform

Enter the password of the second platform in the appearing dialog. The first platform should now be able to access all services of the second platform. As a result, the node of the second platform will change its color to green. Repeat the process by entering the first platform's password in the second platform's JCC. Now the two chat components should be able to communicate with each other.

Setting up a trusted network

In the following an alternative setup for allowing restricted access is described. To be able to test if it works, reset the password settings by deleting the *platform2.settings.xml* file. When you now start the second platform it will generate a different password and also no longer knows about the first platform's password. Therefore chat communication between the two platforms will be disabled.

Now open the security panel in the first platform's JCC, which is identified by a lock symbol. In the text fields at the bottom, enter a network name of your choice and click 'Add'. Add the same network name in the second platform. Instead of having separate passwords for each platform, the security network settings allow establishing a group of platforms that allow access to each other. While not strictly necessary, you can also add a password for the network. You can also add multiple networks for platforms that should be present in more than one group. As long as two platforms share at least one network name with the same password (or no password for both), they will allow restricted communication.

Start a chat component on each platform and verify if it works. Further details about security issues and settings can be found in the security settings chapter in the tool guide .

Application Integration

In the previous lessons, the chat component was always started from the Jadex control center (JCC). Typically, when you are developing an application with active components, you do not want the end users having to access the JCC. Instead you want your components seamlessly integrated in a larger application.

In this chapter it will be shown how to start a Jadex platform and start and access components directly from Java code. In this way, active components can be integrated in any kind of desktop or server Java applications. Here, a Java main class is developed to start the chat as a standalone Java application. Of course, the same code can be used as well for integrating your Jadex components into servlets or other kinds of Java frameworks.

Exercise H1 - Starting a Platform

This exercise shows how a platform can be started from Java code.

Starting a platform with standard arguments is very easy using the *createPlatform(...)* method of the *jadex.base.Starter* class. The return value of this method is a future object that contains the external access of the platform component, once the platform startup has finished successfully. For fetching the future result, you could use a result listener as known from the previous lessons. As the thread running the *main()* method is not managed by the Jadex concurrency model there is another option. Using a so called *ThreadSuspendable*, the main thread can be blocked until the future is available. This technique avoids the necessity of inner classes, which comes with the use of result listeners. Note, that usage of the thread suspendable is only allowed when running on threads, which are not managed by Jadex. If you try to use the thread suspendable in Jadex threads, e.g. in component or service code, the system will probably run into deadlocks.

Write a Main Class

- Create a new java class file named *MainH1.java*.
- In the class add a main method, i.e. a method with the following signature:

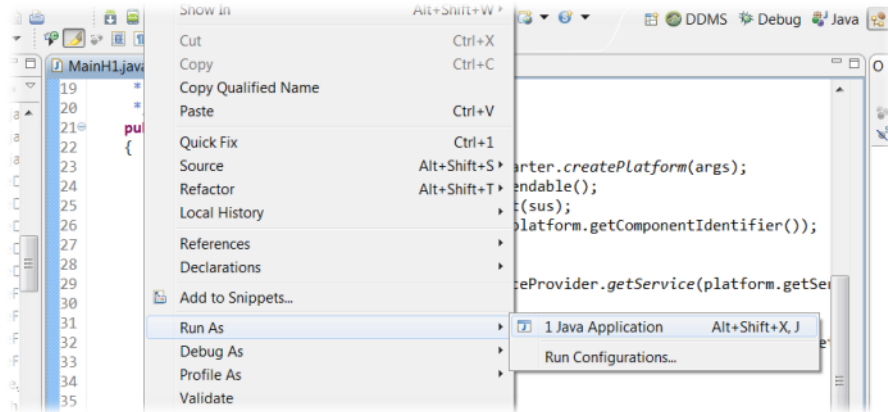
```
public static void main(String[] args)
```

- as explained above, place the following code in the method body:

```
IFuture<IExternalAccess> platfut = Starter.createPlatform(args);
ThreadSuspendable sus = new ThreadSuspendable();
IExternalAccess platform = platfut.get(sus);
System.out.println("Started platform: "+platform.getComponentIdentifier());
```

Test the Application

For starting the newly written class as a Java program in eclipse, right-click in the editor and choose *Run As -> Java Application*.



Starting a Java application from eclipse

If everything is OK the JCC should appear. In addition to the normal Jadex outputs you should also see the ‘Started platform: ...’ line that was printed from the above code.

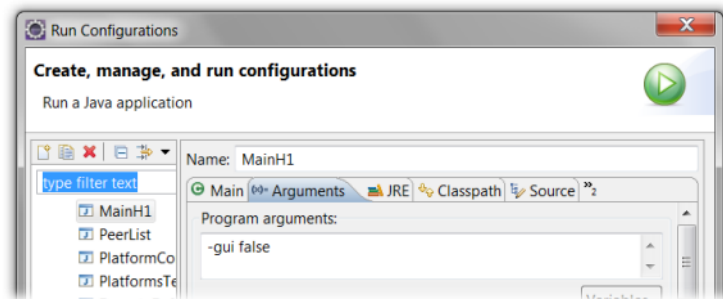
Exercise H2 - Platform Arguments

By default, the Jadex platform opens the JCC and prints some messages on the console. This is usually undesirable, when embedding Jadex in an application. Furthermore, you may want to tweak settings of the platform, e.g. for disabling unnecessary features for improved performance. In this exercise, the platform start code will be adapted to supply some custom arguments.

Supply an Argument at Application Launch

You may have noticed in the previous exercise, that the *args* parameter of the *main(...)* method is passed to the *createPlatform(...)* method. Therefore we can adapt the eclipse launch configuration to supply arguments to the Jadex platform. In the following we want to stop the JCC from opening.

- Open the run configuration created in the previous exercise (e.g. right-click in editor and choose *Run As -> Run Configurations...*).
- Change to the *Arguments* tab and add the line ‘-gui false’.



Changing launch arguments in eclipse

- Run the application and check, that the JCC does not appear.

Supply Default Arguments in the Main Method

Instead of having to supply the arguments from the outside, you will typically want to specify some kind of default settings directly in the application. Anyhow, it is useful to be able to specify additional arguments from the outside, e.g. for testing purposes.

- Copy the *MainH1* class to a new file *MainH2.java*.
- Change the body of the main method to the following:

```
String[] defargs = new String[]
{
    "-gui", "false",
    "-welcome", "false",
    "-cli", "false",
    "-printpass", "false"
};
String[] newargs = new String[defargs.length+args.length];
System.arraycopy(defargs, 0, newargs, 0, defargs.length);
System.arraycopy(args, 0, newargs, defargs.length, args.length);
IFuture<IExternalAccess> platfut = Starter.createPlatform(newargs);
ThreadSuspendable sus = new ThreadSuspendable();
IExternalAccess platform = platfut.get(sus);
System.out.println("Started platform: "+platform.getComponentIdentifier());
```

This code first defines a string array with default settings as follows:

- **-gui false** disables the JCC
- **-welcome false** disables printing of the welcome message with the platform startup time
- **-cli false** disables the command line interface and thus also prevents printing of the command prompt (*Host_123>*)

- **-printpass false** disables printing of platform password

Afterwards a new array is created and filled with first the default arguments and then with the args supplied to the main method. Therefore, in case of conflicts the arguments supplied from the outside override the default settings. Make sure that in the `createPlatform(...)` method the `newargs` array is passed instead of the `args`. Otherwise the default settings will be ignored.

Test the Application

- Start the application with a new launch configuration or change the old launch configuration to refer to `MainH2` instead of `MainH1`.
- Observe that no messages are printed to the console except the `Started platform: ...` statement from the main method itself.
- Change the arguments in the launch configuration (e.g. add `gui -true` for opening the JCC again)
- Restart the application and check if your changes are respected.

Note that there is a large number of useful arguments for customizing the platform. You can e.g. configure the awareness or transport mechanisms or change security settings, etc. For debugging, especially the `-logging true` option is very helpful, as it enables printing of `info` and `warning` messages in addition to the `severe` messages, which are printed by default. An overview of available settings can be found in the platform jadexdoc

Exercise H3 - Creating a component

Now that we have the platform access available, we can do everything programmatically that we could also do with the JCC. Therefore we can access running components and also start and stop new components as needed.

In this exercise, we use the platform access to obtain the component management service (CMS). We then use the CMS to create a chat component.

Obtain the CMS

- Copy the `MainH2` class to a new file `MainH3.java`.
- Extend the main method to search for the CMS service. For this purpose, use the static helper class `SServiceProvider` from package `jadex.bridge.service.search` as follows:

```
IComponentManagementService cms = SServiceProvider.getService(platform.getServiceProvider()  
IComponentManagementService.class, RequiredServiceInfo.SCOPE_PLATFORM).get(sus);
```

The *SServiceProvider* class provides access to the service search mechanism and further provides helper methods for dealing with required and provided services of components. Here, we use the *getService(...)* method for searching for a service. The supplied service provider represents the entry point for the search, from which the components to be searched are derived using the search scope. Here we use the service provider of the platform. The *cms* interface class is supplied to indicate the type of service we are interested in. Finally, the search scope *platform* states that all components and subcomponents of the platform should be searched.

The result of the *getService(...)* method is a future of the corresponding service type. The application is still running on the Java main thread, therefore it is safe to use again the thread suspendable for blocking until the search result is available.

Create a component programatically

- On the *cms*, invoke the *createComponent(...)* method.
- Most arguments can be set to *null*, only the second argument (*model*) is required.
- As *model*, supply the class name of the chat component and append “.class”. Use the chat component from Exercise D2.
- The result is a future of the component identifier of the newly created component.
- Wait for the component identifier using the thread suspendably and afterwards print the id to the console.
- The resulting code should look as follows:

```
IComponentIdentifier cid = cms.createComponent(null, ChatD2Agent.class.getName()+".class", r
System.out.println("Started chat component: "+cid);
```

The arguments to the *createComponent(...)* method are as follows:

- **name:** The name of the new component. When set to *null*, a name is automatically generated.
- **model:** The file name of the component model. Here we derive the name from the chat component built in Exercise D2 to avoid typos. You could also supply a string directly, e.g. “*mypackage/MyAgent.class*” or “*mypackage/My.component.xml*”.
- **info:** An object of type *jadex.bridge.service.types.cms.CreationInfo*, which can be used to supply various options like component arguments, parent, etc. Can be set to *null* if no special options are required.
- **resultlistener:** A result listener that is called when the created component terminates. The result listener will be given the result values that are produced by the component.

The method returns a future result and asynchronously starts initiating the new component in the background. When the new component could be started successfully, its component identifier is provided as a result. The result is only made available after all init code of the component and its services (if any) has completed.

Test the application

- Start the application using a new or updated launch configuration for the *MainH3* class.
- The chat window should appear indicating that the chat component was created.
- Test the chat by sending some messages.
- Close the chat window. The Java VM will exit, because as a default, the Jadex platform shuts down when no more application components are running.

Exercise H4 - Accessing a Component

When integrating your components into some application, you often need some form of coordination and/or data exchange between the components and the remaining application code outside of Jadex. The natural way for realizing this is using services of your components. Sometimes, you may be able to reuse the services that are already present in your components. Otherwise, you need to devise new service interfaces that capture the interaction requirements between a component and outside code.

In this exercise we obtain the chat service of the chat component to print a welcome message in the GUI.

Obtain the Chat Service

- Copy the *MainH2* class to a new file *MainH3.java*.
- Use the *SServiceProvider* helper class to obtain a service from the created chat component. The method *getService(provider, cid, type)* allows fetching a declared service of a specific component directly.
- Call the *message(...)* method of the chat service to display a startup message.
- The required code looks as follows:

```
IChatService chat = SServiceProvider.getService(platform.getServiceProvider(), cid, IChatService.class);
chat.message("Main", "Chat started.");
```

Note that we use the service provider of the platform as search entry point, but we specify the cid of the chat component. Thus we state that we want to search for the *IChatService* only in this specific chat component. As before, the *suspendable* is used to wait and fetch the future result value. The obtained chat service object can be used to invoke the methods of the chat service object.

Test the Application

- Update the launch configuration and start the application.
- Check that the welcome message appears in the chat window.

Exercise H5 - Packaging the Application

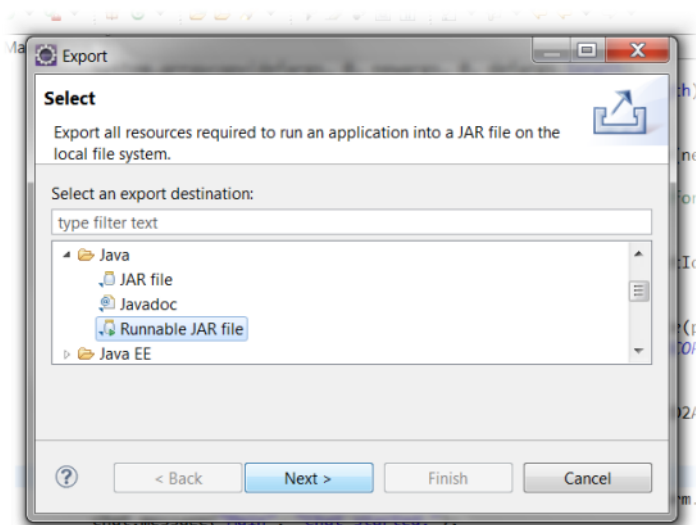
In the previous exercises, you were still starting your chat application from eclipse. If you deploy your application to end users you will want to have some deployment package of your application that can be started by a double-click.

In this exercise it is described how to compose an application for deployment to users.

Alternative 1: Executable Jar

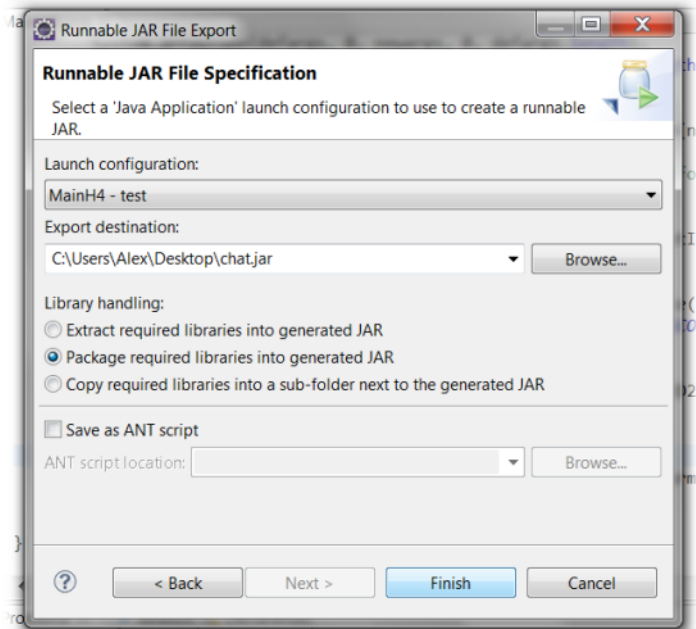
When you develop with eclipse you are almost done, because eclipse allows exporting a project directly as an executable jar file. An executable jar is in the essence a zip file including your Java classes and contains additional meta information, so that Java knows which main class to start, when the jar file is executed.

- Make sure that you have an up-to-date launch configuration and that your application works as expected.
- Right-click on the project and choose *Export...*
- In the appearing dialog choose *Java -> Runnable JAR File* and hit *Next >* (see image for step 1).
- In the next dialog, choose your launch configuration to be exported, the target file name and specify the library handling (see image for step 2).



Exporting a jar file (step 1)

In the second step as shown below you can specify the way the dependencies of the application are exported. Your application depends on the jars from the Jadex distribution. You can either extract, package, or copy the required libraries. Extraction means that classes and support files from the dependency jars are extracted and repackaged directly in the newly produced jar file. For packaging, the dependent jar files are kept unchanged but are included in the new jar file, such that as before only one output file is produced. In the last option 'copy' the dependencies are stored in a separate folder besides the generated jar. Which option you choose is mostly a matter of taste.



Exporting a jar file (step 2)

- Export your application with the chosen options.
- Test the exported application by double-clicking on the jar file.

Alternative 2: Launch Script

If you are a little bit used to Java, you may know that you can start an executable jar as produced in the last section by typing 'java -jar *filename.jar*' in the console. Therefore you can use the same command in a .bat file (on Windows) or an .sh script (on Linux). But it is also quite easy to start your application from the console or a batch script even without producing an executable jar file.

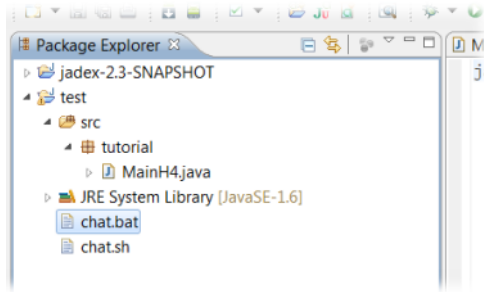
- Locate the jadex-platform-standalone-launch-2.2.jar from the Jadex distribution (the version number might differ depending on which Jadex version you use).
- Create a batch file (.bat or .sh) in your eclipse project using right-click *New -> File*
- Open the script in a text editor and add the following line for windows:

```
java -classpath "<path to jadex launch jar>;bin" tutorial.MainH4
```

- or for Linux (note the ':' instead of the ';'):

```
java -classpath "<path to jadex launch jar>:bin" tutorial.MainH4
```

- Of course in both scripts you have to change the path to the Jadex launch jar to the actual value on your system and also change the package, if your package is not name 'tutorial'.
- The scripts assume that your compiled classes are in the *bin* directory, which is the default for eclipse. Your directory structure should look like the following:



Location of start scripts

- Launch your start script (from a console or by double clicking it in the windows explorer or a linux file browser).

Introduction

The Jadex BDI V3 kernel is a Belief-Desire-Intention reasoning engine for intelligent agents. As the name indicates it is the third version of the Jadex BDI kernel. The V1 kernel version was based on XML and Java and introduced an goal-oriented reasoning mechanism embraces the full BDI reasoning cycle including the selection of goals to pursue (goal deliberation) and the realization phase in which different plans can be tried out to achieve a goal.

In BDI kernel V2 the programming model was kept the same but the engine itself was completely rebuilt based on a RETE rule engine operating on BDI rules.

Finally, in V3 the main objective was to create a new programming model that allows fast prototyping and hides as much of the framework as possible. Thus, in the new V3 kernel BDI agents are written in Java only (no XMLs any more) and annotations are used to designate BDI elements. Another important aspect is the much stronger integration of BDI and object oriented concepts in the new kernel, i.e. it becomes much simpler to program BDI agent having a solid background on object-oriented concepts (supporting e.g. inheritance, POJO programming, dependency injection).

This tutorial is a good starting point for agent developers, that want to learn programming Jadex BDI agents in small hands-on exercises. Each lesson of this tutorial covers one important concept and tries to illustrate why and especially

how the concept can be used in Jadex. Nonetheless, the tutorial cannot illustrate all available concepts and the reader is encouraged to also have a look at the example source contained in the distribution.

- Chapter 2, Starting an Agent describes how to setup the Jadex environment properly and how to start a simple agent.
- Chapter 3, Using Plans explains step by step the usage of plans.
- Chapter 4, Using Beliefs introduces beliefs as agent knowledge form.
- Chapter 6, Using Goals shows how goals can be used to capture the agent objectives in an intuitive way.
- Chapter 5, Using Capabilities explains how beliefs, goals and plans can be composed into reusable agent modules.
- Chapter 7, Using Services covers aspects about BDI service integration including goal delegation to other agents.
- Chapter 8, External Processes explains exemplarily the integration of Jadex agents with external processes.
- Chapter 9, Conclusion finally concludes the lessons.

Application Context

In this tutorial a simple translation agent for single words will be implemented. This agent has the basic task to handle translation requests and produce for a given term in some language the translated term in the desired target language. This base functionality will be extended in the different exercises, but it is not our goal to build up a translation agent, that combines all the extensions, because this would lead to difficulties concerning the complexity of the agent. Instead this tutorial will concentrate on setting up simple agents that explain the Jadex concepts step by step.

How to Use This Tutorial

- Work through the exercises in order, because later exercises require knowledge from the earlier ones.
- Create a new package for each solution you build that is named in the same way as the exercise (e.g. a1, b1). This helps not to confuse the files of different exercises.
- Help us to make this tutorial better with your feedback. When you find errors or have problems that are directly concerned with the exercise descriptions feel free to let us know or edit the corresponding page directly in the Wiki.
- Whenever you encounter problems with Jadex we would be happy to help you. Please use primarily the Jadex mailing list that can be used for asking questions about Jadex.

Chapter 2. Starting an Agent

Setting up the Jadex environment properly is pretty easy and can be done in a few simple steps. The Jadex distribution should be extracted to some local directory, called `JADEX_HOME` here. The easiest way to start the platform is by using the operating dependent scripts in the main directory called `jadex.bat` and `jadex.sh` respectively.

To start the platform via Java you basically have two options. The first is by setting the Java `CLASSPATH` variable by hand to all the jars contained in the `JADEX_HOME/lib` directory. The second one is by using the `-jar` option when starting via the `java` command.

The alternative commands for launching the Jadex Standalone platform are:

```
java jadex.standalone.Platform
```

or

```
java -jar libjadex-platform-standalone-launch-3.0.0-RC1.jar
```

(As the name of the jar depends on the current release please be sure to use the correct version).

Exercise A1 - Creating a Project

Start the Jadex platform with the command explained above. After some short time the Jadex Control Center (JCC) should show up with its user interface.

The JCC provides a project management facility that simplifies working on specific applications. Basically, a project contains settings about the used project folders as well as miscellaneous tool and user interface settings. All your settings - as window settings, added paths and so on - will be stored in a `.project.xml` file and additional properties-files will be created automatically to store the individual settings for the plugins you are using.

To add files to your project, hit the “Add Path” button. For saving the project settings on disk select “Save Settings” from the “File”-menu. Additionally, the settings are also saved automatically when shutting down the JCC via its window close button or “Exit” from the “File”-menu. You can also use multiple projects with different settings. To switch projects simply click on “Load Settings from File” in the “File”-menu and choose the settings file of the project you want to work with.

Verify project behaviour

In order to verify that your project has been set up correctly you should perform some visible changes such as changing the JCC’s window size or switching to another plugin than the starter. After that you should shut down the JCC and

platform and restart it. If the project has been created correctly, the JCC will show up in exactly the same state you left it, i.e. the last active plugin will be activated and the gui settings are remembered, too.

Exercise A2 - Executing Example Applications

The Jadex distribution already contains a couple of example applications. The objective of this exercise consists in trying out the examples and also in roughly understanding their application purpose. Open the JCC and select the starter view as described in exercise A1.

On the left hand side of the starter you can see the agent and application files that can be loaded in a tree like structure. As top-level elements the starter can handle jar files or directories representing the root of Java packages (often named “classes”, “bin” or “build”). Since the newer versions of Jadex a default project is automatically loaded at startup so that you can already see example folders in that panel.

In case you want to add a new directory with agent models, you should choose the



button (“Add Path”) and browse to the corresponding directory you want to add. All Jadex examples are contained in the different ‘jadex-applications-xyz.jar’ files in the JADEX_HOME/lib directory. The content of the new jar will be added as new node to the model tree.

At the bottom you can now see this jar-file being scanned (if the scanning option has not been disabled). Next, you can expand the model tree by double clicking on the corresponding top-level node, which represents the jar-file. After opening the folders “jadex” and “examples” you will see the different example folders containing several different kinds of single- and multi-agent applications. Concretely the following BDI V3 example applications are currently available:

- **Blockworld:** Stacking blocks on a table to reach a specific target configuration of blocks.
- **Cleanerworld:** Simulated cleaner robots collecting waste at day and patrolling at night.
- **Garbagecollector:** Simplified version of the cleaner task using a grid world.
- **Helloworld:** Very simple agent that prints “hello world” to the console output and then kills itself.
- **Marsworld:** Cooperative exploitation of ore on mars by different kinds of robots.
- **Puzzle:** An agent that tries to solve a puzzle by trying out moves and taking them possibly back.
- **Shop:** Example for BDI with services. Buying virtual goods from seller agents.
- **University:** Simple implementation of teaching example from the book “Developing Intelligent Agent Systems: A Practical Guide” written by L.

Padgham and M. Winikoff.

Starting an example can in most cases be done by opening the corresponding folder and searching for an application file (a file ending with “.application.xml”). Such an application definition has the purpose to start-up all relevant application agents and may also setup further application components. In case there is no application file the example can be started by selecting the agent directly (e.g. HelloworldBDI or PuzzleBDI).

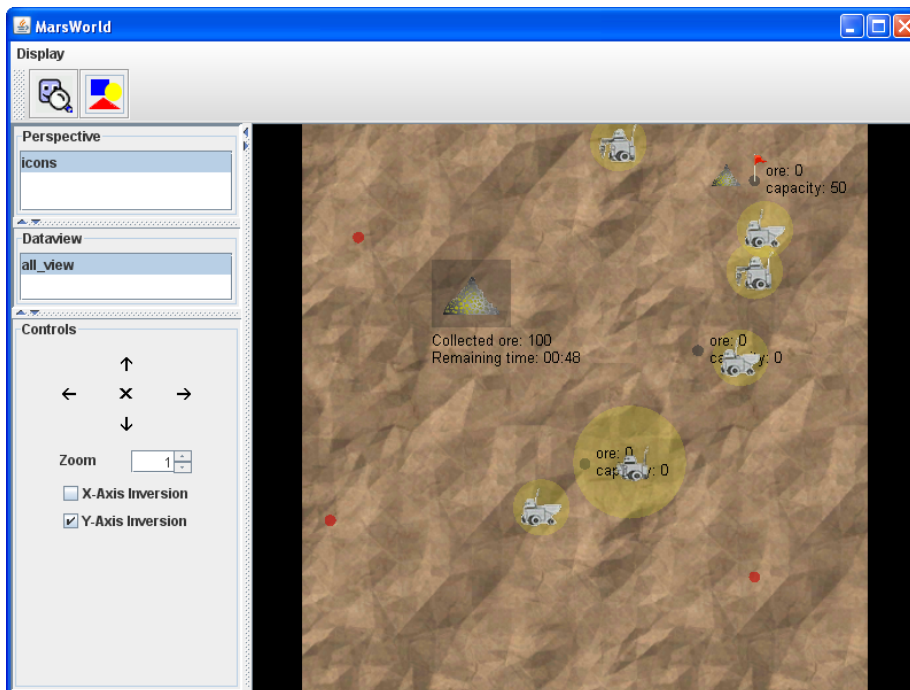


Figure 1 The marsworld example

In the same way you can add other paths and execute your own applications and agents later. In this case you will not add jar-files, but the root directory of your packages.

Explain example behaviour

Choose one of the more complex examples (e.g. marsworld) for a more detailed analysis. Select the application of the example and read through its documentation shown in the starter panel. Then look into the selected example directory and read also the documentation of the agents which belong to that application. Finally, open the source code of these agents in your source code editor and try to grasp roughly of what they are comprised. Write down a (simple!) explanation how the multi-agent system and the involved agents work.

Exercise A3 - Create first simple Jadex agent

Open a source code editor or an IDE of your choice and create a new package called `a1` and an agent class called `TranslationBDI.java` (cf. Figure “A1 Translation agent”).

The agent is a normal Java class that uses the `@Agent` annotation to state that it is an agent. Also please note that it is currently required that the Java file ends with “BDI”. Otherwise it will not be recognized as BDI agent. Optionally, the `@Description` annotation can be used to specify a documentation text that is displayed when the agent is loaded with the JCC.

```
package a1;

import jadex.micro.annotation.Agent;
import jadex.micro.annotation.Description;

@Agent
@Description("The translation agent A1. <br> Empty agent that can be loaded and started.")
public class TranslationBDI
{
}
```

A1 Translation agent

Start your first Jadex agent

Start the JCC and use the “Add Path” button explained above to add the root directory of your example package. Then open the folder until you can see your file “TranslationBDI”. The effect of selecting the input file is that the agent model is loaded.

When it contains no errors, the description of the model, taken from the `@Description` annotation, is shown in the description view. In case there are errors in the model, correct the errors shown in the description view and restart the platform (class reloading is not supported).

Below the file name, the agent name and its default configuration are shown. After pressing the start button the new agent should appear in the agent tree (at the bottom left). It is also possible to start an agent simply by double-clicking it in the model tree.

Please note that when you use a double-click on the model name in the left tree view to start an agent, the settings on the right will be ignored.

Plans

Plans play a central role in Jadex, because they encapsulate the recipe for achieving some state of affair. A plan defines two aspects.

In the **head** of the plan (i.e. in its @Plan annotation) meta information about the plan is defined. This means that in the plan head several properties of the plan can be specified, e.g. the circumstances under which it is activated and its importance in relation to other plans.

The **body** of a plan contains the concrete instruction that should be carried out. The concrete representation of a plan in Jadex can vary as it is the case for beliefs and goals as well. The rationale behind this is that we wanted to achieve language orthogonality between BDI and object oriented concepts. For this reason it is possible to use the following elements as a plan by adding a @Plan annotation to it:

- **Method:** in this case not all plan aspects can be used, e.g. no pre- or context conditions are possible)
- **Inner Class:** in case of a non-static inner class allows for easy access of agent beliefs and fields
- **Class:** facilitates reuse in different agents and projects

For a plan, the triggering events and goals can be specified in the plan head to let the agent know what kinds of events this plan can handle. When an agent receives an event, the BDI reasoning engine builds up the so called applicable plan list (that are all plans which can handle the current event or goal) and candidate(s) are selected and instantiated for execution.

Often a plan does some action and then wants to wait until the action has been done before continuing (e.g. dispatching a subgoal). Therefore a plan can use one of the various waitFor() methods of the plan API, that come in quite different flavors. The plan API can be retrieved as an object via two mechanisms. First, the @PlanAPI annotation can be used above a field of type IPlan in plan classes. The engine will automatically inject the plan API when a plan instance is created. When using a method as plan this is not possible. Hence, the signature of the plan method can be used to retrieve the plan API just by adding a parameter of type IPlan. Please note that in Jadex methods that are invoked by the framework can have any signature. The engine will do its best to automatically determine which values are expected and set them as parameter values. If the engine does not find a suitable value of a given type the value will be null.

Exercise B1 - A Plan as Normal Java Class

In this exercise we will use a plan for translating words from English to German. Create a new TranslationBDI.java file by copying the file from the last lecture.

Creating the Plan

Create a new file called TranslationPlan.java responsible for a basic word translation with the following properties:

- Content of the plan class:

```

@Plan
public class TranslationPlan
{
    protected Map<String, String> wordtable;

    public TranslationPlan()
    {
        // Init the wordtable and add some words
    }

    @PlanBody
    public void translateEnglishGerman()
    {
        // Fetch some word from the table and print the translation
    }
}

```

- In this first version we will use a very simple plan that does not allow for translating words on request. Instead we here just use a hash table as kind of dictionary for a few word pairs. The dictionary should be created in the constructor and some word pairs should be added.
- In the body method (the name of the method and its signature does not matter, the annotation is important) we just look up one word and print the translation in the form:
System.out.println("Translated:" + eword + " - " + gword);
letting eword and gword being the English and German words respectively.

Adding the plan to the agent

- Add the annotation to the agent class: @Plans(@Plan(body=@Body(TranslationPlan.class)))
- Add a field called bdi to the agent class and annotate it with @AgentFeature. The field should be of type IBDIAgentFeature. This will let the engine automatically inject the bdi agent (api) to the pojo agent class.
@AgentFeature protected IBDIAgentFeature bdi;
- Add an agent body method that is automatically invoked when the agent is started and adopt a plan using

```

@AgentBody
public void body()
{

```

```
bdi.adoptPlan(new TranslationPlan());
}
```

Starting and testing the agent

Create a translation agent via the Jadex Control Center and observe the output. You should see it printing the translated word.

Exercise B2 - A Plan as Inner Class

In the lecture we will use an inner class as plan instead of an extra plan class. The functionality remains the same. Again, copy the translation agent class from the last lecture and apply the following changes:

- Remove the @Plans annotation from the class file completely. Only extra plan classes need to be declared in this way. Inline elements will be found automatically when scanning the class file.
- Copy the contents from the plan class of the last lecture in the agent class file (as inner class).

```
@Agent
@Description("The translation agent B2. <br> Declare and activate an inline plan (declared
public class TranslationBDI
{
    ...

    @Plan
    public class TranslationPlan
    {
        ...
    }
}
```

- Adapt the adoptPlan() method call to use the new inner class

Starting and testing the agent

Start the agent as explained in the preceding exercise. Observe if the same output is produced.

Exercise B3 - Plan as Method

Once again, in this lecture the same functionality will be created. But this time, the plan will be represented as method. This can be very helpful, if the plan is

rather simple. Furthermore, using methods as plans helps reducing the number of classes in a project.

Changing the agent

Again, copy the agent file from the last lecture and do the following:

- Copy the word table field from the inner to the agent class
- Copy the init code for the word table to the newly created init method of the agent

```
@AgentCreated
public void init()
{
    ...
}
```

- Adapt the adoptPlan() method call to

```
bdi.adoptPlan("translateEnglishGerman");
```

Instead a plan object we just give the name of the method representing the plan.
- Create a method as plan using the following code

```
@Plan
public void translateEnglishGerman()
{
}
}
```

- Then remove the inner plan class completely.

Starting and testing the agent

Test and verify that the agent behavior is the same as in the last exercise.

Exercise B4 - Using Other Plan Methods

In this exercise we will explore other plan methods. Besides the already known body method three other plan lifecycle methods exist, which are called respectively when the plan passes successfully (@PlanPassed), fails with exception (@PlanFailed) or is aborted (@PlanAborted) e.g. when the context of plan becomes invalid.

This time, we need a translation agent with an inner plan class to be able to add the aforementioned method. Hence, it is most convenient to take the class from

exercise B2 as starting point and copy its content to the new file. Afterwards we need to apply the following changes:

Changing the agent

- Add a try-catch-block to the `adoptPlan()` call and wait for the plan to be finished using `get()` at the end of the invocation. The `get()` turns the future based asynchronous call into a synchronous one. For more information about asynchronous programming with futures in Jadex please refer to the AC User Guide. The agent body method should look like this:

```
try
{
    bdi.adoptPlan(new TranslatePlan()).get();
}
catch(Exception e)
{
    e.printStackTrace();
}
```

Changing the plan

- Add the three lifecycle methods to the plan inner class in the following way:

```
@PlanPassed
public void passed()
{
    System.out.println("Plan finished successfully.");
}
```

```
@PlanAborted
public void aborted()
{
    System.out.println("Plan aborted.");
}
```

```
@PlanFailed
public void failed(Exception e)
{
    System.out.println("Plan failed: "+e);
}
```

- Modify the plan body to throw an exception:

```

@PlanBody
public void translateEnglishGerman()
{
    throw new PlanFailureException();
    // System.out.println("Translated: dog - " + wordtable.get("dog"));
}

```

Starting and testing the agent

After starting the agent you should observe that due to the exception in the plan body the failed method is invoked. In the agent body the exception is rethrown when the `get()` on the result future of `adoptPlan()` is invoked. Also try out what happens when you do not throw the exception in the plan body.

Exercise B5 - Plan Context Conditions

Besides the lifecycle methods that have been introduced in the former exercise a plan may also have a pre- and/or a context condition. The precondition is evaluated before a plan is going to be executed and if it evaluates to false to plan will be excluded. In contrast, the context condition has to hold during all the time a plan is executing. If it turns to false at some point in time, the plan will be aborted. In this exercise we will learn how a context condition can be used.

Creating the agent

As preparation we can copy the agent from the last exercise and modify the following:

- We add a field named `context` of boolean type and put an `@Belief` annotation above it. Details about the meaning of beliefs will be explained in the next chapter.

```

@Belief
protected boolean context = true;

```

- To access the `waitFor` methods we add another `@AgentFeature` of type `IExecutionFeature` to our agent.

```

@AgentFeature
protected IExecutionFeature execution;

```

- In the agent body method we do not wait until plan completion. Instead we wait for one second and afterwards set the `context` field to false.

```

try
{
    bdi.adoptPlan(new TranslatePlan());
    execution.waitForDelay(1000).get();
    context = false;
    System.out.println("context set to false");
}
catch(Exception e)
{
    e.printStackTrace();
}

```

Changing the plan

- In the inner plan class we add a field for the plan API and a method for the context condition. The plan API is of type IPlan and needs the @PlanAPI annotation. This ensures that the API will be automatically injected to the field when the plan is created. The context method should have a @PlanContextCondition annotation. Furthermore, we want the condition to be reevaluated whenever the belief context changes. This is achieved by adding a dependency to the context belief via the beliefs declaration in the annotation. The method itself should simply return the value of the context field.

```

@PlanAPI
protected IPlan plan;

@PlanContextCondition(beliefs="context")
public boolean checkCondition()
{
    return context;
}

```

- Finally, the plan logic has to be changed in order to be active a longer period of time. To achieve this we first print 'Plan started' and then use a waitFor() statement to let the plan wait for 10 seconds. The wait methods are accessible via the injected plan API. Finally, we add a print statement with 'Plan resumed'.

```

@PlanBody
public void translateEnglishGerman()
{
    System.out.println("Plan started.");
}

```

```

plan.waitFor(10000).get();
System.out.println("Plan resumed.");

System.out.println("Translated: dog - " + wordtable.get("dog"));
}

```

Starting and testing the agent

This time the agent should start executing the plan but automatically abort it after one second when the context becomes invalid. To verify this you should check if you see the print of the plan aborted method.

Using Beliefs

An agent's beliefbase represents its knowledge about the world. The agent is aware of this knowledge and can use it to reason. On the one hand, the beliefs can drive the actions of an agent by e.g. initiating goals or plans and on the other hand the beliefs also control the ongoing behaviour by determining when a goal is achieved or by rendering plans applicable or not. In Jadex BDI V3 beliefs are represented in an object-oriented way as:

- **Field:** This representation is the most common one and treats a field of an agent as its belief. The field can be of any type whereby special support exists for collection types and arrays.
- **Getter/Setter Method Pair:** Using a getter/setter pair as belief allows for putting additional logic into the getter or setter. It also allows for using beliefs without a field.
- **Unimplemented Getter/Setter Method Pair:** Using a getter/setter pair that is declared native, i.e. without implementation, can be used to have abstract beliefs in capabilities. Such abstract beliefs can already be used in the capability but its representation is assigned by the using agent (or capability). This is useful if the belief should be shared among different capabilities.

If you already know the former versions of Jadex BDI, you may be aware of the distinction between beliefs and belief sets. This distinction is not necessary in V3 any longer and instead all elements are marked with the @Belief annotation. Please note that in case of a getter/setter pair it is required to add @Belief to both methods. The function of making things beliefs is that the agent becomes aware of changes of these elements. This means that if the value of a belief is set to a new value the agent recognizes this change and can act according to this change.

Exercise C1 - Belief Triggering Plan

In this exercise we will develop a translation agent that checks if only good word pairs are added to his dictionary. For this purpose we will make the wordtable become a belief and create a check plan that is activated always when the dictionary changes. This time we start with a fresh agent file and do the following:

- Create a new TranslationBDI agent Java class file and add the @Agent annotation to the class
- Add two fields to the class representing the agent API and the wordtable

```
@Agent
protected BDIAgent agent;
```

```
@Belief
protected Map<String, String> wordtable;
```

- Add an init method for the agent (using @AgentCreated) and create the wordtable map in it. Additionally, add some example word pairs as usual. As last entry add the following colloquial word pair, which we will check for in the check plan.

```
wordtable.put("bugger", "Flegel");
```

- Add a method based plan that reacts to the belief and checks whether an added word is allowed. If a colloquial word is added print a warning to the console. Please note that the added wordpair is automatically passed to the plan method whenever the wordtable changes.

```
@Plan(trigger=@Trigger(factadded="wordtable"))
public void checkWordPairPlan(ChangeEvent event)
{
    ChangeInfo<String> change = ((ChangeInfo<String>)event.getValue());
    if(change.getInfo().equals("bugger"))
        System.out.println("Warning, a colloquial word pair has been added: "+change.getInfo());
}
```

Starting and testing the agent

Create a translation agent via the Jadex Control Center and observe the output. You should see it printing the warning.

Exercise C2 - Dynamic Beliefs

Besides normal beliefs it is sometimes helpful to have a belief that directly depends on other beliefs and is automatically reevaluated whenever one of the beliefs changes it relies on. For such dynamic beliefs it is required that they are fields with an init expression directly in its declaration, i.e. e.g. *private String name = othertype+id*, assuming that othertype and id are other beliefs.

- Create a TranslationBDI class by copying it from the last exercise.
- Change the belief definition in two ways. First already create the wordtable as part of the declaration and second add a new belief named alarm of type boolean. The alarm expression should check if the wordtable contains the key 'bugger'.

```
@Belief
protected Map<String, String> wordtable = new HashMap<String, String>();
```

```
@Belief(dynamic=true)
protected boolean alarm = wordtable.containsKey("bugger");
```

- Change the plan to now react on changes of the alarm belief and in case of an alarm just add a print statement in the plan body. The ChangeEvent object can be injected into all plan methods. Unlike using the changed fact itself like it was done with the wordpair in Exercise C1, using the change event has the advantage that you can also inspect the old value. For this purpose, the change event contains a ChangeInfo object.

```
@Plan(trigger=@Trigger(factchangeds="alarm"))
public void checkWordPairPlan(ChangeEvent event)
{
    ChangeInfo<Boolean> change = (ChangeInfo<Boolean>)event.getValue();
    // Print warning when value changes from false to true.
    if(Boolean.FALSE.equals(change.getOldValue()) && Boolean.TRUE.equals(change.getValue()))
    {
        System.out.println("Warning, a colloquial word pair has been added.");
    }
}
```

Starting and testing the agent

Start the agent and verify that it behaves the same way as in the last exercise.

Exercise C3 - Getter/Setter Belief

In this and the following exercises in this chapter we will use a different example as it better fits to show further belief features. The example is a very simple clock which is able to print the current time to the standard out.

- Create a file called ClockBDI and add the `@Agent` annotation to the class.
- Add two fields. One called `time` of type `long` and another called `formatter` of type `SimpleDateFormat`. The formatter can be initialized with `new SimpleDateFormat("dd.MM.yyyy HH:mm:ss")`. It will be used to print the current time and date.
- Add a getter and a setter method for the time belief:

```
@Belief
public long getTime()
{
    return time;
}
```

```
@Belief
public void setTime(long time)
{
    this.time = time;
}
```

- Add a method based plan that reacts on fact changes of the time belief. The plan body should just print out the time belief.

```
@Plan(trigger=@Trigger(factchanges="time"))
protected void printTime()
{
    System.out.println(formatter.format(getTime()));
}
```

- Add an agent body in which you call `setTime()` with the current time. The current time can be obtained using `System.currentTimeMillis()`.

Starting and testing the agent

Start the agent and check that it prints out the current time.

Exercise C4 - Getter/Setter Belief without Field

This lecture will show that you can use also a getter/setter belief without an underlying field representation.

- Copy the clock agent from the last lecture and remove the time field.
- Modify the getter method to just return the current time via `System.currentTimeMillis()`.
- Modify the setter method to just do nothing. Optionally, you can also delete the parameter of the method.
- In the body of the agent just call `setTime()`.
- The plan remains completely the same as in the last exercise.

Starting and testing the agent

Start the agent and check that it prints out the current time. Think about why it works? You just call an empty method (`setTime()`), don't you?

Exercise C5 - Belief with Update Rate

In order to print out the current time regularly and not just once we will use a belief with update rate. This means that the value of the belief is automatically reevaluated in certain time intervals.

- Copy the agent file from the last exercise and keep the plan as well as the formatter. Everything else can be deleted (also the body method of the agent).
- Add a belief named `time` of type `long` and assign it `System.currentTimeMillis()`. Furthermore set the update rate in the belief annotation to 1000, i.e. one second.

```
@Belief(uptodate=1000)
protected long time = System.currentTimeMillis();
```

Starting and testing the agent

Start the agent and check that it prints out the current time every second.

Using Goals

Goal-oriented programming is one of the key concepts in the agent-oriented paradigm. It denotes the fact that an agent commits itself to a certain objective and maybe tries all the possibilities to achieve its goal. A good example for a goal that ultimately has to be achieved is the safe landing of an aircraft. The agent will try all its plans until this goal has succeeded, otherwise it will not have the opportunity to reach any other goal when the aircraft crashes.

When talking about goals one can consider different kinds of goals. What we discussed above is called an achieve goal, because the agent wants to achieve a certain state of affairs. Similar to an achieve goal is the query goal which aims at information retrieval. To find the requested information plans are only executed when necessary. E.g. a cleaner agent could use a query goal to find out where the nearest wastebin is.

Another kind is represented through a maintain goal, that has to keep the properties (its maintain condition) satisfied all the time. When the condition is not satisfied any longer, plans are invoked to re-establish a normal state. An example for a maintain goal is to keep the temperature of a nuclear reactor below some specified limit. When this limit is exceeded, the agent has to act and normalize the state.

The fourth kind of goal is the perform goal, which is directly related to some kind of action one wants the agent to perform. An example for a perform goal is an agent that as to patrol at some kind of frontier.

The goal representation has changed a bit in V3 compared to the former versions of Jadex. In Jadex V3 a goal is a Java POJO, i.e. a user defined object. Furthermore, the different kinds of goals need not to be specified explicitly. Instead, it is sufficient to just use the type of conditions that are required to produce the desired behaviour.

Concretely, a goal type can come in the following flavors in V3:

- **Inner Class:** If a goal is private to an agent it is often elegant and helpful to use an inner class to represent the goal type. The inner class has natural access to the fields and beliefs of the agent which makes programming less complex. In some cases it is also required to use a static inner class. In this case the aforementioned advantage is not existent, but you could pass the agent as explicit argument in the constructor to gain access to the agent aspects.
- **Class:** A goal can also be represented as a normal Java class. In this case there is no direct connection to the agent available and one again has to pass whatever is need via the constructor call or other methods.

D1 - Using a Top-Level Goal

The first thing we will try out in this exercise is dispatching a top-level goal. The difference between a top-level and a subgoal can be understood as its part in the BDI goal-plan hierarchy.

For each goal different plans can be tried out, which in turn may produce subgoals to fulfill parts of their work.

These subgoals again may have other subgoals leading to the already mentioned goal-plan tree. In this sense a top-level goal is just a goal that has no parent, i.e. which is on the top level of the hierarchy.

- We create a new TranslationBDI agent Java file and add the @Agent annotation to the class itself. Furthermore we need two fields, one for the agent API called agent and another one for the wordtable of type Map<String, String>. As you will remember, we need to add the @Agent annotation to the agent field and we make the wordtable a belief of the agent by adding @Belief.
- Create and setup the wordtable in the init method of the agent. The method needs to use the @AgentCreated annotation to be recognized as init method of the agent. Furthermore, add some word pairs to the word table in the method, too.
- Now, we create a new goal type as inner class of the agent. To make the class a goal the @Goal annotation has to be added to the class definition. We name the class Translate and give it two String fields, one called eword and one called gword for the English and German word respectively. The English word will be passed as parameter to the goal and the German word will be delivered as result. Hence, we also add a constructor that takes the English word as parameter and saves it in the corresponding field.
Finally, we add getter and setter methods for both fields.

```

@Goal
public class Translate
{
    protected String eword;

    protected String gword;

    public Translate(String eword)
    {
        this.eword = eword;
    }

    public String getEWord()
    {
        return eword;
    }

    public String getGWord()
    {
        return gword;
    }

    public void setGWord(String gword)
    {
        this.gword = gword;
    }
}

```

```
}  
}
```

- Besides the goal we also need a plan to handle the goal. For this purpose we add a new method plan called translate. In the @Plan annotation as trigger the goal has to be specified.

Furthermore, we define a parameter named goal of type Translate for the method. This will allow us to fetch the goal the plan is executed for and extract the word it should translate.

The plan body should just fetch the word via goal.getEWord() and feed it to the wordtable to look up the translation. Afterwards, the gword should be set as result in the goal via goal.setGWord(gword).

```
@Plan(trigger=@Trigger(goals=Translate.class))  
protected void translate(Translate goal)  
{  
    String eword = goal.getEWord();  
    String gword = wordtable.get(eword);  
    goal.setGWord(gword);  
}
```

- Finally, the agent body method has to be created. In this method we actually create an instance of our new goal type and dispatch it as top-level goal. We then wait for the result and print it out.

```
@AgentBody  
public void body()  
{  
    String eword = "cat";  
    Translate goal = (Translate)agent.dispatchTopLevelGoal(new Translate(eword)).get();  
    System.out.println("Translated: "+eword+" "+goal.getGWord());  
}
```

Starting and testing the agent

After starting the agent it should print out the word for which we have created and dispatched a goal.

D2 - Using Parameters and Results

The approach used in the last exercise is perfectly feasible, but has a few minor drawbacks. One point is that it is not very comfortable to extract the parameters of a goal manually. The same drawback exists for the goal result in the above

solution. Using goal parameters and results this can be overcome in a very simple way.

- As starting point we use the agent file of the last exercise, copy it and modify some minor aspects.
- First, remove the getter and setter methods in the goal. Instead a `@GoalParameter` annotation to the `eword` field and a `@GoalResult` parameter to the `gword` field.

```
@Goal
public class Translate
{
    @GoalParameter
    protected String eword;

    @GoalResult
    protected String gword;

    public Translate(String eword)
    {
        this.eword = eword;
    }
}
```

- In the agent body method we can change the invocation of the goal dispatch to directly retrieve the result of the goal. The framework automatically scans the goal for a result annotation and delivers the value of that field (or getter method) to the caller.

```
@AgentBody
public void body()
{
    String eword = "cat";
    String gword = (String)agent.dispatchTopLevelGoal(new Translate(eword)).get();
    System.out.println("Translated: "+eword+" "+gword);
}
```

- In the plan we can now also simplify the parameter handling by just declaring a method parameter for our goal parameter. The engine will automatically match goal parameters with plan parameter declarations and deliver them as needed.

```
@Plan(trigger=@Trigger(goals=Translate.class))
protected String translate(String eword)
{
```



```
    return wordtable.get(eword);
}
```

Starting and testing the agent

After starting the agent it should behave in the same way as in the last exercise.

D3 - Goal Retry

A goal is different from a plan because it describes an objective without exactly stating how it should be achieved. This means that different plans can be tried out to finally reach a given objective. Besides the specification of plans that can in principle help achieving a goal, the means-end reasoning process can be adjusted in many ways using the various BDI flags. One of the fundamental flags is the retry setting which defines if another plan can be tried out when the first one fails or does not achieve the goal completely. In this exercise we will try out the retry behaviour by using two plans from which the first will fail.

- Create a TranslationBDI file by copying from the last exercise.
- Modify the existing plan to throw a PlanFailureException after having printed out a text message, e.g. "Plan A".
- Add a second method plan named translateB that should look the same like first, i.e. the same plan annotation, same parameter. In the plan print "Plan B" and return the translated word from the map.

```
@Plan(trigger=@Trigger(goals=Translate.class))
protected String translateA(String eword)
{
    System.out.println("Plan A");
    throw new PlanFailureException();
}
```

```
@Plan(trigger=@Trigger(goals=Translate.class))
protected String translateB(String eword)
{
    System.out.println("Plan B");
    return wordtable.get(eword);
}
```

Starting and testing the agent

After starting the agent you should see that first plan A and afterwards plan B is executed. Think about what happens if the second plan would also throw an exception? Also ask yourself what the agent would do when the plans would be declared in different order or if the second would have a higher priority? Verify your thoughts by trying these things out in the source code.

D4 - Goal Creation Condition

Until now we have created goals manually, but often one also wants to create a goal in response to a belief change. This we will try out in the following. Concretely, we want our translation agent to create a new translation goal whenever our belief with the English word changes.

- Again we start by copying the code from lecture D2.
- We add a belief for the current English word of type String named eword.

```
@Belief
```

```
protected String eword;
```

- In the goal class we add a creation condition. Here, we just add the corresponding annotation @GoalCreationCondition to the constructor of the goal. Additionally, we have to state which change provoke the creation of a new goal. In this case we just say that always when our eword belief is modified we want to obtain a new goal.

```
@GoalCreationCondition(beliefs="eword")
```

```
public Translate(String eword)
```

```
{  
    this.eword = eword;  
}
```

- In the agent body method we just assign different values to the eword belief.

```
@AgentBody
```

```
public void body()
```

```
{  
    eword = "cat";  
    eword = "milk";  
}
```

Starting and testing the agent

After starting the agent you should see that for each belief assignment a new print out is produced. In this exercise we have used the constructor as creation condition because we wanted to create a goal on every change of that belief. But where to place condition code if we want to perform some checks and create a goal only in certain circumstances?

D5 - Goal Recur

The process of means-end reasoning is started when a goal becomes active in the agent. It continues to execute plans until the goal is achieved or no more plans can be executed. If this is the case and no more plans exist although the goal has not been fulfilled yet the recur setting determines how is proceeded. Per default, goals are considered as short-term objectives and recur is turned off. The corresponding behaviour is that the goal fails with an exception and reasoning is finished.

If this should not happen and the goal should persist in the agent even if it cannot be achieved immediately the goal can be made a long-term goal by turning on the recur flag. As result the goal will be paused until the recur condition indicates that means-end reasoning should be executed again. This also means that the set of already tried plans is cleared and means-end reasoning will be performed in the same way as in the first round.

In this exercise we will change the translation agent to keep a translation goal even when a word is not contained in the dictionary. it will then be paused until the corresponding word pair has been added. As recur condition we will use any detected changes to the agent's dictionary.

- Create a new TranslationBDI file as copy of the solution of D2.
- Turn on the goal recur mode by setting recur=true in the @Goal annotation.

```
@Goal(recur=true)
public class Translate
```

- Add a recur condition to the goal that reacts on changes to the dictionary map. To achieve this create a new method named checkRecur with boolean return value. Add the @GoalRecurCondition to the method and set beliefs to wordtable.

```
@GoalRecurCondition(beliefs="wordtable")
public boolean checkRecur()
{
    return true;
}
```

- Change the plan body to throw a `PlanFailureException` when a word is not contained in the dictionary.

```

@Plan(trigger=@Trigger(goals=Translate.class))
protected String translate(String eword)
{
    String ret = wordtable.get(eword);
    if(ret==null)
        throw new PlanFailureException();
    return ret;
}

```

- In the agent body we create the translation goal and additionally schedule a timed action that will automatically add the word that we will look up. For this purpose we use the method `scheduleStep` which is available via the agent API. As parameters we first add a new component step which adds the new word pair to the dictionary using `wordtable.put("bugger", "Flegel")`. The second parameter is the delay for the step, here we use 3000 to state that we want it to be executed in 3 seconds. After having scheduled the step we create and dispatch a translation goal with "bugger" as translation request.

```

@AgentBody
public void body()
{
    agent.scheduleStep(new IComponentStep<Void>()
    {
        public IFuture<Void> execute(IInternalAccess ia)
        {
            wordtable.put("bugger", "Flegel");
            return IFuture.DONE;
        }
    }, 3000);

    String eword = "bugger";
    String gword = (String)agent.dispatchTopLevelGoal(new Translate(eword)).get();
    System.out.println("Translated: "+eword+" "+gword);
}

```

Starting and testing the agent

Start the agent and check whether the goal is reactivated by the recur condition triggers.

D6 - Maintain Goals

As we have illustrated in the introduction of this lecture, in Jadex different goal types can be used. In the new BDI V3 we have changed the way these goal types can be used. Instead of declaring a goal type explicitly (as in BDI V1 & V2) in V3 it is sufficient to use the corresponding condition types, i.e. a maintain condition in case of a maintain goal. In the translation example we will use a maintain goal to restrict the number of word pairs in the dictionary.

- We start by creating a TranslationBDI Java file by copying it from the last exercise.
- We delete the translation goal and instead create a new MaintainStorageGoal as inner class. In the goal annotation we set the exclude mode to never (`excludemode=ExcludeMode.Never`). This allows a plan to be executed again and again without being excluded.
- Furthermore, we add two methods (without parameters and boolean return value), one called `maintain` and the other one `target`. To the first we add the `@GoalMaintainCondition(beliefs="wordtable")` and to the second `@GoalTargetCondition(beliefs="wordtable")` annotation. This leads to a reevaluation of the conditions whenever the dictionary `wordtable` changes. We want the `maintain` condition to trigger when the number of entries in the dictionary exceeds. As consequence a plan for removing entries is triggered and it will remove entries until less than 3 word pairs are contained.

```
@Goal(excludemode=ExcludeMode.Never)
public class MaintainStorageGoal
{
    @GoalMaintainCondition(beliefs="wordtable")
    protected boolean maintain()
    {
        return wordtable.size()<=4;
    }

    @GoalTargetCondition(beliefs="wordtable")
    protected boolean target()
    {
        return wordtable.size()<3;
    }
}
```

- As next step we add a method plan called `removeEntry` without parameters and return value. In the plan annotation add the `MaintainStorageGoal` as trigger. In the method just fetch an arbitrary entry from the dictionary `wordtable` and remove it and print out which entry has been removed.
- Finally, we add an agent body method called `body` using the `@AgentBody`

annotation. In the method we first create and dispatch a maintain storage goal as top-level goal of the agent. Next, we create the wordtable hash table and add four different word pairs.

To activate the maintain goal we add a new word pair every two seconds. For this purpose we create a component step that declares an integer field cnt (as counter).

In the execute method of the step we add a new word pair using the cnt to make it unique each time. The cnt should be increased after it has been used for one word pair.

Afterwards, we print the contents of the dictionary and use the wait method of the agent to reschedule the step after it has been executed. In order to activate the step in the agent we also use the waitFor method at the end of the agent body method.

```
@AgentBody
public void body()
{
    agent.dispatchTopLevelGoal(new MaintainStorageGoal());

    wordtable = new HashMap<String, String>();
    wordtable.put("milk", "Milch");
    wordtable.put("cow", "Kuh");
    wordtable.put("cat", "Katze");
    wordtable.put("dog", "Hund");

    IComponentStep<Void> step = new IComponentStep<Void>()
    {
        int cnt = 0;
        public IFuture<Void> execute(IInternalAccess ia)
        {
            wordtable.put("eword_#" + cnt, "gword_#" + cnt);
            cnt++;
            System.out.println("wordtable: " + wordtable);
            agent.waitFor(2000, this);
            return IFuture.DONE;
        }
    };

    agent.waitFor(2000, step);
}
```

Starting and testing the agent

Start the agent and observe the print outs on the console. You should see that constantly entries are added. When five entries are in the dictionary the maintain goal is activated and entries get removed until only two remain. The process starts in the same way again.

Reusability is a key aspect of software engineering as it allows for applying a once developed solution at several places. Regarding BDI agents reusability can be achieved by two different approaches. First, in BDI V3 it is possible to exploit the existing Java inheritance mechanism and design a base agent class that contains common functionality and is extended by different application agent classes. Of course, this mechanism suffers from the fact that in Java no multi-inheritance is possible. Hence, in case you want to reuse different functionalities these can be encapsulated in so called BDI capabilities. A BDI capability represents a module that may contain belief, goals and plans like a normal agent. Capabilities realize a hierarchical (de)composition concept meaning that it is possible to include any number of subcapabilities that may again represent composite entities.

A module has to provide an explicit boundary which allows for connecting it with an agent or with another module. In contrast to BDI V2, in which it had to be explicitly declared which beliefs, goals and plans are exported and thus visible to the outside of a module, in BDI V3 these specifications have been pushed to the Java level. This means that the visibility modifiers you use in Java determines also if beliefs, goals and plans are visible.

There is basically one additional feature that goes beyond these rules. In order to allow the specification of abstract beliefs, which should be available in the module but are made concrete and are assigned at the level of the outer, i.e. including module, unimplemented beliefs can be specified. Such unimplemented beliefs are represented as native getter/setter pairs without method body. In the outer capability an explicit belief mapping has to be stated which describes the connection of a local and the abstract belief of the submodule.

In Jadex V3 a capability is typically represented as a **Class**:

As a capability should enable reuse it is the normal case to use a separate class file for the module. The module is declared and instantiated as normal field in the agent with corresponding meta information in terms of an annotation.

E1 - Creating a Capability

In this first exercise we just create a capability that encapsulates the translation agent behaviour. The agent itself is reduced to use the capability and dispatch a translation goal from the capability.

- Create a new class file called TranslationCapability and add the @Capability annotation above the class definition.

```

@Capability
public class TranslationCapability
{
    ...
}

```

- Add a belief named wordtable of type Map<String, String> and create an instance of it.
- Add an empty constructor to TranslationCapability, which adds some word pairs to the word table.

For the actual goal we will use an inner class called Translate:

- Create a goal as inner class named Translate. Add two fields of type String named eword and gword to the inner class and annotate them with @GoalParameter and @GoalResult respectively. Also add a constructor which takes the eword as parameter and assigns it to the goal parameter.
- Add a method plan that reacts to the translate goal. It should take the eword as parameter and return the translated word using the normal lookup in the word table. (Annotate with @Plan(trigger=@Trigger(goals=Translate.class)))

Last we will need to create the actual agent:

- Create an empty agent file called TranslationBDI with the corresponding annotation.
- Add a field of type IBDI-AgentFeature called bdi and add the @AgentFeature annotation.
- Add a field called capability of type TranslationCapability, annotate it with @Capability and assign an instance of it.

```

@Capability
protected TranslationCapability capa = new TranslationCapability();

```

- Create an agent body which creates and dispatches a translation goal from the included capability. Wait for the goal to be finished and print out the result of the translation.

```

String eword = "dog";
String gword = (String) bdi.dispatchTopLevelGoal(capa.new Translate(eword)).get();
System.out.printf("Translating %s to %s", eword, gword);

```

Starting and testing the agent

After starting the agent you should see again the print out of the translated word.

E2 - Using an Abstract Belief

In this exercise we will show how an abstract belief can be used. The application idea here is that the agent itself manages the word table (instead of the capability) and provide another plan to search for synonyms in that table. (In this simple case an alternative solution would have been making the word table accessible via public getter/setter methods. But the design here suggests that if more functionalities share a data structure that none of them really owns exclusively that it is better to move it to the agent level).

- Copy both files from the last lecture and perform the following modifications.
- Delete the wordtable belief and the constructor from the capability definitions.
- Add a native getter/setter pair, i.e. an abstract belief named get- and setWordtable.

```
@Belief
public native Map<String, String> getWordtable();
```

```
@Belief
public native void setWordtable(Map<String, String> wordtable);
```

- In the translate plan body replace the direct field access with a get-Wordtable call.
- In the agent file change the capability definition to include the necessary belief mapping.

```
@Capability(beliefmapping=@Mapping(value="wordtable"))
protected TranslationCapability capa = new TranslationCapability();
```

- Add/move the definition of the word table belief to the agent.
- Add an agent init method and create word pairs as well as synonyms. For example the following:

```
@AgentCreated
public void init()
{
    wordtable.put("coffee", "Kaffee");
    wordtable.put("milk", "Milch");
    wordtable.put("cow", "Kuh");
    wordtable.put("cat", "Katze");
    wordtable.put("dog", "Hund");
    wordtable.put("puppy", "Hund");
    wordtable.put("hound", "Hund");
}
```

```

wordtable.put("jack", "Katze");
wordtable.put("crummie", "Kuh");
}

```

- Add a plan called findSynonyms which iterates through the word table and collects synonyms.

```

@Plan
protected List<String> findSynonyms(ChangeEvent ev)
{
    String eword = (String)((Object[])ev.getValue())[0];
    List<String> ret = new ArrayList<String>();
    String gword = wordtable.get(eword);
    for(String key: wordtable.keySet())
    {
        if(wordtable.get(key).equals(gword))
        {
            ret.add(key);
        }
    }
    return ret;
}

```

Using Services

So far we have explored how BDI can be used to define the internal behaviour of an agent. In this part we move on towards multi-agent scenarios and show how a BDI agents can be made to interact with each other. The typical way for realizing interactions with active components is using services. A service is defined by an interface that determines to available methods and a service implementation that can be either a separate class of just part of the agent itself. If you are unfamiliar with services please have a look at the active components user guide .

F1 - Creating a Service

In the first exercise we will equip the translation agent with a corresponding service. We will additionally create a user agent that opens a small user interface. The user interface allows for entering English words that will be translated on request. Internally, the user agent searches for a translation service and delegates the request to it.

- First create a new Java interface called ITranslationService. Add a method called translateEnglishGerman to it. The method should take a String

parameter called `eword` and return a futurized String (`IFuture<String>`).

```
public interface ITranslationService
{
    public IFuture<String> translateEnglishGerman(String eword);
}
```

- Create a Java class called `TranslationBDI` that implements the translation interface. Add the `@Agent` and `@Service` annotations to the class. Furthermore, add a new provided service using the `@ProvidedServices` and in it the `@ProvidedService` annotation. Set the type of the provided service to `ITranslationService`.

```
@Agent
@Service
@ProvidedServices(@ProvidedService(type=ITranslationService.class))
public class TranslationBDI implements ITranslationService
{
    ...
}
```

- Add two fields to the agent. First, we need the agent API that should be injected to a field called `agent` and that is of type `BDAgent`. Second, we need the `wordtable`. As in previous lectures declare it with name `wordtable` and type `Map<String, String>`.
- Add an agent `init` method using the `@AgentCreated` annotation. Create the word table in it and add some word pairs to it.
- Implement the interface method by just looking up the word in the map and returning it via a new future.

```
public IFuture<String> translateEnglishGerman(String eword)
{
    String gword = wordtable.get(eword);
    return new Future<String>(gword);
}
```

- Create a new Java class called `UserAgent`. The user agent should declare also a field called `agent` for the agent API. Additionally, it should add an agent body method (`@AgentBody`) that creates the user interface. To simplify this task the corresponding code is displayed below. Inside of the body method first a thread switch to the Swing thread is performed (using `SwingUtilities.invokeLater`). It is a general Swing requirement that all gui related actions should always be performed only on the Swing thread. Otherwise you might encounter strange behavior due to race conditions that might occur sometimes. Within the `Runnable` that is executed by Swing first a `JFrame` is created. Two textfields and one button are added.

The rest of the code is in charge of displaying the gui at the center of the screen.

```
@Agent
public class UserBDI
{
    @Agent
    protected BDIAgent agent;

    @AgentBody
    public void body()
    {
        SwingUtilities.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame f = new JFrame();
                PropertiesPanel pp = new PropertiesPanel();
                final JTextField tfe = pp.createTextField("English Word", "dog", true);
                final JTextField tfg = pp.createTextField("German Word");
                JButton bt = pp.createButton("Initiate", "Translate");
                f.add(pp, BorderLayout.CENTER);
                f.pack();
                f.setLocation(SGUI.calculateMiddlePosition(f));
                f.setVisible(true);

                ...
            }
        });
    }
}
```

- One last part is missing. If a user enters a word that should be translated and presses the Translate button a service invocation has to be created. For this purpose add an inline action listener to the button and in its actionPerformed method search for a translation service. If found, invoke the translation method and display the result in the other textfield (or the error that occurred). As starting point the code for searching the service is outlined below:

```
SServiceProvider.getServices(agent.getServiceProvider(), ITranslationService.class, Required
.addResultListener(new IntermediateDefaultResultListener<ITranslationService>()
{
    public void intermediateResultAvailable(ITranslationService ts)
    {
        ...
    }
});
```

Starting and testing the agents

From the JCC, start both agents. The user interface should appear after the user agent has been started. Enter a word and press the Translate button. You should see the translated word appearing immediately in the text field below.

F2 - Mapping a Service to Plans

One of the strength of BDI is that it provides a flexible runtime execution by selecting suitable plans at runtime. This concept cannot only be used with goals but also directly with plans. This means we can created plans and just state that these plans realize a service call. In this case an incoming service call is automatically delegated to a suitable plan (checking the pre- and context conditions of the plans. Please note that only one plan is executed and no retries are performed (is this is necessary we need to map the service to a goal and not a plan as described in the next exercise).

- The interface and the user agent need no changes. Just copy them from the last exercise.
- In the translation agent we first need to state that we want the bdi agent to implement the translation interface via plans. This is done by declaring the implementation of the provided service to be the BDI Agent. Additionally remove the 'extends ITranslationService' part of the class definition. The interface is now only implemented indirectly via plans. Hence, also remove the translateEnglishGerman method from the agent completely.

```
@ProvidedServices(@ProvidedService(name="transser", type=ITranslationService.class,
implementation=@Implementation(BDIAgent.class)))
```

- Add a new plan that uses the dictionary to translate words. We want to execute this plan only is the word is contained in the dictionary. Thus, we use an inner class as plan and add a precondition method. Additionally, we add a plan body that takes as argument an object array representing the parameters of the service call. We need to fetch the first parameter, cast it to String and look it up in the dictionary.

```
@Plan(trigger=@Trigger(service=@ServiceTrigger(type=ITranslationService.class)))
public class TranslatePlan
{
    @PlanPrecondition
    public boolean checkPrecondition(Object[] params)
    {
        return wordtable.containsKey(params[0]);
    }
}
```

```

@PlanBody
public String body(Object[] params)
{
    String eword = (String)params[0];
    String gword = wordtable.get(eword);
    System.out.println("Translated with internal dictionary dictionary: "+eword+" - "+gword);
    return gword;
}
}

```

- We add a second plan that will allow us to translate words not contained in the internal dictionary. Instead we will use an online dictionary and look up the word. The result is retrieved as html page which needs to be parsed to extract the translation. The parsing code is presented below. Just copy the snippet and make it to a method plan of the agent using the @Plan annotation. It should have the same trigger as the other plan.

```

public String internetTranslate(Object[] params)
{
    String eword = (String)params[0];
    String ret = null;
    try
    {
        URL dict = new URL("http://wolfram.schneider.org/dict/dict.cgi?query="+eword);
        System.out.println("Following translations were found online at: "+dict);
        BufferedReader in = new BufferedReader(new InputStreamReader(dict.openStream()));
        String inline;
        while((inline = in.readLine())!=null)
        {
            if(inline.indexOf("<td>")!= -1 && inline.indexOf(eword)!= -1)
            {
                try
                {
                    int start = inline.indexOf("<td>")+4;
                    int end = inline.indexOf("</td", start);
                    String worda = inline.substring(start, end);
                    start = inline.indexOf("<td", start);
                    start = inline.indexOf(">", start);
                    end = inline.indexOf("</td", start);
                    String wordb = inline.substring(start, end== -1? inline.length()-1: end);
                    wordb = wordb.replaceAll("<b>", "");
                    wordb = wordb.replaceAll("</b>", "");
                    ret = worda;
                    System.out.println("Translated with internet dictionary: "+worda+" - "+wordb);
                }
            }
        }
    }
    catch(Exception e)
}

```

```

    {
        System.out.println(inline);
    }
}
in.close();
}
catch(Exception e)
{
    e.printStackTrace();
    throw new PlanFailureException(e.getMessage());
}
return ret;
}
}

```

Starting and testing the agents

Again, start both agents from the JCC. Now try out if internal as well as internet translations are displayed when entering translation requests in the gui.

F3 - Goal Delegation

Sometimes, mapping a service call to goal is more appropriate than a plan. This is the case if the BDI means-end reasoning should be used for executing the service call. Another advantage of a service to goal mapping is that it allows for goal delegation between different agents. This means we can just create a translation goal in the user agent and dispatch it. The goal will automatically be forwarded (as service call) to the translation agent which will reify the call to a goal and try to achieve it.

- Copy the unchanged ITranslationService interface.
- Create a new class called TranslationGoal representing the shared goal between both agents. For this reason we do not want to define it as inner class of one of the agents. Use the @Goal annotation to make it become a goal. Moreover, add two fields of type String: one called gword and one called eword. Make eword become a goal parameter (@GoalParameter) and gword become the goal result (@GoalResult). Add a constructor taking eword as parameter and generate getter/setter methods for both fields.

```

@Goal
public class TranslationGoal
{
    @GoalResult
    protected String gword;
}

```

```

@GoalParameter
protected String eword;

public TranslationGoal(String eword)
{
    this.eword = eword;
}
...
}

```

- In the user agent we need to change that a translation goal is used and that it has delegation capabilities. The idea is to allow for defining a plan that is represented by a required service. Such a mapping is defined using the `@ServicePlan` annotation. It refers to the name of a previously defined required service (here 'transser').

```

@RequiredServices(@RequiredService(name="transser", type=ITranslationService.class,
binding=@Binding(scope=RequiredServiceInfo.SCOPE_PLATFORM))
@Goals(@Goal(clazz=TranslationGoal.class))
@Plans(@Plan(trigger=@Trigger(goals=TranslationGoal.class), body=@Body(service=@ServicePlan

```

- The code in the action listener of the translate button has to be changed to create a translation goal instead of a service call. As we need to dispatch a goal on the agent thread (and not on the Swing thread which is active when the button is pressed) first a thread switch has to be applied. This is done using a component step which is executed on the agent. Then just create and dispatch the goal and use `get()` to wait for the result of the future. Afterwards set the result in the textfield on the swing thread. Also catch exceptions and display errors in case they occur.

```

agent.scheduleStep(new IComponentStep<Void>()
{
    public IFuture<Void> execute(IInternalAccess ia)
    {
        try
        {
            final String gword = (String)agent.dispatchTopLevelGoal(new TranslationGoal(tfe.getText())
                // set word in textfield on swing thread
            )
        }
        catch(Exception e)
        {
            // set the exception message in textfield on swing thread
        }
    }
});

```


- Our translation agent is very simple in this lecture. We change the annotation part to publish the translation goal as a service, i.e. when the service is called a new goal is created and after processing the result will be set as result of the call. As in this case our interface only has one service method it is sufficient to state the service interface. Otherwise the method would also have to be defined.

```

@Agent
@Service
@Goals(@Goal(clazz=TranslationGoal.class, publish=@Publish(type=ITranslationService.class)))
public class TranslationBDI
{
    ...
}

```

- Keep the two field definitions and the agent init method.
- Delete both plans and instead add a new simple method plan that reacts on the translation goal. It just looks up the word and return the translation.

```

@Plan(trigger=@Trigger(goals=TranslationGoal.class))
public String translatePlan(String eword)
{
    return wordtable.get(eword);
}

```

Starting and testing the agents

Start both agents from the JCC and verify that translation requests get executed.

External Processes

One prominent application for agents is wrapping legacy systems and “agentifying” them. Hence, it is an important point how separate processes can interact with Jadex agents as these applications often use other means of communications such as sockets or RMI.

A Jadex agent executes behavior sequentially and does not allow any parallel access to its internal structures due to integrity constraints.

For this reason it is disallowed and discouraged to block the active plan thread e.g. by opening sockets and waiting for connections or simply by calling *Thread.sleep()*. This can cause the whole agent to hang because the agent waits for the completion of the current plan step. When external processes need to interact directly with the agent, they have to use methods from the so called

jadex.runtime.IExternalAccess interface, which offers the most common agent methods.

Exercise G1 - Socket Communication

We create a simple translation with a plan that sets up a server socket which listens for translation requests. Whenever a new request is issued (e.g. from a browser) a new goal containing the client connection is created and dispatched. The translation plan handles this translation goal and sends back some HTML content including some text and the translated word.

- Create a new agent class called TranslationBDI.
- Add again the fields for the agent API and the word table.
- Add another field called server that stores the ServerSocket.
- Create a goal as inner class called Translate that has a field called client of type Socket. Also add a constructor with a parameter of type Socket and add getter/setter methods for it.

```
@Goal
public class Translate
{
    protected Socket client;

    public Translate(Socket client)
    {
        this.client = client;
    }
    ...
}
```

- Add an agent init method and first put the code for creating and initializing the word table. Afterwards we want to setup the server socket and listen for incoming client requests. To achieve this we create a Runnable that sets up the connection:

```
Runnable run = new Runnable()
{
    public void run()
    {
        try
        {
            server = new ServerSocket(port);
        }
        catch(IOException e)
        {
```

```

        throw new RuntimeException(e.getMessage());
    }

```

- If the socket could be opened we start waiting in an endless loop in a blocking fashion for incoming calls. Whenever a request is received we schedule a step on the agent and dispatch a translation goal with the new client socket.

```

while(true)
{
    final Socket client = server.accept();
    agent.scheduleStep(new IComponentStep<Void>()
    {
        @Classname("translate")
        public IFuture<Void> execute(IInternalAccess ia)
        {
            agent.dispatchTopLevelGoal(new Translate(client));
            return IFuture.DONE;
        }
    });
}

```

You need to put the loop code above in a try catch construct to shut down the server when a component termination exception occurs (i.e. the agent was terminated).

```

if(server!=null)
{
    try
    {
        server.close();
    }
    catch(Exception e)
    {
    }
}

```

- At the end of the agent init method start a new thread with the runnable using:

```

Thread t = new Thread(run);
t.start();

```

- What is still missing is the plan that reacts on the translation goals. We define it as a method with a corresponding trigger. In the body we process the HTTP request by parsing and reading the word from the socket and

write back the translated word as HTTP response.

```
@Plan(trigger=@Trigger(goals=Translate.class))
public void translate(Translate trans)
{
    Socket client = trans.getClient();

    try
    {
        BufferedReader in = new BufferedReader(new InputStreamReader(client.getInputStream()));
        String request = in.readLine();
        int slash = request.indexOf("/");
        int space = request.indexOf(" ", slash);
        String eword = request.substring(slash+1, space);
        String gword = wordtable.get(eword);
        System.out.println(request);
        PrintStream out = new PrintStream(client.getOutputStream());
        out.print("HTTP/1.0 200 OK\r\n");
        out.print("Content-type: text/html\r\n");
        out.println("\r\n");
        out.println("<html><head><title>TranslationM1 - "+eword+"</title></head><body>");
        out.println("<p>Translated from english to german: "+eword+" = "+gword+".");
        out.println("</p></body></html>");
        out.flush();
        client.close();
    }
    catch(IOException e)
    {
        throw new RuntimeException(e.getMessage());
    }
}
```

Starting and testing the agent

Start the agent and open a browser to issue translation request. This can be done by entering the server url and appending the word to translate, e.g. <http://localhost:9099/dog>. The result should be printed out in the returned web page.

Conclusion

We hope you enjoyed working through the tutorial and now are equipped at least with a basic understanding of the Jadex BDI reasoning engine. Nevertheless,

this tutorial does not cover all important aspects about agent programming in Jadex.

Most importantly the following topics have not been discussed:

Goal Deliberation

This tutorial only mentioned the different goal types available in Jadex (perform, achieve, query and maintain). It does not cover aspects of goal deliberation, i.e. how a conflict free pursuit of goals can be ensured.

Jadex offers the built-in *Easy Deliberation* strategy for this purpose. The strategy allows to constrain the *cardinality* of active goals. Additionally, it is possible to define *inhibition links* between goals that allow to establish an ordering of goals. Inhibited goals are suspended and can be reactivated when the reason for their inhibition has vanished, e.g. another goal has finished processing.

Plan Deliberation

If more than one plan is applicable for a given goal or event the Jadex interpreter has to decide which plan actually will be given a chance to handle the goal resp. event. This decision process called plan deliberation can be customized with *meta-level reasoning*. This means that building the applicable plan list can be completely customized on the user level by using the `@GoalAPLBuild` annotation in a goal.

Please have a look at the puzzle example (SokratesBDI) to see how it can be used.

If you have any comments or improvement resp. extension proposals don't hesitate to contact us.

1 Chapter 1 - Introduction

The Jadex Processes project aims at providing modelling and execution facilities for workflows. Main focus is on graphical forms of process representation (e.g. the Business Process Modelling Notation - BPMN) and direct execution of modelled processes (i.e. without prior code generation).

Currently, with BPMN and GPMN two workflow types are supported. Both workflow types are designed as active components and can thus be executed on the Jadex Active Components middleware. The BPMN engine is realized as an interpreter for (extended) BPMN diagrams that are produced by the Jadex BPMN editor. In order to be executable the diagrams need to be annotated with Java expressions describing the semantics of the main elements like activities

or branching conditions. GPMN workflows are goal-oriented and allow the definition of processes at a higher abstraction level.

This tutorial provides step-by-step instructions for learning how to use the Jadex process infrastructure and BPMN modelling features. You will learn how to install and use the process infrastructure (i.e. the Jadex platform) for executing modelled processes and how to install and use the extended eclipse BPMN modeller tool. In particular, the following topics are covered in the upcoming chapters:

- Installation describes the steps necessary to install and run the Jadex process engine and modelling tools.
- Basic Processes illustrates how to use the editor and platform to create and execute simple processes.
- Data and Parameters covers how data can be accessed from and passed between process activities.
- Events and Messages shows how to react to and issue events as well as send and receive messages.
- Custom Functionality describes the various ways to extend Jadex BPMN with custom functionality.

Chapter 2 - Installation

In this chapter, you will learn how to start the Jadex BPMN editor. You will also find some instructions on setting up a proper eclipse working environment for executing Jadex BPMN processes.

Prerequisites

- Download and install a recent Java environment from <http://www.oracle.com/technetwork/java/javase/downloads/index.html> (if not already present).
- Download and install a suitable eclipse distribution from <http://www.eclipse.org/downloads/> (if not already present).
- Download the latest Jadex build .zip from <http://www.activecomponents.org/download/> and unpack it to a place of your choice (only necessary if you don't want to use maven).

Exercise A1 - Eclipse Project Setup

In this lesson you will set up an initial eclipse environment that will be used in the following lessons. Please follow the instructions carefully and compare your setup to the screenshots to verify that everything went fine.

We will first describe how a setup is done based on downloaded Jadex, afterwards using maven. Thus, you can choose one of both options.

Alternative 1: Using Downloaded Jadex

Start eclipse. Start the 'New Java Project' wizard, set the project name to e.g. 'bpmntutorial' and click 'Finish' - the project will be created.

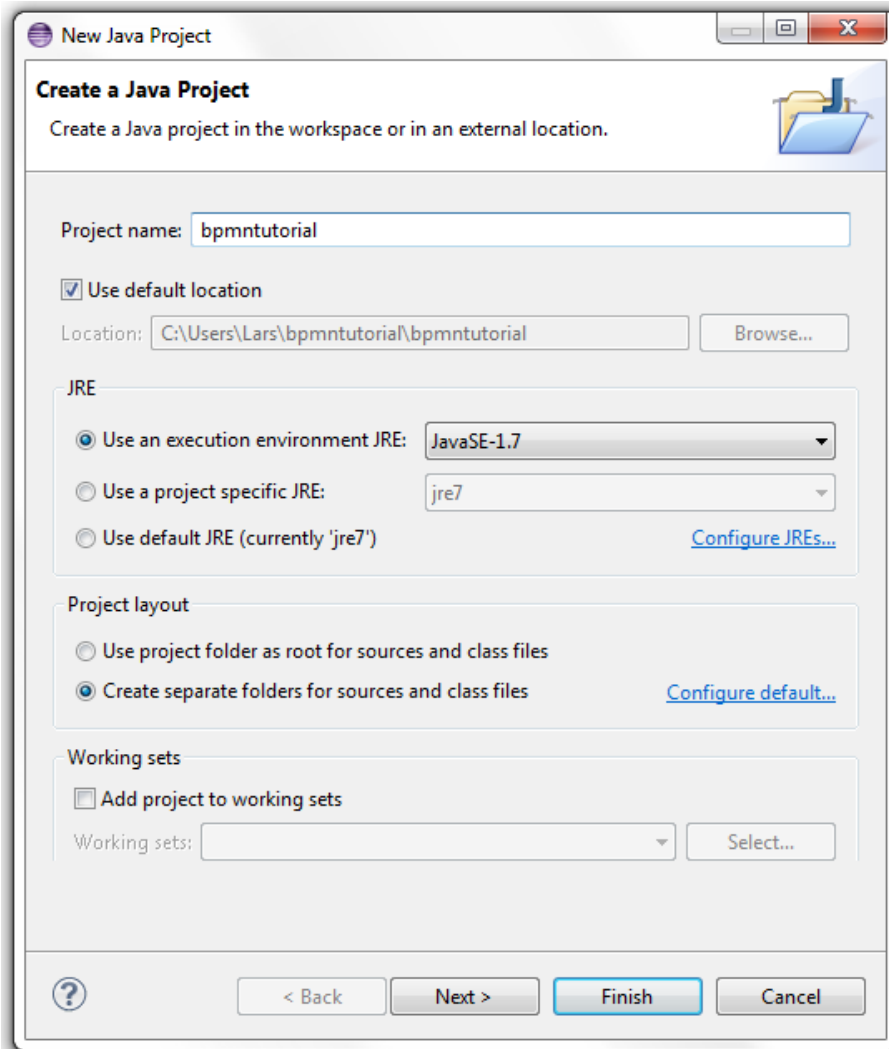


Figure 11: 02 Installation/1.png

Now we need to add the Jadex jars to the build path of our project. For this

purpose right click on the project folder and select 'properties'. Switch to the 'build path' options and click 'add external jars'. Navigate to the Jadex directory and add all jars that are in the 'lib' folder as shown below.

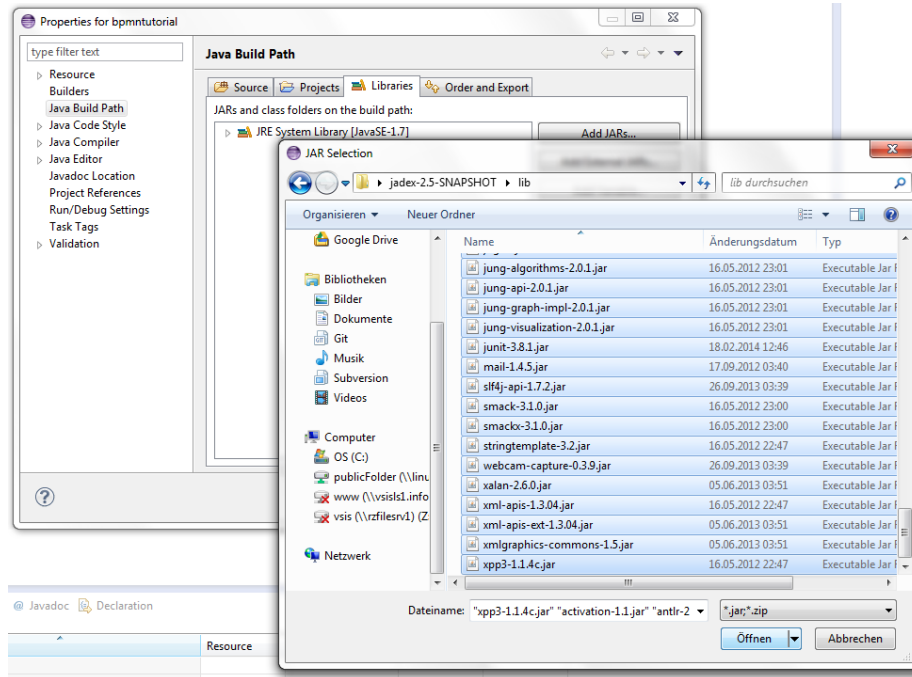


Figure 12: 02 Installation@2.png

Alternative 2: Using Jadex via Maven

Create a new maven project by right-clicking in the package explorer and selecting 'new' -> 'other'. Choose 'Maven Project' and click 'next'. In the 'New Maven Project' dialog activate the checkbox 'create a simple project' and click 'next'.

Afterwards enter a group and artifact id, e.g. 'jadex' and 'bpmntutorial' and click 'Finish'.

Now we have to add a dependency to Jadex in the 'pom.xml'. If you want to use the latest Jadex nightly builds, it is necessary to add the Jadex repository to the pom.xml. Releases can be directly obtained from the Maven central repository. Below, it is shown what has to be added to the pom.xml for using the 2.5 nightly build.

```
<dependencies>
  <dependency>
```

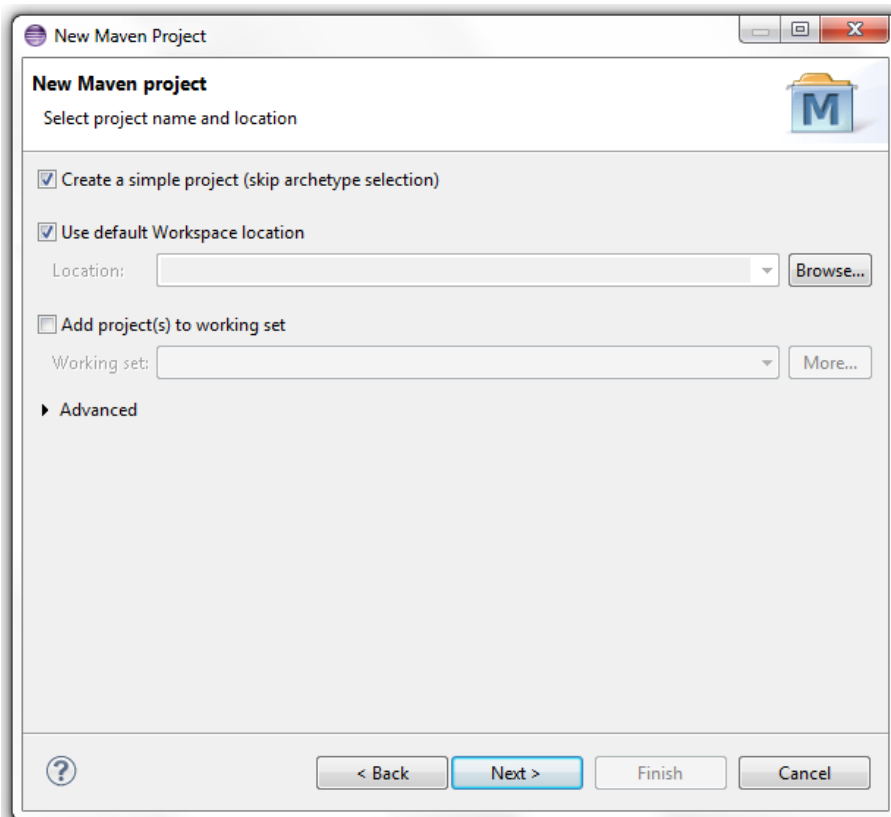



Figure 13: 02 Installation@3.png

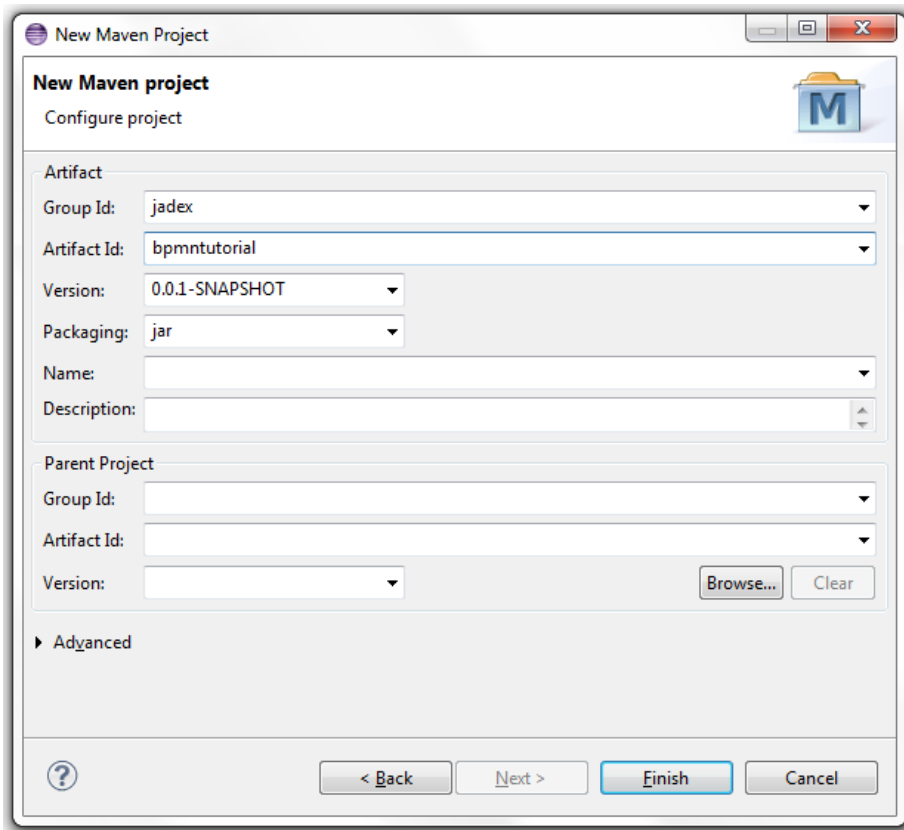


Figure 14: 02 Installation@4.png

```

    <groupId>net.sourceforge.jadex</groupId>
    <artifactId>jadex-distribution-standard</artifactId>
    <version>2.5-SNAPSHOT</version>
  </dependency>
</dependencies>

<repositories>
  <repository>
    <id>jadex-snapshots</id>
    <url>http://www0.activecomponents.org/nexus/content/repositories/snapshots</url>
  </repository>
</repositories>

```

Please note, that you can also use other Jadex servers (www0, www1, www2, www3) or the generic www address. In the latter case a server is chosen randomly, which might not be helpful because the working builds may differ.

Exercise A2 - Starting the BPMN Editor

Since version 2.5 Jadex comes with a standalone BPMN editor that is directly bundled in the distributed. It can be either start via the command line by navigating to the Jadex folder and executing 'bpmn_editor.bat/sh' depending on the operating type you are using.

As we develop BPMN process via eclipse in this tutorial we will also add a new runtime configuration for starting the editor. For this purpose open the 'run configurations...' dialog and enter 'BPMN Editor' as configuration name. If no project is selected, choose our new 'bpmntutorial' project. As main class enter 'jadex.bpmn.editor.BpmnEditor' or search it via the corresponding button.

After starting the run configuration, the BPMN editor gui should pop up. It should look similar to below.

Exercise A3 - Running Example Processes

In this lesson we will create a launch configuration to start the Jadex platform. To see that everything works, we will execute some example processes that are distributed with the Jadex package.

Eclipse Launch Configuration

Enter the 'run configurations' dialog again and create a new configuration for a Java application. You can name it e.g. 'Jadex Platform' and enter 'jadex.base.Starter' as main class. Instead you can of course also use the search dialog as shown below. Hit 'Run' to start the Jadex platform.

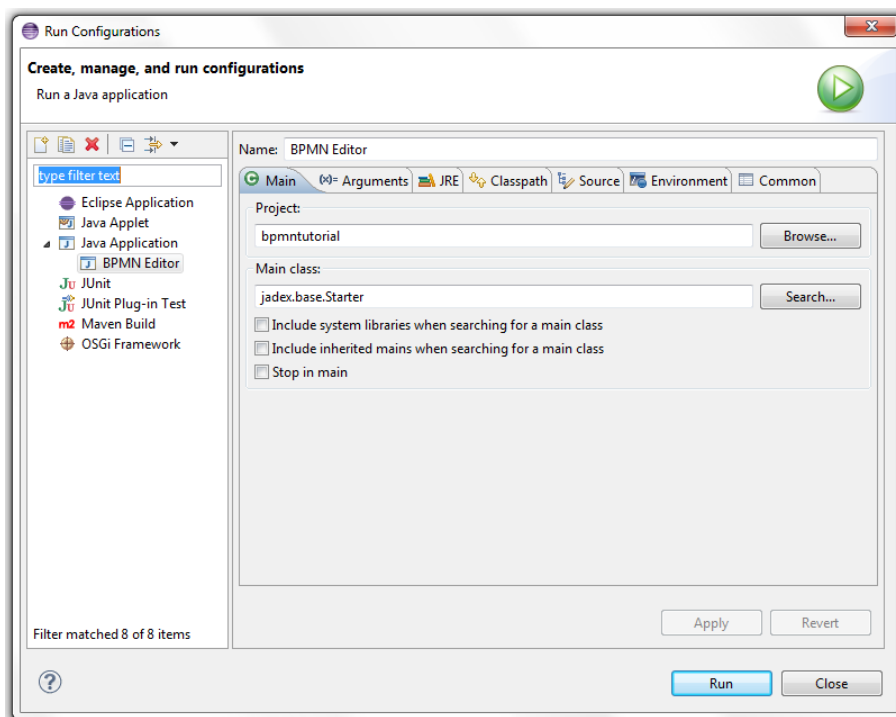


Figure 15: 02 Installation@5.png

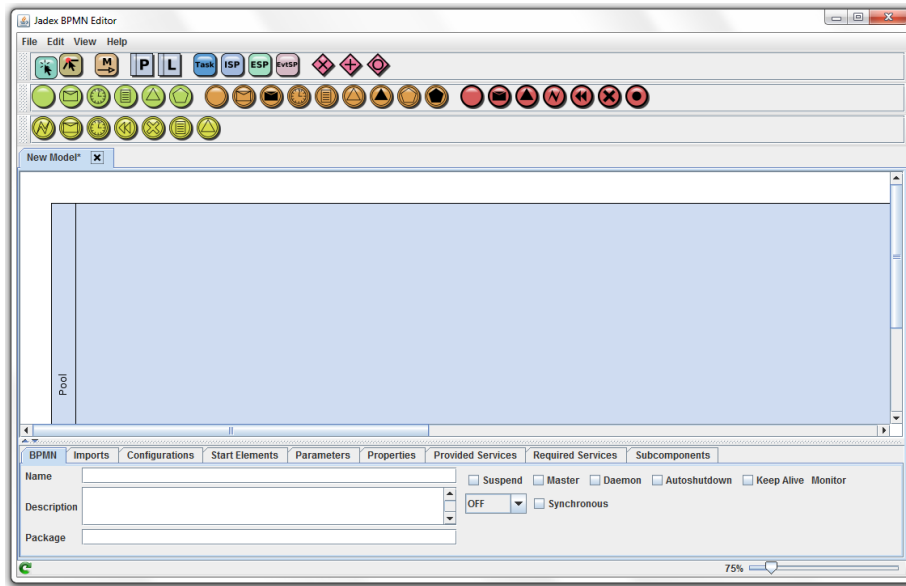


Figure 16: 02 Installation@6.png

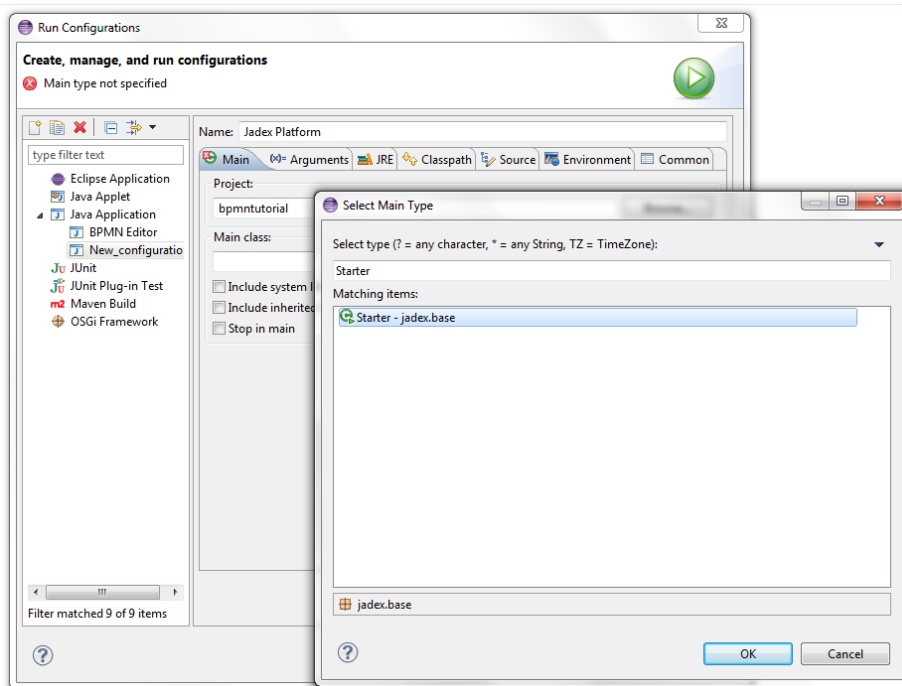


Figure 17: 02 Installation@7.png

Selecting and Starting a Process

If you managed to successfully start the Jadex platform, the Jadex control center (JCC) window will appear (see below). The JCC is a management and debugging interface for the Jadex platform and the components that run on it.

To execute a process you need to add the corresponding resource path to the JCC project. Right-click in the upper left area (called the model explorer, as it is used to browse for models of e.g. processes) and choose ‘Add Path’.

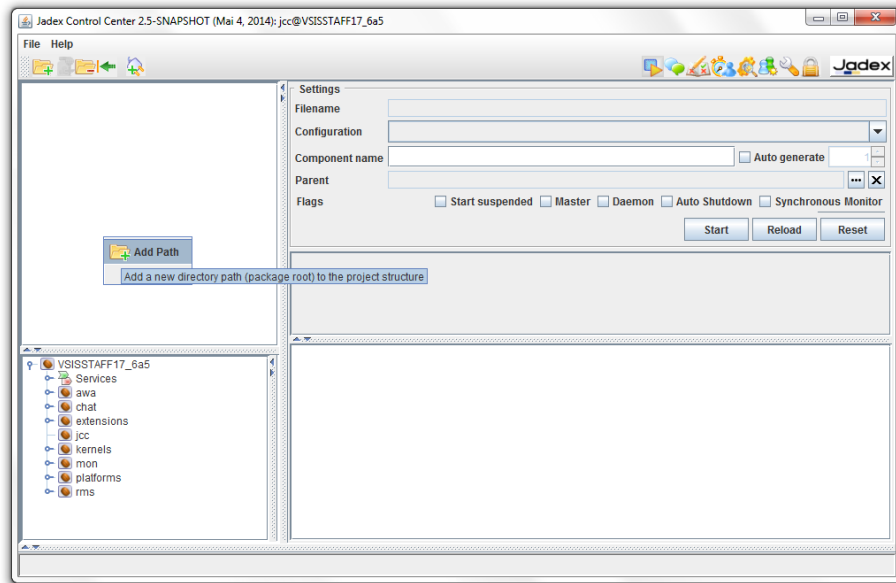


Figure 18: 02 Installation@8.png

A file requester appears that should initially present the directory, where you unpacked the Jadex distribution. Open the *lib* directory and select the file *jadex-applications-bpmn-xyz.jar* (with xyz as placeholder for the current version). You can now unfold the contents of the jar file and browse to the helloworld example. After you selected the *HelloWorld.bpmn2* in the tree, you can start the process by clicking ‘Start’.

The process will be executed, thereby printing some messages to the (eclipse) console.

Saving the JCC Project

As you probably do not want to add the jar file again, each time you start the Jadex platform, you should save the current settings of the JCC. From the ‘File’ menu choose ‘Save Settings’.

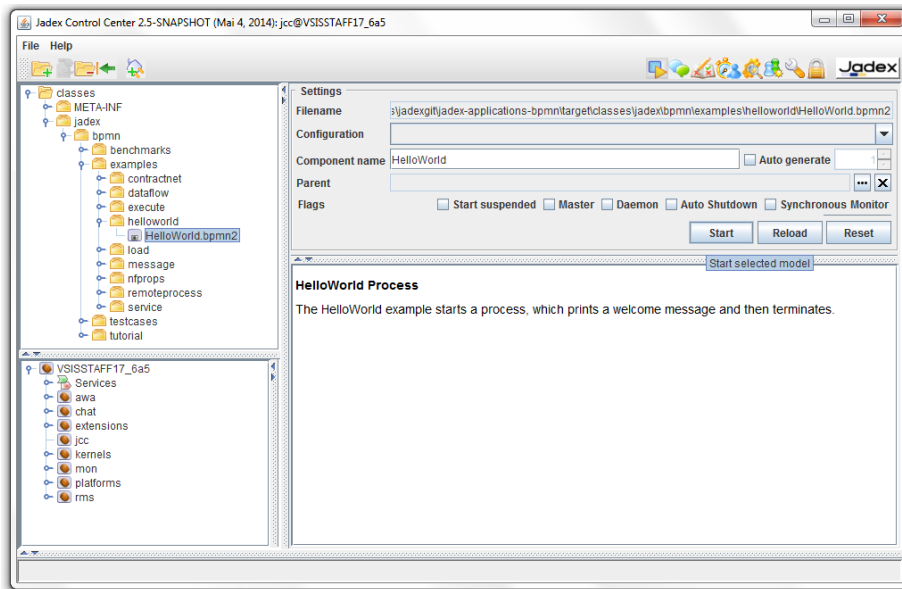


Figure 19: 02 Installation@9.png

Using the Visual BPMN Debugger

Jadex additionally provides a visual debugger for BPMN processes. We will show here how the tool can be used to control the execution of the helloworld process. As a first step we start the helloworld process again, but this time as suspended. For this purpose activate the ‘Start suspended’ checkbox in the ‘Settings-Flags’ section of the BPMN process model (on the right hand side of the JCC). After having started the process it will be displayed as suspended, which is indicated by a ‘zzz’ on the component icon.

Now we are ready to use the BPMN debugger. First we have to switch to the debugger view by activating the corresponding JCC plugin via the toolbar

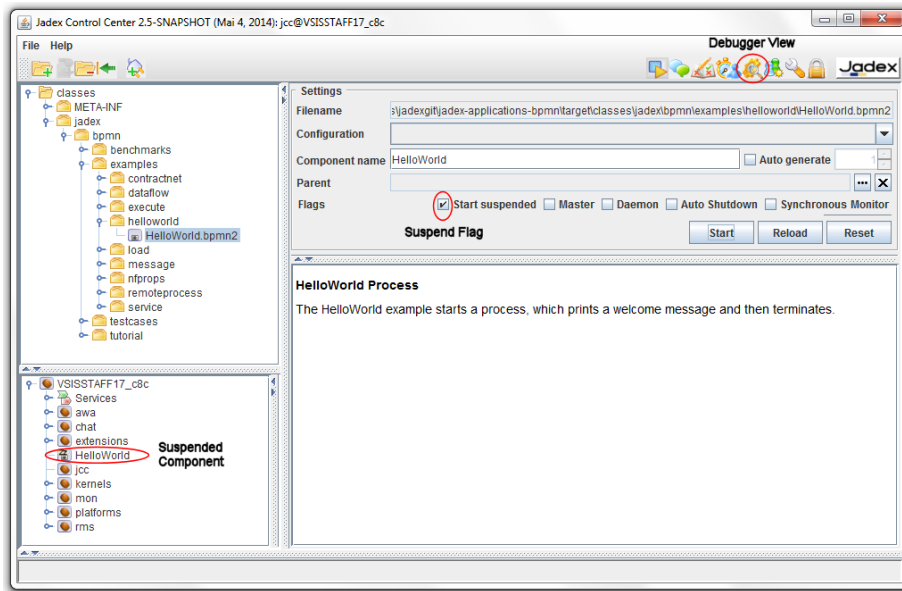
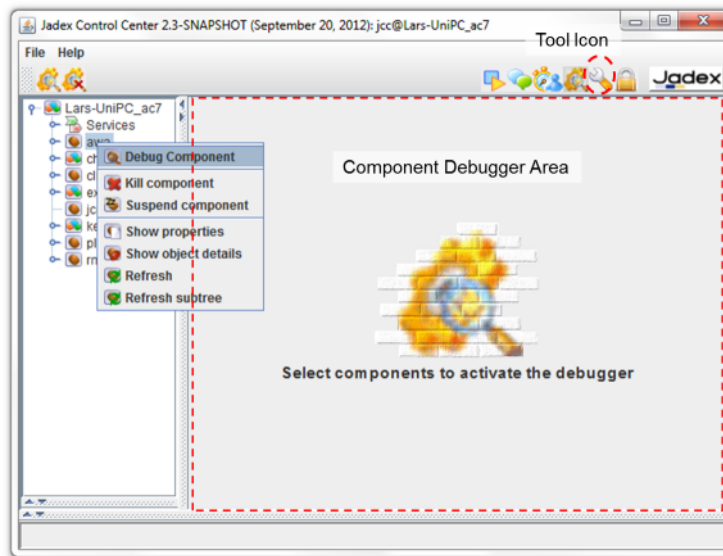


Figure 20: 02 Installation@10.png



button . In the debugger view you can see the running components on the left and the debugging panel on the right. This panel depends on the concrete type of component, i.e. for BPMN the debugger panel looks different than for BDI or micro agent components. Double-click the helloworld component in the left side of the window to activate debugging for that component. On the right hand side, the debugger

panel consists of three different areas. On the top-left the breakpoint panel shows the available breakpoints in the process. On the top-right the bpmn diagram is shown and below the currently existing process threads including their state and variables are displayed.

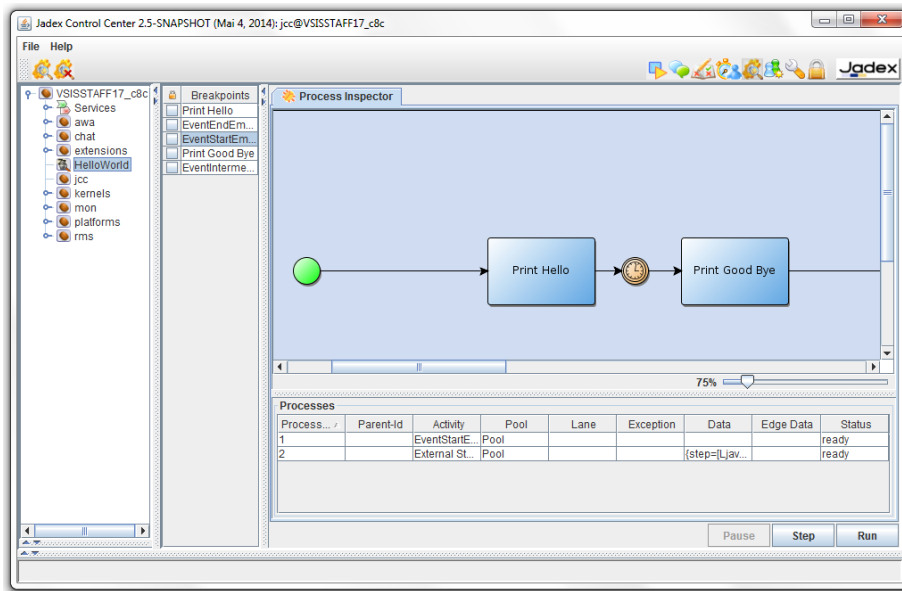



Figure 21: 02 Installation@11.png

To execute a step of the process two options are available. First, you can directly double-click activated elements in the visual process diagram. Such activated elements are displayed in green (in the figure above the start event is the only activated element). As an alternative you can press the 'Step' button at the bottom, which will execute a step of one process thread. If you want to determine which process thread is selected you have to select it in the 'processes table' above.

You can also debug multiple processes at once. Try starting multiple instances of the process. Each of the created instances can be independently controlled in the debugger tool. A double-click on a process instance in the debugger opens/closes the debugger view for this process. An active debugger view is represented by a symbolic magnifying glass on the process icon. A single-click on such an icon will switch to the debugger view of this process instance.

The debugger also allows you to set breakpoints in your process. The available breakpoints are shown in the left area of the debugger view. The breakpoints correspond to the elements in the process diagram (e.g. tasks but also other elements such as events or gateways). The checkbox left besides each element allows to activate/deactivate the breakpoint for this element. When a running

process hits a breakpoint, the process is automatically suspended. E.g., when the next activity of a process has an active breakpoint, the process is stopped before the activity is executed. To resume a suspended process, you can also use the 'Run' button in the debugger view. The process is then executed until it finishes or hits the next breakpoint. Try setting a breakpoint and clicking 'Run'. Observe how the process executes some activities before it stops at the breakpoint. The visual debugger shows elements with an active breakpoint using

a red stop sign .

Execute Example Processes

Execute some other example processes, e.g., from the 'execute' or 'message' folder. Open the diagrams of these processes in eclipse to get an initial impression of the features of Jadex BPMN modeling and execution. These features will be introduced in the lessons of the subsequent chapters.

Chapter 3 - Basic Processes

In this chapter you will learn how to model your own processes. This chapter covers basic issues such as activities and control flow. It is assumed that you have some initial understanding of BPMN and its graphical elements. If you think that you need more background information on BPMN, please refer to documentation available elsewhere. Below is a short list of suggestions, but you will easily find other sources of information on the Web.

- Official BPMN homepage (www.bpmn.org)
 - Introduction to BPMN (article, PDF)
 - BPMN Tutorial (slides, PDF)
 - BPMN Specification (download site)
- BPMN Corner at Uni Potsdam
 - BPMN 2.0 Poster (pdf)
 - BPMN 1.2 Poster (pdf)
- Wikipedia entry for BPMN

Exercise B1 - Creating a First Process

In this lesson, you will create and execute a first process. First, start the Jadex BPMN editor using the run configuration created in the last section. The editor will automatically create a new unnamed BPMN model. To save the model under a custom name e.g. 'B1_simple' use 'Save as' in the 'File' menu. The destination folder should be located in the Java 'src' folder of your eclipse 'bpmntutorial' project, i.e. you need to navigate to the eclipse workspace to find it. For easy

reference the tutorial files are named according to the corresponding lesson, but you are free to choose a different name.

Jadex Project Setup

The just created process can already be executed without further editing. As you have created the BPMN process using the Jadex editor, first you will have to refresh the 'bpmntutorial' project using 'refresh' from the popup menu or by selecting the project and pressing 'F5'. Now, start the Jadex platform using your existing launch configuration (see Exercise A3). The JCC window will appear, probably showing the example project that you created in Lesson A3. Right-click in the model explorer and choose 'Add Path'. Browse to your eclipse workspace and select the 'bin' or 'classes' folder from the eclipse project that you created in the beginning of this lesson. When you unfold the contents, you should find the package(s) that you created and the process contained within.

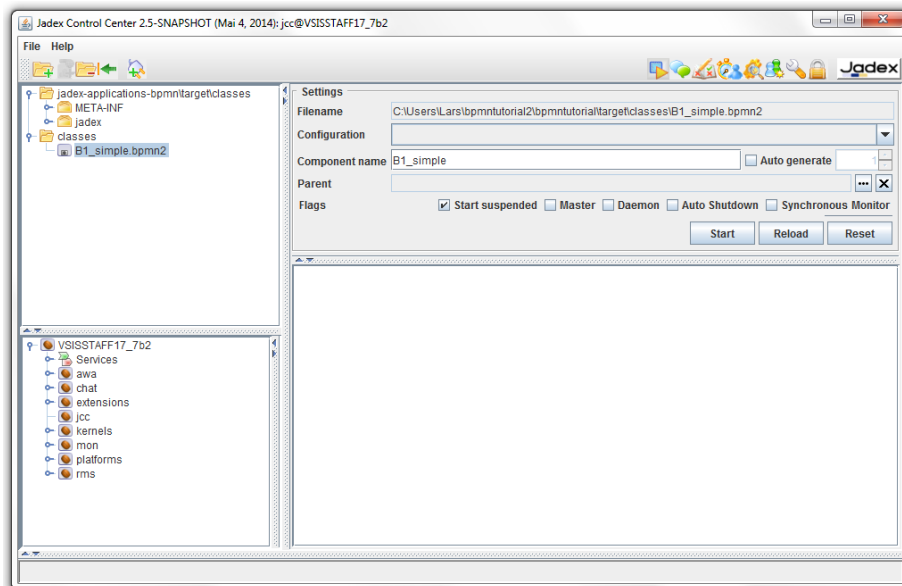


Figure 22: 03 Basic Processes@1.png

Select your process in the tree and click the 'Start' button. You might think that nothing happens, but actually the process is instantiated and executed. The reason that you cannot observe anything is that the process does not contain activities and therefore immediately terminates without producing any output. In the next sections, you will change this and actually see some process output. Before going back to the diagram editor, you should save the JCC project ('Save Settings' in 'File' menu).

Process Properties

The lower area in the BPMN editor allows you to see and edit the details of the currently selected diagram element (e.g. a task in the process). In addition to properties of the visible diagram elements, the process as a whole also has properties that can be edited. You can access the process properties by selecting a pool or the empty background of the diagram.

You can see that a process has the following properties:

- **Name:** The process name - should be the same as the filename.
- **Description:** A text with documentation about the process.
- **Package:** The package in which the process is contained (like the package of a Java class file).
- **Imports:** Import classes and packages that you want to use inside the process.
- **Configurations:** A configuration allows for starting a process with a specific set of settings.
- **Start Elements:** For each configuration the elements that should be started can be selected (e.g. if a pool should be active).
- **Parameters:** Parameters can be used to hold global data. Additionally, parameters can be made to arguments and results as well.
- **Provided Services:** Services that are offered by the process.
- **Required Services:** Services that are needed by the process.
- **Subcomponents:** The subcomponent model definitions.

The description can contain arbitrary text and HTML tags. The description is, e.g., displayed in the JCC, when selecting the process. Enter a description for your process, restart the Jadex platform (or just reload the process) and see how the description is displayed. Please note, that you always have to refresh the eclipse project, otherwise the changes will not be recognized.

You can also enter a package for your process. The package should always correspond to the directory structure, where your process is located. Otherwise you will run into problems later, when you try to use your Java classes in the process. Imports, parameters, arguments and results will be covered in later lessons and can stay empty for now.

Printing to the Console


Finally, you probably want to see that the process is really executed, when you click the start button. This can be achieved by changing the task in the process to print some text to the console. Open the diagram in the BPMN editor (if not already open) and add a 'Task' element. Selecting the task will show its properties in three tabs (Task, Properties, Parameters) in the lower area:

- **Task:** In the task view you can enter the Java class that should be executed

when the task is invoked. Below the classname usage information of the task is presented including a description and the used parameters.

- **Properties:** Properties are settings that are directly related to the BPMN element, e.g. a time duration for a timer event. Thus, all BPMN elements of the same type expose the same properties.
- **Parameters:** Input and output parameters for the activity.

Jadex provides some ready-to-use task implementations, which can be chosen from the drop-down list. The available contents of the list is found by scanning the class path of the editor. To include the standard Jadex task implementation we need to add the Jadex jars to the classpath of the editor.

To do this, go to the 'File' menu and open the 'Settings' dialog. Switch to the 'Class Path' tab and choose 'Add Project'. Here, choose the Jadex distribution directory. The editor will automatically scan the folder structure for jars and add them to the classpath. You should see the Jadex jars in the dialog afterwards. After exiting the settings dialog you will see the editor refreshing its class cache used for the autocompletion. You can also manually start rescanning by pressing the refresh button  at the lower left of the editor panel.

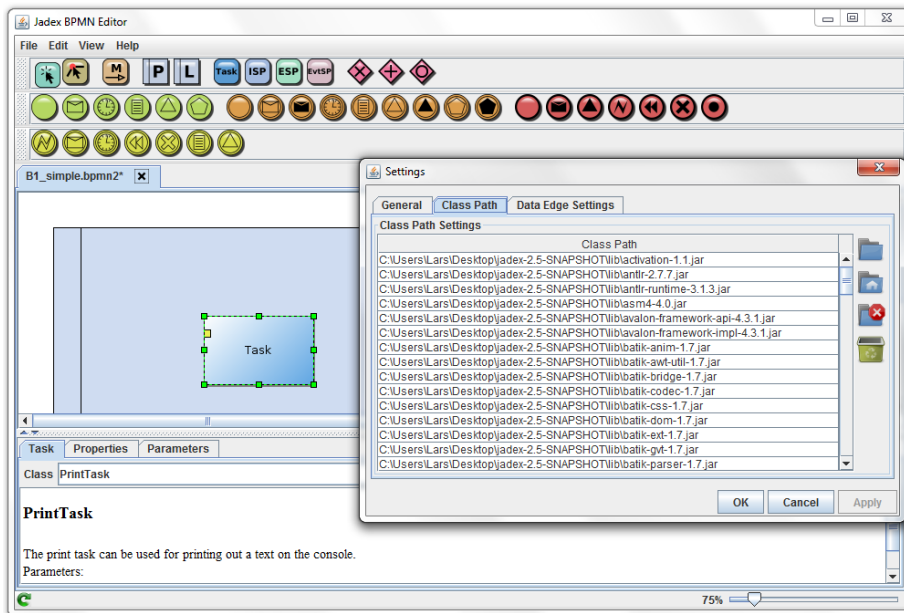



Figure 23: 03 Basic Processes@3.png

You can also implement your own tasks, by writing corresponding Java classes. The available task implementations as well as how to produce your own tasks will be covered later. For this lesson, just select the 'jadex.bpmn.runtime.task.PrintTask', which allows printing some text to the

console.

You will see that some description text about the task is displayed. Among other things, the description tells you that this task implementation expects an input parameter 'text' of type String. To set the text that should be printed we first have to switch to the 'Parameters' tab and afterwards include the default

parameters of the selected task class. This is done by clicking this button . Enter "The task has been executed" in the 'Initial Value' column of the parameter table. The value is entered as a Java expression, which is why you have to enclose your text in quotes. To make the process better readable, also draw a start and end event and connect them to the task. It should look like the diagram below.

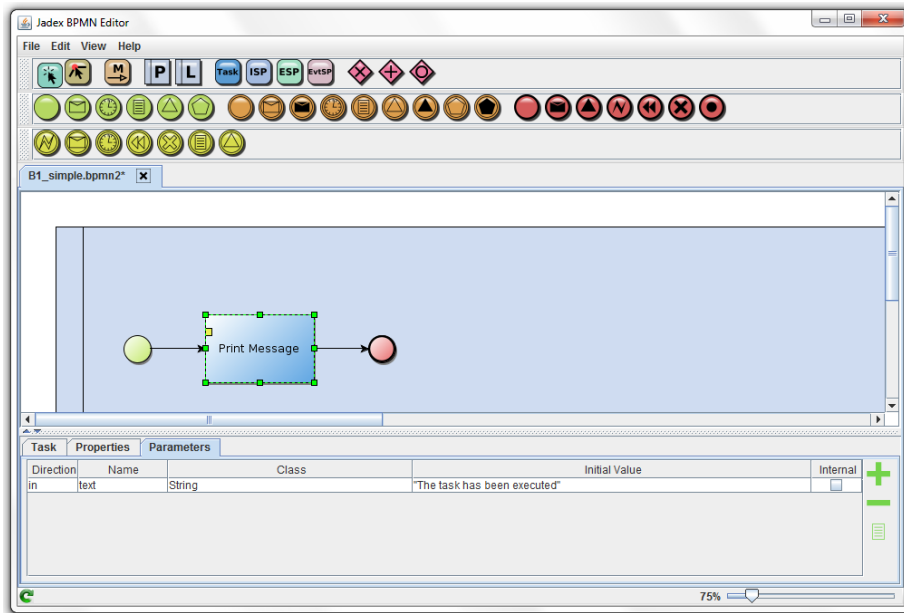


Figure 24: 03 Basic Processes@5.png

Save the model, refresh Eclipse and restart the Jadex platform. Observe that your text gets printed to the eclipse console every time that you start your process.

Exercise B2 - Sequence of Tasks

In this lesson you will learn how to execute tasks in sequence, i.e. one after another.

Creating Tasks and Flow Connectors

Create a new BPMN diagram with a name of your choice, e.g. 'B2_Sequence'. Create three tasks connected by flow connection arrows. Flow connection arrows have a continuous line and a solid head. There are different options to create tasks and flow connections. You can select the task or flow connection element in the palette above the diagram and add the task at the required place or draw a connection between tasks. Another way is using the input/output connectors that appear when clicking in the middle of existing elements as shown below. Just drag a connector to an empty place and select the element to be created.

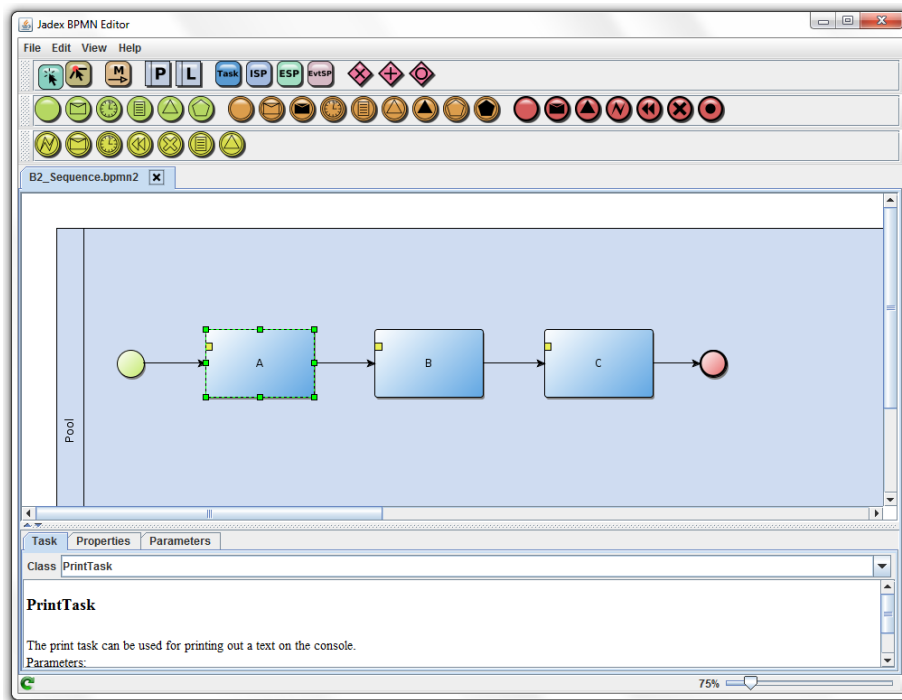


Figure 25: 03 Basic Processes@6.png

To easily observe how the different tasks are executed, change the task implementations to the `PrintTask` and enter some message in each of the text parameters (see last lesson for details). Execute your process using the JCC and observe the console output. You might have to refresh the model tree (e.g. by right-clicking of the folder and selecting 'Refresh' or by just pressing [F5]) for the new process to show up.

Exercise B3 - Parallel Activities

This lesson introduces forms of parallelism in processes. Each process can execute any number of so called process threads, which are independent control flows inside the process. Such control flows can appear and vanish during the execution of a process, e.g. at split or join nodes.

Creating the Process

Create a new process called, e.g. 'B3_Parallel'. Add tasks, connectors and gateways as shown below.

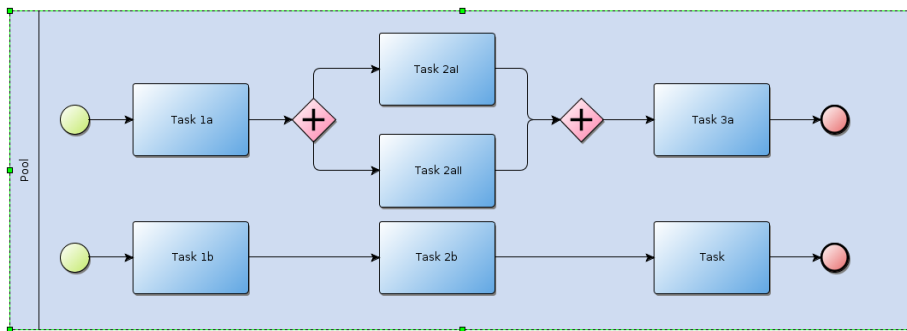


Figure 26: 03 Basic Processes@7.png

In this process, parallelism is introduced at two places. First, the tasks 'Task 1a' and 'Task 1b' are parallel to each other. A process always starts execution at start events and node(s) without incoming control flow connections. As the process has two start events it has two starting points.

Second, the tasks 'Task 2a I' and 'Task 2a II' are parallel tasks, because of the explicit parallel gateway 'Gateway 1' before the nodes. Parallel gateways are sometimes also called AND gateways. The two control flows of 'Task 2a I' and 'Task 2a II' are merged together by the second AND gateway 'Gateway 2'. The two forms of AND gateways are also called 'split' and 'join' gateways. They are represented by the same symbol, but can be distinguished, because a split has only one *incoming* edge while a join has only one *outgoing* edge. The semantics of the join is that 'Task 3a' may only be executed after both 'Task 2a I' and 'Task 2a II' have been completed.

Observing the Execution

To see some results during the execution, you can make use of the PrintTask as in the previous lessons. You can also observe the execution of the process in the debugger.

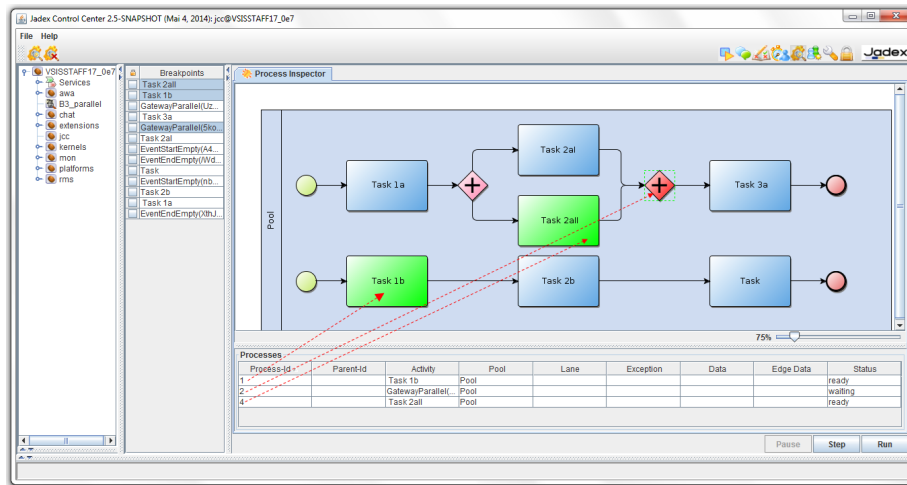


Figure 27: 03 Basic Processes@8.png

In the example screenshot you can see, that there are currently three control flows in the process with numbers 1, 2 and 4. Process thread 1 has proceeded to 'Task 1b', thread 2 is at the parallel join gateway and thread 4 is at 'Task 2aII'. Please note that thread 1 and 4 are ready, i.e. can execute the next step while thread 2 is waiting for the second thread at the gateway and thus cannot immediately proceed. The different states (waiting vs. ready) are also signalled by the color of the corresponding elements in the diagram (red vs. green). Play around with the process in the debugger. Also try out using breakpoints. You will notice, that the process is suspended whenever one of the control flows hits a breakpoint. Because the process is suspended as a whole this means that also the other control flows will stop executing when a breakpoint is hit.

Exercise B4 - Conditional Branch

In this lesson, an XOR gateway is used to split the control flow into one of two branches. Therefore it is shown how to add conditions to flow connectors. Create a new BPMN diagram called, e.g., 'B4_Choice'. Draw BPMN elements as shown in the picture.

The process simulates the toss of a coin. Either the 'Head' activity should be executed or the 'Tail' activity. Note the use of the XOR gateway to distinguish between the two cases. The expression `Math.random() > 0.5` is Java code. To enter the condition expression, first click on the upper sequence flow. This will activate the properties tab in the lower area of the editor. In this view the expression can be placed in the 'Condition' input field.

Math.random() is a function that generates a random value between 0 and 1. The

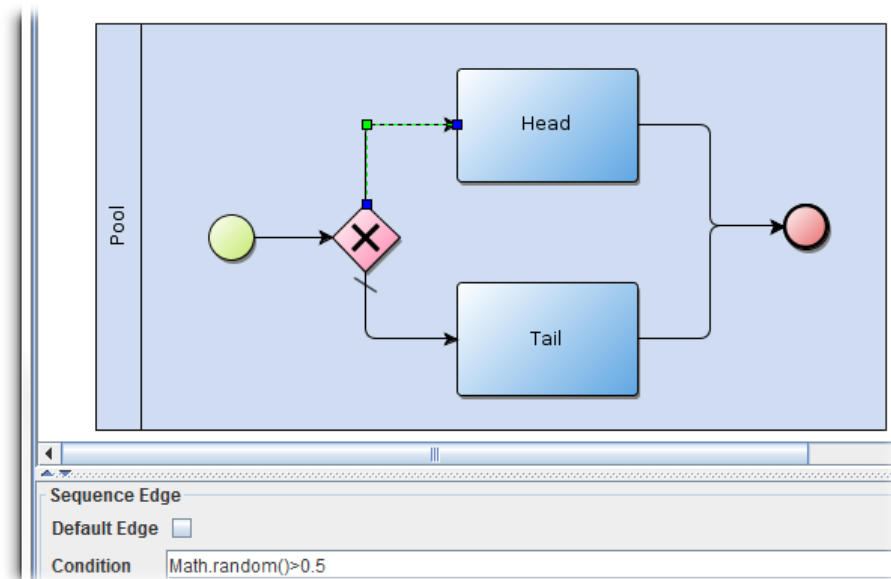


Figure 28: 03 Basic Processes@9.png

expression is evaluated when the process is executed. When the expression is true (i.e. a value greater than 0.5 is generated) the upper path is chosen. Otherwise the lower path is chosen, called the default branch. The default branch is indicated by the small dash. You can set the default branch by activating the ‘Default edge’ checkbox in the properties panel.

There are some more notable things about this process that you might have figured out yourself already. First, the XOR split does not have a corresponding XOR join, e.g. after the ‘Head’ and ‘Tail’ activities. Such a join is not necessary, because there are no multiple control flows executing at once. Remember that in the previous example, the AND join was the place where the process execution had to wait that the activities on both branches were completed before continuing the execution. An XOR join could be added for clarity in the B4 example process, e.g. to make the process more readable. Yet, at an XOR join the execution would not stop, because there is always *only one* incoming branch executed at all.

To execute the example process (either version) edit the ‘Head’ and ‘Tail’ to print some text to the console. Observe that when executing the process several times, either one or the other activity is executed.

Exercise B5 - Subprocesses

Besides basic tasks, BPMN also supports complex tasks, which are themselves composed of one or more activities. These complex tasks are called subprocesses. In Jadex, subprocesses can be either internal or external. Activities of internal subprocesses are drawn into the same diagram as the outer process. External subprocesses have their own diagram, which is referenced in the diagram of the outer (parent) process.

The subprocess element of the BPMN editor is used to specify both types of subprocesses. In this lesson, an *external* subprocess will be defined. An example of an *internal* subprocess can be found in Exercise C3 .

Defining a Subprocess

Draw a new process diagram as shown below. The 'Print Finished' task should be a PrintTask that prints out some finished message. Instead of drawing tasks into to subprocess (as you would do with an internal subprocess), the file name of an external diagram is specified. The 'file' property is used for this purpose. Enter the file name of another process, e.g. 'B2_Sequence.bpmn'.

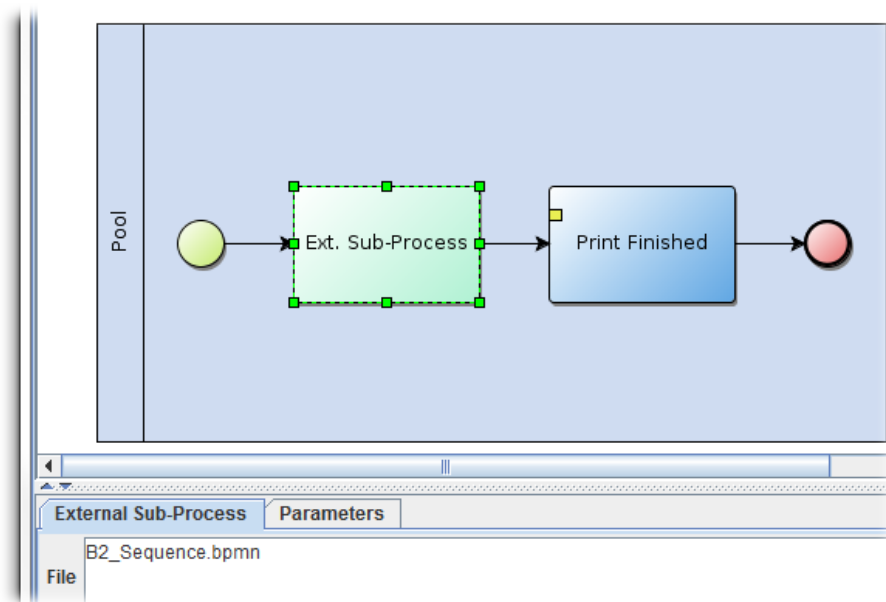


Figure 29: 03 Basic Processes@10.png

It is useful to understand, how the file of the subprocess will be loaded at runtime. The process files are loaded from the classpath in the same way that Java loads

Java classes. If you used a package for the processes, it is useful to set the package property as recommended in Exercise B1 . Because the B5 and B2 processes are in the same package, you do not need to fully qualify the name of the subprocess. Otherwise, you will have to write e.g. 'jadex/bpmn/tutorial/B2_Sequence.bpmn' or add a 'jadex.tutorial.bpmn.*' to the imports section of the outer process (in case you used jadex.bpmn.tutorial as package).

Execute the Process

Execute the process and observe its output. Verify that the tasks of the external subprocess get executed before the final task of the parent process. In the parent process, the subprocess and the final task element are in the sequence relation. Therefore the outer process waits for the subprocess to finish before executing any further activities.

You may also start the process in suspended mode and watch its execution in the debugger. As the outer process is in step mode, the subprocess will be started in step mode also. Thus you can see it appearing as a child of the outer process in the process tree.

Chapter 4 - Data and Parameters

This chapter covers the handling of data in processes. Data is made available as parameters, which can be global to the process or local to a task. Therefore, it is possible to map data to input/output values of activities as well as, e.g., storing results for later use.

Exercise C1 - Global Parameters

In this lesson global parameters are introduced. Global parameters can hold data value that belong to a process instance as a whole. The data is available to all tasks of the process and can be read and/or written depending on the parameter specification.

Specifying Parameters

Create a new diagram as shown below. Edit the process properties (i.e. select the pool) and switch to the parameters section. Add two parameters, one for holding the name of the customer and one for counting the number of logins. The first parameter is of type string and second one is an integer. You can use any Java type for a parameter, including custom Java classes. The value that can be specified in the process properties represents an initial value for

the parameter that is set when the process is instantiated. The value has to be specified as Java expression, e.g. a string has to be enclosed in quotes.

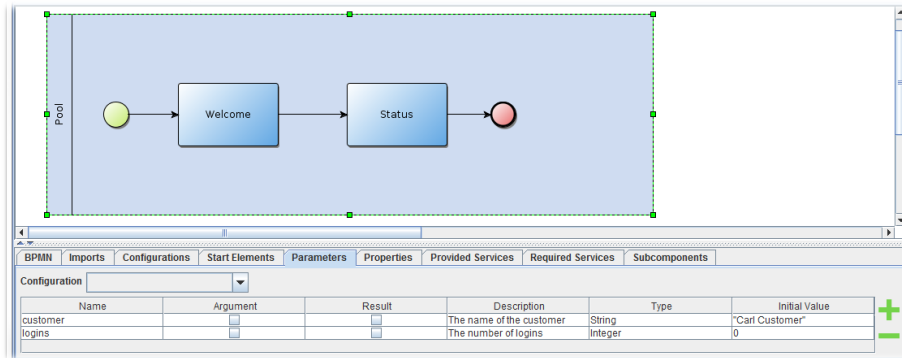


Figure 30: 04 Data and Parameters@1.png

Use a print task for the 'Welcome' and 'Status' activities. Print some text that includes the parameter values in the string expressions. E.g. set the texts to "Welcome"+customer+"!" for the welcome task and "Customer"+customer+" has logged in "+logins+" times." for the status task. If you run the process, you will see that the values of the parameters are filled in the text.

Updating Parameter Values

Now we want to increment the value of the 'logins' parameter after the welcome task. This can be done by using a new task of type 'WriteParameterTask' (or also 'WriteContextTask' which is only applicable for global parameters). Add a new task between the existing ones and set its type to 'WriteParameterTask'. In the task parameters fetch the default ones and delete the 'key' parameter (it is only needed in we want to insert values into collections of maps). The 'name' should be set to 'logins' and the value to 'logins+1', which access the old value and increments it.

Execute the process and observe that the incremented value is printed now.

Exercise C2 - Local Parameters

Some of the previous examples already have used local parameters for the 'text' value of the print task. This lesson takes a closer look at how local parameters work.

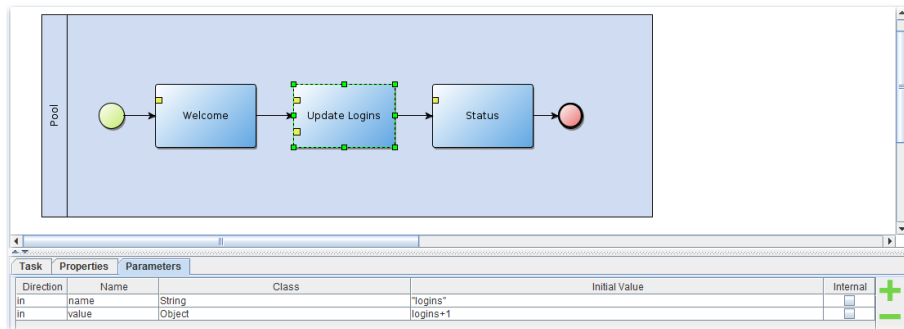


Figure 31: 04 Data and Parameters@2.png

User Interaction

Create a process as shown in the figure. Use the class 'jadex.bpmn.runtime.task.UserInteractionTask' for the 'Enter Address' task. The user interaction task displays all declared parameters and allows to input values for the parameters with direction 'out' and 'inout'. If you find using 'out' for input values counterintuitive you should take the perspective of the task and not the user. The values provided by the user represent the output of the task after its execution.

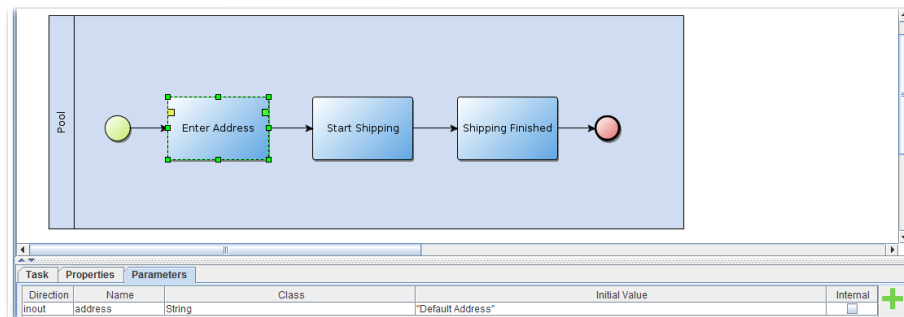


Figure 32: 04 Data and Parameters@3.png

Add a parameter 'address' of type string. Leave the other tasks empty for now (i.e. do not set a task class). Execute the process to see how the user interaction task works.

Using Local Parameter Values

Data flow between tasks is modelled using data edges. A data edge is a connection between the out parameter of a task with an in parameter of another task. Data edges allow for transferring parameter values between arbitrary tasks of the same

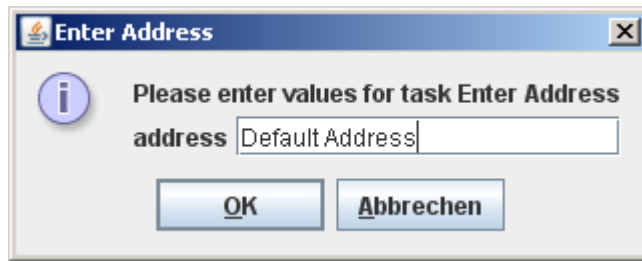


Figure 33: 04 Data and Parameters@userinteraction.png

level, i.e. one cannot draw a data edge to a task inside a subprocess - instead one has to first connect the subtask itself per data edge and afterwards route it to the corresponding task.

In the example we want to access the address the user entered in both subsequent tasks. To enable this we need to draw two data edges that connect the 'address' out parameter with in parameters of the other tasks. Make both tasks to PrintTasks and fetch the corresponding 'text' parameter. Then draw two edges connecting the 'address' parameter with the 'text' parameter for each task. In order to print not only the address itself, we want to modify the parameter value when transferred to the 'text' parameter by using a 'Value Mapping'. This can be found as property of the data edges. Set the first mapping to '“Shipping to:”+address' and the mapping of the second task to '“Arrived at:”+address' as indicated below.

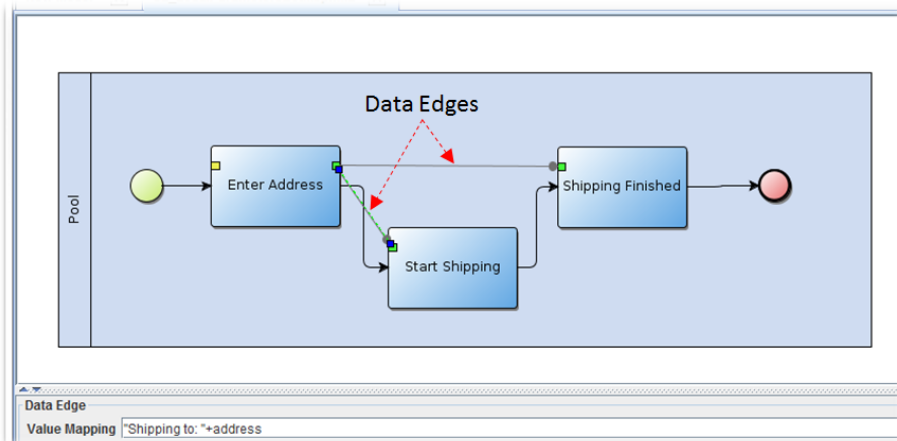


Figure 34: 04 Data and Parameters@4.png

Run the process and observe if the data arrives at the tasks.

Exercise C3 - Parameter Scopes

In BPMN, an internal subprocess represents a task that is recursively composed of inner tasks (see Exercise B5). The tasks in the subprocess are executed as if they were a separate process, but they have access to the context of the outer process.

Therefore subprocesses can be used to define custom scopes for parameters. This lesson shows how the example from the last lesson can be improved by using a subprocess as a parameter scope.

Create a Subprocess

Create a process similar to the one from C2, but place the shipment activities into a subprocess ‘Handle Shipping’. Add a parameter called ‘address’ of type string to the subprocess.

Edit the two last activities of the subprocess to print out some message that includes the ‘address’ value, e.g., ‘‘Shipping to:’’+address’ and ‘‘Arrived at:’’+address’. In the first activity, which uses the interaction task remove the ‘address’ parameter. The interaction task will check if its current task has own parameters and if no ones are declared it will search further upwards for parameters. Here, the subprocess declares a parameter ‘address’ which it will show and automatically assign to the user value.

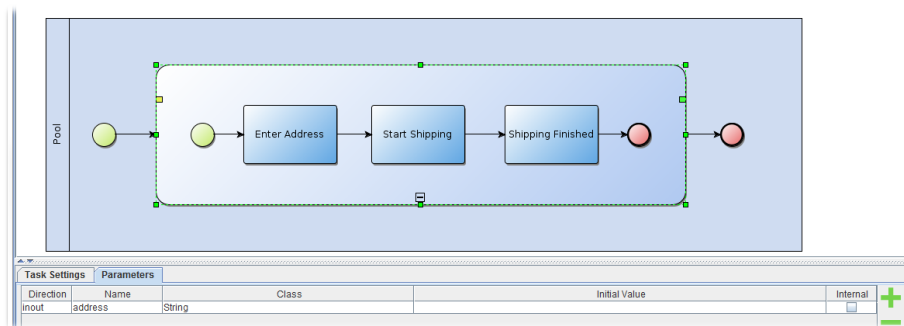


Figure 35: 04 Data and Parameters@5.png

Execute the process and verify that it works as expected.

This example shows that a subprocess scope can be used to avoid having to pass parameters through a sequence of tasks. Still the parameters of the subprocess do not clutter up the global namespace.

Chapter 5 - Events and Messages

This chapter deals with the dispatching and processing of events inside processes. Events can be e.g. timing events, errors or custom internal events.

BPMN distinguishes *start*, *intermediate* and *end* events. Intermediate events are further subdivided into *thrown* and *caught* events, i.e. events, which are produced or consumed, respectively. Start events are always considered to be caught (consumed), while end events are thrown (produced). The BPMN Poster presents an overview of available event types.

In the previous lessons you have already come across the empty start and end events. These empty events have no additional semantics and only serve for readability purposes. In the following event types will be introduced, that influence how the process is executed.

Exercise D1 - Timer Events

The timer event represents some passing of time. As time is outside the control of any process, this event only exists as *caught* event, i.e. a process can react to the passing of time, but the process can not influence time itself.

Create the Time Event Process

Create a process as shown in the figure below. This example represents a reminder process, e.g. when some order is late. First, an initial reminder is sent. If nothing happens after some delay, a second reminder is sent.

After placing the time event in the diagram, left-click on the event element to edit its properties in the properties view. Edit the *duration* property and enter some value, e.g., '3000'. The event duration specifies the time that the process will wait before continuing. It is specified in milliseconds, i.e. entering '3000' will cause the process to wait for three seconds.

Observe the Process Execution

Use print tasks for the first and second reminder, such that you can observe the process execution in the console. Start the process and verify that there is a three seconds delay between the first and the second message.

Now start the process in suspended mode and switch to the debugger. Execute the process stepwise. You can notice that the event intermediate activity changes its status from ready to waiting (switching from red to green), when you click 'Step', but the process does not proceed to the next activity. This means that the process has started the time event activity and still waits for this activity to

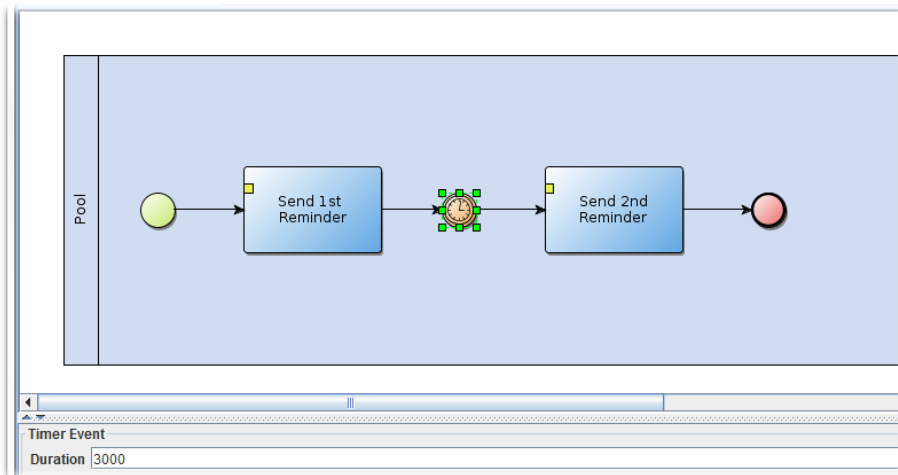


Figure 36: 05 Events and Messages@1.png

finish. The time event activity finishes after three seconds, thus after the time has passed, the process proceeds to the second reminder activity automatically.

Exercise D2 - Exceptions

Exceptions are situations and events that deviate from the normal course of action. Usually you want to describe in a process what happens, when everything goes well. But unexpected things can happen and sometimes this needs to be reflected in the process description.

Thus, exception events can be attached to subprocess elements. When an exception occurs during the execution of a subprocess, the subprocess does not continue, but instead the exception exit is triggered.

An Exception Process

Consider the following process. It starts with one activity: 'Credit Check'. If the credit check is successful the process proceeds to the 'Credit Approved' task. If some problem occurs during the credit check, the exception exit is triggered and the process moves on to the 'Credit Denied' task.

Failed and Successful Tasks

Use a `UserInteractionTask` for the credit check. When executing the process, a dialog will be opened offering you two buttons 'OK' and 'Cancel' (or similar

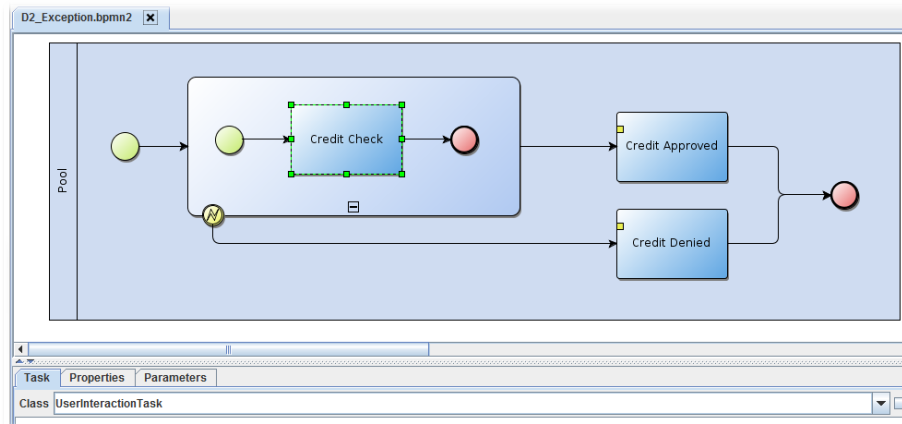


Figure 37: 05 Events and Messages@2.png

options based on your locale). Pressing cancel will cause an exception in the task. The process catches this exception and continues with the credit denied task.

Try out what happens, when you omit the exception handler. Delete the exception handler and the credit denied task from the diagram. Execute the process and choose ‘Cancel’. An exception like ‘java.lang.RuntimeException: Task not completed’ will be printed to the console and the process will be removed from the platform. Whenever a task produces an exception which is not handled in the process, the process will be terminated. Thus it is important to use exception handlers appropriately, when you have tasks that might fail.

Exercise D3 - Receiving Messages

Using messages, processes can communicate with the outside world. The outside world may be another process or an external software system (e.g. a web service). A message is represented as a set of name/value pairs, which specify, e.g., the sender and receiver of the message as well as its content.

Message Intermediate Event

Draw a process as shown below. The process has a preparatory task (e.g. print out a message to the console), then waits for a message being received and finally prints out the content of the received message.

Per default, the message intermediate event reacts to any received message, so you do not need to edit this element at first. The received message is stored in a param-

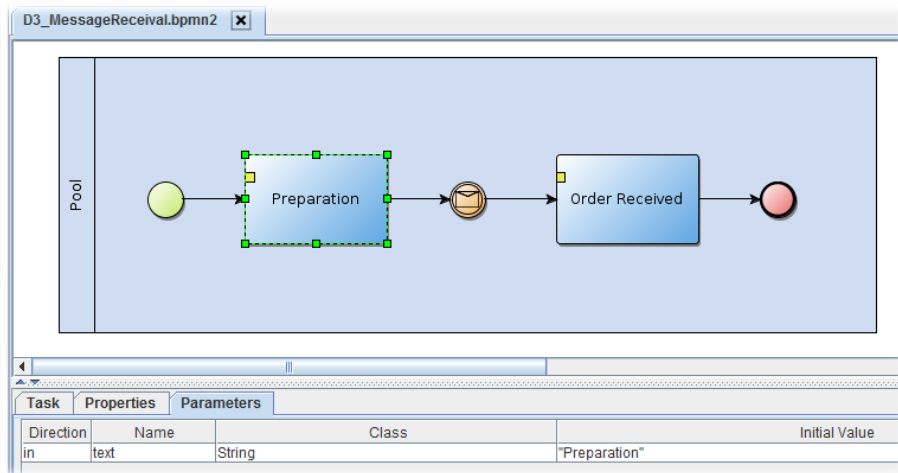



Figure 38: 05 Events and Messages@3.png

eter called 'event', which can be used in the subsequent activity. E.g. use `aprintt` task for the 'Order Received' activity with `"+event.content"` to print out the content of the received message.

Using the Conversation Center

To send the process a message, you can use the conversation center. Choose the conversation center from the toolbar in the JCC. It is represented by the

envelope icon ().

The tree on the left of the conversation center shows the components that are currently running on the platform. If you have started your process in the starter tool, you should see it in the tree. If you double click on an icon on the tree, the component will be added as a receiver of the current message.

The two lists in the middle show messages that have been sent or received previously. These lists will probably be empty when you first open the conversation center. The last sent messages are saved when you close the project or exit the platform. This is useful if you have created some test messages to send to your processes, so you can easily reuse them in subsequent sessions.

The right side shows the currently edited message. You can see that the message has a number of properties that can be specified (sender, receiver, etc.). Double click on your process in the left tree and observe that it is added as a receiver. If you want to remove a receiver, you may click on the 'X' icon besides the receivers box. This will clear all receivers. The text box at the bottom is used for the content of the mes-

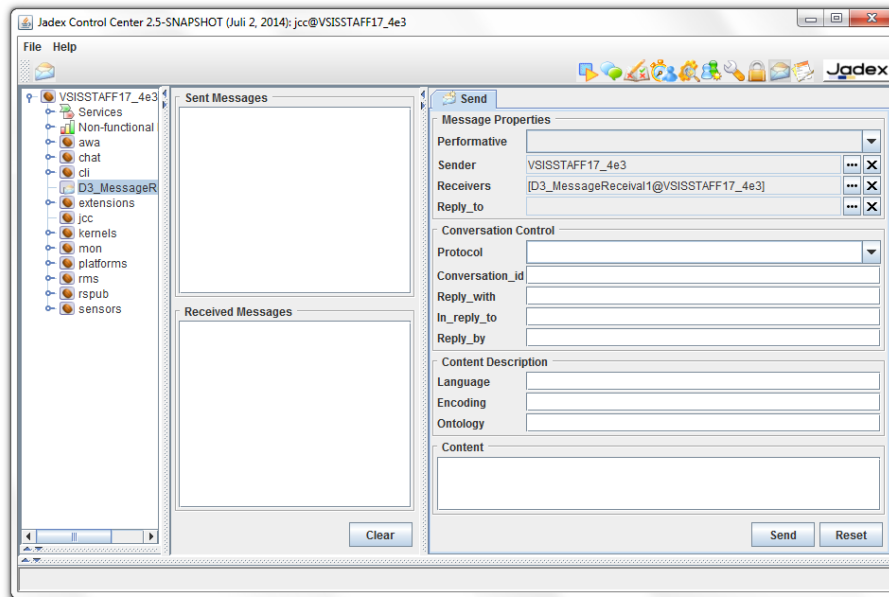


Figure 39: 05 Events and Messages@4.png

sage. This is what the process will print out, because we used the expression *'event.content'* in the text parameter. You can also access the other parameters of the message in a similar way, e.g. *'event.performative'*.

After setting the receiver and the content, hit the 'Send' button below the message. Observe that the process continues and prints out the text as received in the message.

FIPA Message Structure

You may wonder, what all the message properties are about. Jadex allows (in principle) different types of messages. The type of the message constrains the available parameters of a message. Currently, the only available type is "fipa" which defines parameter(set)s according to the FIPA message specification (e.g., parameters for the receivers, content, sender, etc. are introduced). Through this message typing Jadex does not require that only FIPA messages are being sent, as other options may be added in future. In the following table, all available parameter(set)s are itemized. For details about the meaning of the FIPA parameters, see the FIPA specifications available at FIPA ACL Message Structure Specification . The meanings of all of these parameters are shortly sketched in the following table.

Name	Class	Meaning
performative	String	Speech act of the message that expresses the senders intention
sender	IComponentIdentifier	The senders component identifier, which contains besides other
reply_to I	ComponentIdentifier T	he component identifier of the component to which should be r
receivers [set] IC	omponentIdentifier Ar	bitrary many (at least one) component identifier of the intende
content	Object	The content (string or object) of the message. If the content is
language	String	The language in which the content of the message should be en
encoding	String	The encoding of the message.
ontology	String	The ontology that can be used for understanding the message o
protocol	String	The interaction protocol of the the message if it belongs to a co
reply_with S	tring R	reply-with is used for assigning a reply to a original message. Th
in_reply_to St	ring Us	ed in reply messages and should contain the reply-with content
conversation_id S	tring T	he conversation-id is used in interactions for identifying messag
reply_by D	ate T	he reply-by field can contain the latest time for a response mes

Reserved FIPA message event parameters

Message Matching based on Parameters

Sometimes a process should not react to just any message, but only to messages that match a given template. In the message intermediate event, you can add parameters and corresponding values. If such parameters are specified, the event will only match those messages, where the parameter values are the same as in the event specification.

Edit the properties of the message intermediate event in the diagram. Add a parameter ‘performative’ with the value ‘“request”’. Use quotes for the value, because the value should be a string in this case.

Save and start the process. Send messages to the process using the conversation center. First send some messages with a performative other than ‘request’. Verify that the process does not react to these messages. Then send a request message and check that the process reacts to it.

Exercise D4 - Multiple Events and Timeouts

When waiting for an event such as an incoming message, this branch of the process is blocked until the event occurs. In reality, sometimes events do not happen as expected. This would cause a process to be blocked forever. A common way to deal with this problem is using timeouts, i.e. only waiting for a limited amount of time. If the events does not happen within the given time frame, the process continues with another branch.

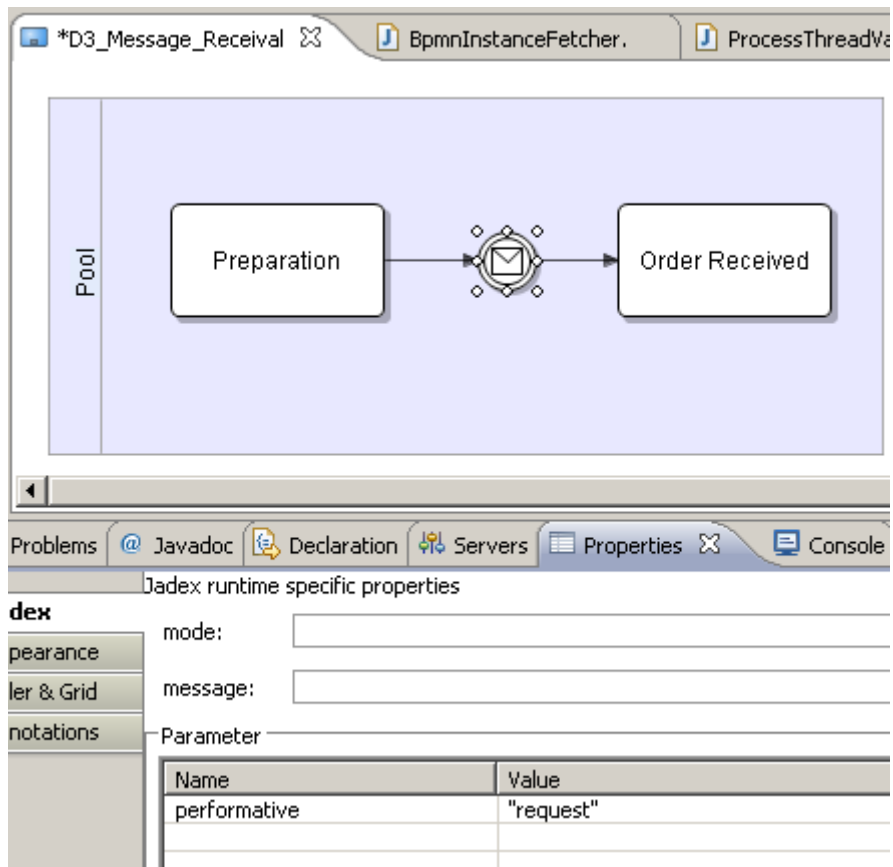


Figure 40: 05 Events and Messages@eclipsemessagereceivalparameter.png

In BPMN there exists a *multiple event* element, which represents a choice between a set of events. You can connect an arbitrary number of other events to the multiple event. If one of these events happens, the process continues on the branch, where the event is placed. Therefore, you can e.g. wait for two different messages specified by different message event elements (e.g. distinguishing the message content or performative) and continue on different branches of the process depending on the message received.

To wait for a message with a timeout, you can use a multiple event to combine the message event with a time event. If the message is not received within the time specified in the time event, the time event will trigger and the process continues.

A Process with a Timeout

The above mentioned timeout pattern is shown in the following process description. The process combines the elements from exercises D1 and D3.

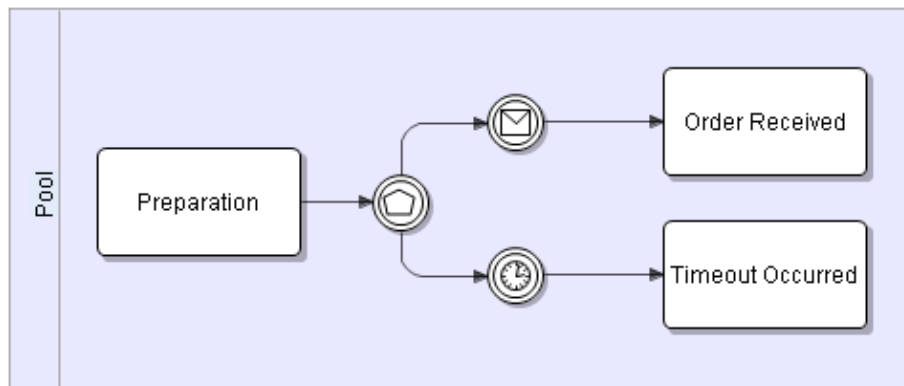



Figure 41: 05 Events and Messages@eclipsetimeout.png

Note the use of the multiple event (). In eclipse, it is available from the *Intermediary Events* section of the palette and is called *Multiple intermediate event*. Set a conveniently long timeout like e.g. 5000 milliseconds in the time event, because you need to send the process a message manually for testing.

Testing the Timeout Process

Start the process in the starter tool. Wait the specified amount of time to see, if the timeout works as expected. Now start the process again, switch to the conversation center and send the process an appropriate message. Verify that the process handles the message as expected.

If you have problems sending the message in time, you can increase the timeout value. Alternatively, you can start the process in suspended mode. Now you have as much time as you want to send the process a message. After sending the message you can resume the process. The process will then handle the message and terminate.

A process may receive an arbitrary number of messages even when suspended. Each process has its own message queue, where received messages are stored until they are handled.

Exercise D5 - Sending Messages

To send a process a message, we need to know how to address the process. The addressing depends on the message type. The `fipa` message type that Jadex supports out-of-the-box uses so called component identifiers for addressing. You have seen component identifiers in the previous lessons when using the conversation center.

A component identifier is composed of a unique name, usually of the form '@', e.g. 'MessageReceival@lars'.

Additionally, a component identifier may hold a number of transport addresses, which tell the platform how to reach the component, when it resides on a remote platform.

Sending a Message in a Process

For sending a message, add a message intermediate event as shown below. As mentioned earlier, BPMN distinguishes between *catching*- and *throwing*-events. The events we have used so far, were all catching-events which are represented by white icons within the BPMN event symbol. In contrast, throwing events are represented by black icons (see below). While catching a message is an analogy to receiving, throwing is an analogy to sending a message. Because of this, the event's mode has to be changed by right-clicking on the symbol and choosing "Set as a throwing shape". Add message parameters as required (cf. ToDo: Update screenshot !

We want to send a message to the process from Exercise D3. Thus, set the performative to request as expected by the D3 process and enter an appropriate content for the message. Finally, we need to specify the recipient of the message. You can use the helper method `createComponentIdentifier()` to create a local component identifier. The method is defined in the BPMN interpreter object, which is available from the reserved variable '\$interpreter'. Supply the local name of the process without the platform name, e.g., 'MessageReceival'.

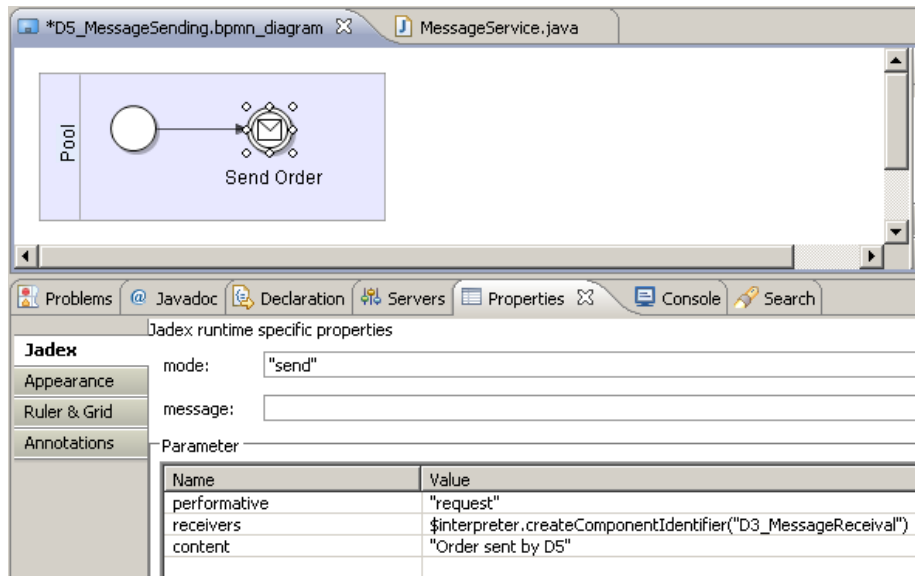


Figure 42: 05 Events and Messages@eclipsemessagesending.png

Testing the Process

Start the process. A warning message ‘Message could not be delivered to receiver(s)’ will be printed to the console. This is because a message was sent, but the platform did not find the intended receiver. Whenever you send a message to a non-existent recipient, you will observe such a warning message.

Now start the process from exercise D3. The D3 process waits indefinitely for a request message. When the process is started, start the D5 process. The message will be delivered and the D3 process will print out the received order. If it does not work as expected, make sure that the name of the D3 process instance is the same as specified as receiver in the D5 process description.

Observing Messages using the Communication Analyzer

The JCC includes a tool to monitor the messages that are exchanged between processes: the communication analyzer (🔍). Open the tool by selecting its icon from the upper right tool bar in the JCC. The window is divided in two main areas. On the left, there is a tree of the currently running agents. On the right you can see the observed messages in four different views: as a table, as a sequence diagram, as a 2D graph, and as a bar or pie chart. Furthermore, details of a selected message can be shown at the bottom.

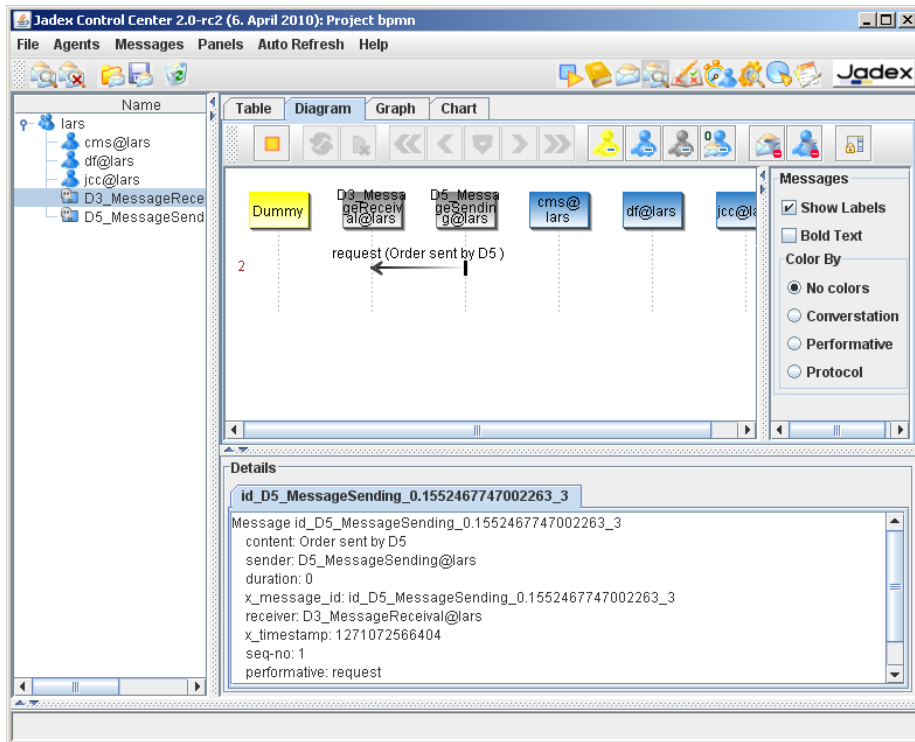


Figure 43: 05 Events and Messages@jadexcomanalyzer.png

To observe the message between D3 and D5 perform the following steps. Start D3 (using the starter tool). Switch to the communication analyzer and double click on the D3 process. You will see that the icon is enriched with a small looking glass. This means that the communication analyzer now observes this process and therefore notices any message that this process may send or receive. Now start the D5 process. The message will be exchanged and both processes will terminate. The tree in the communication analyzer will now show both processes with a small ghost icon, meaning that these processes are no longer alive.

The different views of the communication analyzer can consume quite some resources. Therefore they have to be activated separately. Switch to the 'Diagram' view and click the start button (the yellow triangle at the top left). The recorded message will be displayed. Double click on the message arrow to see details about the message (timestamp, etc.). Switch to other view like 'Graph' and see how the message is displayed there.

The communication analyzer is a very powerful and complex tool. When you right-click on a message, you get access to various filtering options. The filters are very helpful, if you have a larger application with many messages being exchanged. A complete discussion of all the features of the communication analyzer is outside the scope of this tutorial. Feel free to experiment with the tool to explore its functionality in more depth.

todo: further topics

- data
 - parameters in AND-join
 - loops
 - complex Java values (business objects)
 - reserved variables: `$interpreter`, `$thread`, `$event`, `$platform`, `$clock`, `$args`, `$results`
 - arguments and results
- tasks
 - predefined tasks (esp. create comp for external subprocess)
 - custom tasks
- other elements (lanes etc. - discuss only, no lesson?)

Chapter 6. Custom Functionality

In the previous chapters the, processes have been built only from the built-in functionality of Jadex BPMN. For any practical application one needs to add application-specific functionality to the system. This chapter deals with the various ways to extend Jadex BPMN and introduce custom functionality.

Exercise E1 - Custom Java Objects

Many processes are about the processing of data. While data can be represented in basic data types such as integer and string it is usually advantageous to have application-specific data type, so called domain or business objects. This exercise shows how to use custom Java classes to represent data and operations on that data.

For this purpose, we take the example of an insurance company that wants to sell a contract to a new customer. Based on customer properties, it is decided, if a high-risk or a standard contract is sold.

Defining the Customer Object

The customer should be represented in its own Java class. In addition to customer properties such as name, age, gender, and marital status, the class should implement a function to assess the risk of the customer. A simple business rule is used here: a customer is assumed to be risk taking, if it is a single male of age below 40. The code of the Customer class is shown below.

```
package jadex.bpmn.tutorial;

public class Customer
{
    protected String name;
    protected String gender;
    protected int age;
    protected boolean married;

    public Customer(String name, String gender, int age, boolean married)
    {
        this.name = name;
        this.gender = gender;
        this.age = age;
        this.married = married;
    }

    public boolean isRiskTaking()
    {
        return gender.equals("male") && age < 40 && !married;
    }

    public String toString()
    {
        return "Customer("+name+")";
    }
}
```

```

    }
}

```

For simplicity Java class should be created in the same package as the process. If you want to create the class in a different package, you need to add a corresponding import statement in the process (see below).

Creating the Process

Create a new process in the same package as the customer class. Make sure to set the ‘package’ property of the process accordingly, otherwise the customer class can not be resolved and the process will not execute.

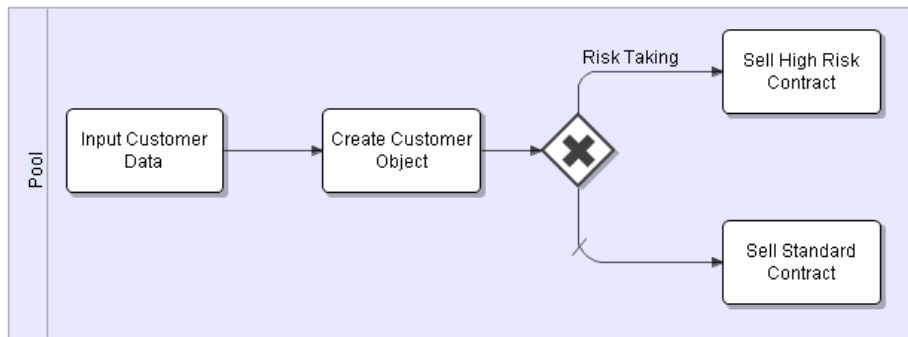


Figure 44: 06 Custom Functionality@06 Custom Functionality@eclipsecustomobject.png

Use a `UserInteractionTask` with ‘name’, ‘gender’, ‘age’, and ‘married’ as in-parameters for the ‘Input Customer Data’ activity. Set the types of the parameters to string, string, int, and boolean as required for the attributes of the customer Java class. Also, set some values for the parameters. This will save typing when later testing the process.

The ‘Create Customer Object’ activity does not require a task class. Just add an out-parameter ‘customer’ of type ‘Customer’ with value ‘new Customer(name, gender, age, married)’. The value is a Java expression, which calls the constructor of the customer class with the values provided by the previous ‘Input Customer Data’ activity.

Set the condition of upper flow connector (‘Risk Taking’) to ‘customer.isRiskTaking()’. The condition executes the business rule defined in the Java class. The ‘Sell High Risk Contract’ and ‘Sell Standard Contract’ activities can be set to `UserInteractionTask` again. Thus, when executing the process, we can observe, which path has been taken.

Executing the Process

Start the process and enter some customer values (or use the default values, if you have specified some in the properties). Test that a standard or high risk contract will be sold depending on the entered customer data.

If the process produces an error, e.g. ‘Class Customer not found in imports’, you should make sure that the customer class is in the same package as the process and that the ‘package’ property of the process is set appropriately. Also, you can verify that the directory that you added in the JCC contains the process .bpmn file as well as the compiled Java .class file.

Exercise E2 - Custom Tasks

Most of the functionality of a process is encapsulated in the tasks. For complex functionality, one usually needs to provide custom implementations of task behavior that go beyond simple print statements. Writing custom tasks is the simplest way to add application-specific business functionality to the BPMN engine. Tasks can be developed to be only used in a single process or to be reused across many processes of an application.

Jadex BPMN allows two types of tasks: 1) simple atomic tasks, that block process execution until they are finished, and 2) asynchronous tasks, that only block the branch of the process that contains the task, while other branches of the process can continue to execute. This lesson introduces the first type of task. The following lesson will then introduce the second type and highlight the differences between both.

A Simple OK Task

Suppose we want to have a simple task that opens a requester with only an ‘OK’ button. This functionality can easily be achieved using the JOptionPane from swing. The following code shows how to wrap this functionality into a task implementation that can be embedded into a Jadex BPMN process.

```
package jadex.bpmn.tutorial;

import javax.swing.JOptionPane;

import jadex.bpmn.runtime.BpmnInterpreter;
import jadex.bpmn.runtime.ITaskContext;
import jadex.bpmn.runtime.task.AbstractTask;

public class OKTask extends AbstractTask
```

```

{
    public void doExecute(ITaskContext context, BpmnInterpreter instance)
    {
        String message = (String)context.getParameterValue("message");
        String title = (String)context.getParameterValue("title");
        JOptionPane.showMessageDialog(null, message, title, JOptionPane.INFORMATION_MESSAGE);
    }
}

```

To implement a simple (synchronous) task, extend the *jadex.bpmn.runtime.task.AbstractTask* class. This class defines one abstract method, that you have to implement: *doExecute(ITaskContext context, BpmnInterpreter instance)*. In this method you can put any custom functionality as required by your application, e.g. simple calculations, calling legacy systems, etc. The parameters *context* and *instance* provide access to the running process, e.g. to read or write process data.

The ‘OKTask’ first reads two parameter values from the process context, which are used for the requester title and message. Then the *JOptionPane* is activated using the extracted parameter values.

Planning a Party

We use a simple checklist process to demonstrate the custom task. The process includes three unrelated tasks that are all mapped to the new ‘OKTask’ implementation. The process terminates, when all requesters have been closed. The picture below shows the process and also the inclusion of the ‘OKTask’.

As we expect that the checklist items can be checked in any order, we do not impose ordering restrictions between the tasks, i.e. there are no flow connectors in the process. Also, because the process and the custom task implementation reside in the same package, we can simply write ‘OKTask’ instead of a fully qualified name like ‘jadex.bpmn.tutorial.OKTask’. If you have problems (‘Class OKTask not found in imports’) remember to set the package property of the process accordingly.

In the BPMN diagram, the tasks contain two parameters: ‘message’ and ‘title’. These are the parameters that we used in the task implementation.

When the process is executed, the requesters for the tasks appear one after another. Because the *JOptionPane* blocks the thread when *showMessageDialog()* is called, the whole process waits until the requester is closed. Only then the process will activate the next task and open the next requester.

The simple atomic or blocking task type introduced in this lesson is most useful for functionality that quickly completes, such as a calculation or a database query. During the task execution, the whole process is blocked, which is advantageous with respect to consistency, e.g. no other task can alter process data during an ongoing calculation. The task type is less well-suited for long-term activities,

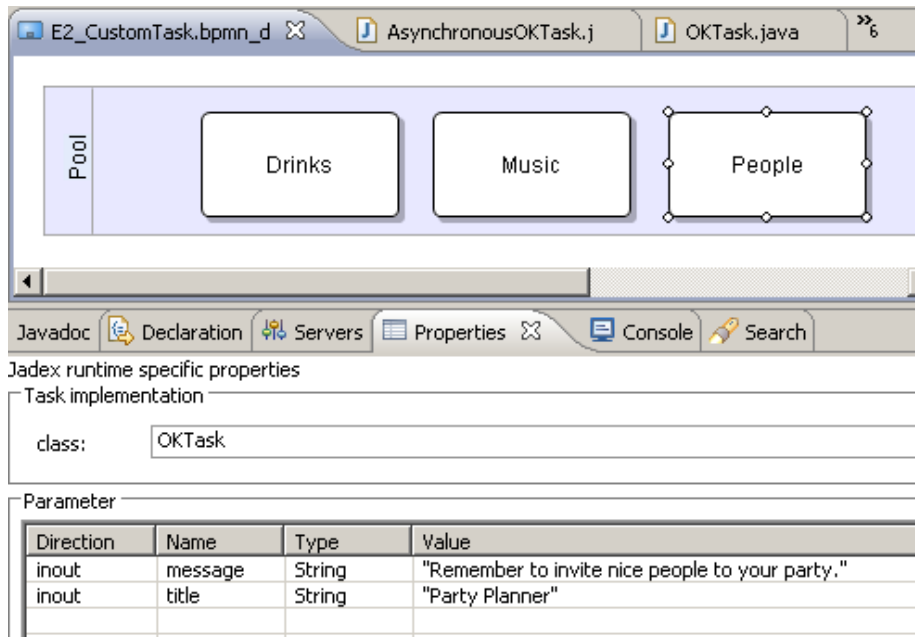


Figure 45: 06 Custom Functionality@eclipseparty.png

e.g. that involve human interaction. For this kind of activities asynchronous tasks as introduced in the next lesson are a better fit.

Task Meta Informaion

The *OKTask* requires the *message* and *title* parameter to be specified in the process. This is usually not obvious for other developers that might want to use your custom tasks. Therefore, you can add annotations to the task class that e.g. declare the parameters. Add the annotations from the 'jadex.bpmn.annotation' package to the *OKTask* as shown below.

```
@Task(description="A task that displays a message using a JOptionPane.", parameters={
    @TaskParameter(name="message", clazz=String.class, direction=TaskParameter.DIRECTION_IN, c
    @TaskParameter(name="title", clazz=String.class, direction=TaskParameter.DIRECTION_IN, des
})
public class OKTask extends AbstractTask
{
    ...
}
```

The annotations provide a human readable description for the task as well as information about the task parameters, such as name, type (clazz), direction, initial value (not shown) and a description of the parameter. When you edit the party process in the BPMN editor, you will notice that this information is displayed, when the task class is selected. In addition, you have the option to add the parameters to the parameter section of the properties editor.

Exercise E3 - Asynchronous Tasks

Consider the party planning checklist process from the last exercise. It makes perfect sense to execute the three tasks of the process (organizing drinks, music, and people) in parallel and the process description in BPMN imposes no restrictions with respect to parallel task execution. Yet, the implementation of the task causes the activities to be executed in sequence - one after the other.

Whenever there are external activities involved in a process, e.g. a human user working on a worklist item, it should be considered to execute this activity asynchronous to the process. Thus, different external activities (e.g. organizing drinks and inviting people) can be executed in parallel, when the process description allows this.

This lesson shows how to implement an asynchronous task.

Asynchronous Task Implementation

For a simple blocking task, you can implement the *doExecute()* method of the *AbstractTask* class. When the method returns, the task is completed and the process continues. For an asynchronous task, the implementation is somewhat more complex, because the activation of the task and the completion have to be handled separately.

To implement an asynchronous task, you have to implement the *execute()* method of the *jadex.bpmn.runtime.ITask* interface. When the method is called, the task execution is activated (e.g. inserting an entry into the work list of a workflow management system). When the method returns, the process will not continue, but wait until the callback result listener that is supplied as argument to the *execute()* method is called. Until then, the process may execute other parallel activities. When the task is completed (e.g. the work item is marked as finished by a user), the result listener has to be called, causing the process to continue.

```
package jadex.bpmn.tutorial;

import jadex.bpmn.runtime.BpmnInterpreter;
import jadex.bpmn.runtime.ITask;
```

```

import jadex.bpmn.runtime.ITaskContext;
import jadex.commons.SGUI;
import jadex.commons.concurrent.IResultListener;

import java.awt.Point;
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;

import javax.swing.JDialog;
import javax.swing.JOptionPane;

public class AsynchronousOKTask implements ITask
{
    public void execute(ITaskContext context, BpmnInterpreter process, final IResultListener l)
    {
        String message = (String)context.getParameterValue("message");
        String title = (String)context.getParameterValue("title");
        int offset = context.hasParameterValue("y_offset") ? ((Integer)context.getParameterValue("y_offset")).intValue() : 0;

        JOptionPane pane = new JOptionPane(message, JOptionPane.INFORMATION_MESSAGE);
        final JDialog dialog = new JDialog((JDialog)null, title, false);
        dialog.setDefaultCloseOperation(JDialog.DO_NOTHING_ON_CLOSE);
        dialog.setContentPane(pane);
        dialog.pack();
        Point loc = SGUI.calculateMiddlePosition(dialog);
        dialog.setLocation(loc.x, loc.y+offset);

        pane.addPropertyChangeListener(new PropertyChangeListener()
        {
            public void propertyChange(PropertyChangeEvent e)
            {
                String prop = e.getPropertyName();
                if(prop.equals(JOptionPane.VALUE_PROPERTY))
                {
                    dialog.setVisible(false);
                    listener.resultAvailable(this, null);
                }
            }
        });
        dialog.setVisible(true);
    }
}

```

Chapter 1. Introduction

The Jadex BDI kernel is a Belief-Desire-Intention reasoning engine for intelligent agents. The term reasoning engine means that it can be used together with different kinds of (agent) middleware providing basic agent services such as a communication infrastructure and management facilities. Currently, two mature adapters are available. The first adapter is the Jadex Standalone adapter which is a small but fast environment with a minimal memory footprint and the second is available for the well-known open-source JADE multi-agent platform [Bellifemine et al. 2007].

In this tutorial the Jadex Standalone adapter is used, but in principle the used adapter is not of great importance as it does not change the way Jadex agents are programmed or way the Jadex tools are used. The concepts of the BDI-model initially proposed by Bratman [Bratman 1987] were adapted by Rao and Georgeff [Rao and Georgeff 1995] to a more formal model that is better suitable for multi-agent systems in the software architectural sense. Systems that are built on these foundations are called Procedural Reasoning Systems (PRS) with respect to their first representative. Jadex builds on experiences gained from leading existing BDI systems such as JACK [Winikoff 2005] and consequently improves previously not-addressed BDI weaknesses like the concurrent handling of inconsistent goals with built-in goal deliberation [Pokahr et al. 2005a].\

- Chapter 2, Starting an Agent describes how to setup the Jadex environment properly and how to start a simple agent.
- Chapter 3, Using Plans explains step by step the usage of plans.
- Chapter 4, Using Beliefs introduces beliefs and beliefsets as agent knowledge form.
- Chapter 5, Using Capabilities explains how beliefs, goals and plans can be composed into reusable agent modules.
- Chapter 6, Using Goals shows how goals can be used to capture the agent objectives in an intuitive way.
- Chapter 7, Using Events covers aspects about information exchange on the intra and inter-agent level and builds up a multi-agent scenario.
- Chapter 8, External Processes explains exemplarily the integration of Jadex agents with external processes.
- Chapter 9, Conclusion finally concludes the lessons.
- Bibliography contains the references.

Application Context

In this tutorial a simple translation agent for single words will be implemented. This agent has the basic task to handle translation requests and produce for a given term in some language the translated term in the desired target language. This base functionality will be extended in the different exercises, but it is not our

goal to build up a translation agent, that combines all the extensions, because this would lead to difficulties concerning the complexity of the agent. Instead this tutorial will concentrate on setting up simple agents that explain the Jadex concepts step by step.

How to Use This Tutorial

- Work through the exercises in order, because later exercises require knowledge from the earlier ones.
- Don't destroy your solutions of an exercise by modifying the old files. The different exercises often use the plans and agent description files (ADF) of a preceding exercise. Copy all files and apply a simple naming scheme which contains the name of the exercise in the plan and ADF file names, e.g. the ADF in the exercise A1 is called TranslationA1.agent.xml and in exercise B1 TranslationB1.agent.xml.
- Help us to make this tutorial better with your feedback. When you find errors or have problems that are directly concerned with the exercise descriptions feel free to let us know or edit the corresponding page directly in the Wiki.
- Whenever you encounter problems with Jadex we would be happy to help you. Please use primarily the Jadex mailing list that can be used for asking questions about Jadex.

Chapter 2. Starting an Agent

Setting up the Jadex environment properly is pretty easy and can be done in a few simple steps. Generally, Jadex is realized as reasoning engine meaning that it can be used on top of different (agent) middlewares. In this tutorial we will use the Standalone version of Jadex. The Jadex distribution should be extracted to some local directory, called JADEX_HOME here. Then you basically have two options how to start Jadex. The first is by setting the Java CLASSPATH variable by hand to all the jars contained in the JADEX_HOME/lib directory. The second one is by using the -jar option when starting via the java command.

The alternative commands for launching the Jadex Standalone platform are:

```
java jadex.standalone.Platformor\ java -jar jadex-launch-2.0-rc1.jar\
```

Exercise A1 - Creating a Project

Start the Jadex platform with the command explained above. After some short time the Jadex Control Center (JCC) should show up with its user interface. The JCC provides a project management facility that simplifies working on specific


applications. Basically, a project contains settings about the used project folders as well as miscellaneous tool and user interface settings. All your settings - as window settings, added paths and so on - will be stored in a `.project.xml` file and additional properties-files will be created automatically to store the individual settings for the plugins you are using. \

To add files to your project, hit the “Add Path” button. For saving the project settings on disk select “Save Settings” from the “File”-menu. Additionally, the settings are also saved automatically when shutting down the JCC via its window close button or “Exit” from the “File”-menu. You can also use multiple projects with different settings. To switch projects simply click on “Load Settings from File” in the “File”-menu and choose the settings file of the project you want to work with.

Verify project behaviour. \ In order to verify that you project has been set up correctly you should perform some visible changes such as changing the JCC’s window size or switching to another plugin than the starter. After that you should shut down the JCC and platform and restart it. If the project has been created correctly, the JCC will show up in exactly the same state you left it, i.e. the last active plugin will be activated and the gui settings are remembered, too.

Exercise A2 - Executing Example Applications

The Jadex distribution already contains a couple of example applications. The objective of this exercise consists in trying out the examples and also in roughly understanding their application purpose. Open the JCC and select the starter view as described in exercise A1. On the left hand side of the starter you can see the agent and application files that can be loaded in a tree like structure. As top-level elements the starter can handle jar files or directories representing the root of Java packages (often named “classes”, “bin” or “build”). Since the newer versions of Jadex a default project is automatically loaded at startup so that you can already see example folders in that panel. In case you want to add

a new directory with agent models, you should choose the  button (“Add Path”) and browse to the corresponding directory you want to add. All Jadex examples are contained in the different ‘jadex-applications-xyz.jar’ files in the `JADEX_HOME/lib` directory. The content of the new jar will be added as new node to the model tree. At the bottom you can now see this jar-file beeing scanned (if the scanning option has not been disabled). Next, you can expand the model tree by double clicking on the corresponding top-level node, which represents the jar-file. After opening the folders “jadex” and “examples” you will see the different example folders containing several different kinds of single- and multi-agent applications. Concretely the following BDI example applications are currently available:

- **Alarmclock:** An agent that shows up a small clock user interface and is able to manage alarm times and play mp3 songs when an alarm is due.
- **Blackjack:** Agents that play blackjack. It is also possible for a human player to join the table and play against the dealer.
- **Blockworld:** Stacking blocks on a table to reach a specific target configuration of blocks.
- **Booktrading:** Buyer and seller agents that negotiate about book prices.
- **Cleanerworld:** Simulated cleaner robots collecting waste at day and patrolling at night.
- **Garbagecollector:** Simplified version of the cleaner task using a grid world.
- **Helloworld:** Very simple agent that prints “hello world” to the console output and then kills itself.
- **Hunterprey:** Hunters search for preys (living food) in a virtual world.
- **Marsworld:** Cooperative exploitation of ore on mars by different kinds of robots.
- **Ping:** Simple agents that send ping and reply messages.
- **Puzzle:** An agent that tries to solve a puzzle by trying out moves and taking them possibly back.

Starting an example can in most cases be done by opening the corresponding folder and searching for an application file (a file ending with “.application.xml”). Such an application definition has the purpose to start-up all relevant application agents and may also setup further application components. In case there is no application file the example can be started by selecting the agent directly (e.g. Helloworld or Puzzle).

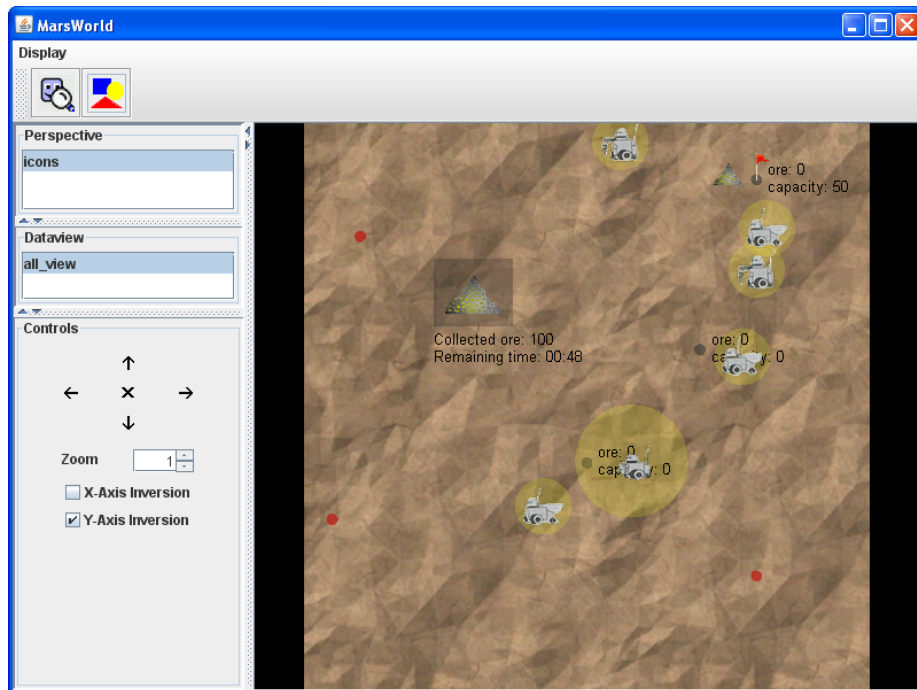


Figure 1 The marsworld example

In the same way you can add other paths and execute your own applications and agents later. In this case you will not add jar-files, but the root directory of your packages.

Explain example behaviour. Choose one of the more complex examples for a more detailed analysis. Select the application of the example and read through its documentation shown in the starter panel. Then look into the selected example directory and read also the documentation of the agents which belong to that application. Finally, open the source code of these agents in your source code editor and try to grasp roughly of what they are comprised. Write down a (simple!) explanation how the multi-agent system and the involved agents work.

Exercise A3 - Create first simple Jadex agent

Open a source code editor or an IDE of your choice and create a new agent definition file (ADF) called TranslationA1.agent.xml (cf. Figure 2, “A1 XML ADF”). We recommend using eclipse with web-tools plug-in for editing ADFs or some other advanced XML-Editor. In this file all important agent startup properties are defined in a way that complies to the Jadex schema specification. First property of the agent is its type name which must be the same as the file name (similar to Java class files), in this case it is set to TranslationA1. Additionally you can specify a package attribute, which has a similar meaning as

in Java programs and serves for grouping purposes only (you will need to alter the package name with respect to your actually used directory structure). All plans and other Java classes from the agent's package are automatically known and need not to be imported via an import tag.

```
\
\  
  
<!-- A simple translation agent. -->  
<agent xmlns="http://jadex.sourceforge.net/jadex"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
  xsi:schemaLocation="http://jadex.sourceforge.net/jadex  
    http://jadex.sourceforge.net/jadex-bdi-2.3.xsd"  
  name="TranslationA1"  
  package="jadex.bdi.tutorial">  
</agent>
```

```
\ \  
Figure 2 A1 XML ADF
```

Start your first Jadex agent. Start the JCC and use the “Add Path” button explained above to add the root directory of your example package. Then open the folder until you can see your file “TranslationA1.agent.xml”. The effect of selecting the input file is that the agent model is loaded. When it contains no errors, the description of the model, taken from the XML comment above the agent tag, is shown in the description view. In case there are errors in the model, correct the errors shown in the description view and press *reload*. Below the file name, the agent name and its default configuration are shown. After pressing the start button the new agent should appear in the agent tree (at the bottom left). It is also possible to start an agent simply by double-clicking it in the model tree. *Please note that when you use a double-click on the model name in the left tree view to start an agent, the settings on the right will be ignored.*

BEGIN MACRO: html param: clean="false" wiki="true"

<!--You can also start a second JCC by choosing it from:

```
\   
jadex/tools/jcc/JCC.agent.xml\
```

and giving it a name like JCC2.->

END MACRO: html

Chapter 3. Using Plans

Plans play a central role in Jadex, because they encapsulate the recipe for achieving some state of affair. Generally, a plan consists of two parts in Jadex. The plan body is a standard Java class that extends a predefined Jadex framework class (*jadex.bdi.runtime.Plan*) and has at least to implement the abstract *body()* method which is invoked after plan instantiation. The plan body is associated to a plan head in the ADF. This means that in the plan head several properties of the plan can be specified, e.g. the circumstances under which it is activated and its importance in relation to other plans.

In contrast to other well-known PRS-like systems, Jadex supports two styles of plans. A so called *service plan* is a plan that has service character in the sense that a plan instance of the plan is usually running and waits for service requests. It represents an easy way to react on service requests sequentially without the need to synchronize different plan instances for the same plan. Therefore a service plan can setup its private event waitqueue and receive events for later processing, even when it is working at the moment.

A so called *PRS-style or passive plan* is the normal version of a plan, as can be found in all other PRS-systems. This means that usually such a plan is only running, when it has a task to achieve. For this kind of plan the triggering events and goals must be specified in the agent definition file to let the agent know what kinds of events this plan can handle. When an agent receives an event, the BDI reasoning engine builds up the so called applicable plan list (that are all plans which can handle the current event or goal) and candidate(s) are selected and instantiated for execution. PRS-style plans are a good choice, when the parallel execution of one kind of task is needed or is at least not disturbing. For more detailed information about plans have a look in the BDI User Guide .

BEGIN MACRO: html param: clean="false" wiki="true"

Often a plan does some action and then wants to wait until the action has been done before continuing (e.g. dispatching a subgoal, sending a message and waiting for the reply). Therefore a plan can use one of the various wait-For() methods, that come in quite different flavors. Coming back to the examples mentioned, e.g. the `dispatchSubgoalAndWait(IGoal subgoal, long timeout)` can be used to dispatch a subgoal and wait for its completion (optionally with some timeout). Similar, for sending a message and waiting for a reply the `sendMessageAndWait(IMessageEvent me, long timeout)` method can be used. For an extensive overview of all available methods, please refer to the BDI User Guide or the API docs contained in the Jadex release. <!-- API documentation.->\

Exercise B1 - Service Plans

In this exercise we will use a service plan for translating words from English to German. Create a new TranslationB1.agent.xml file by copying the TranslationA1.agent.xml file and modify all occurrences of “A1” to “B1”.

<p/>

Create a new file called EnglishGermanTranslationPlanB1.java responsible for a basic word translation with the following properties:

- Create the plan as extension to the jadex.bdi.runtime.Plan class:

```
public class EnglishGermanTranslationPlanB1 extends Plan {
    // Plan attributes.

    public EnglishGermanTranslationPlanB1() {
        // Initialization code.
    }

    public void body() {
        // Plan code.
    }
}
```

- Import the needed classes: *jadex.bridge.fipa.SFipa*, *jadex.bdi.runtime.IMessageEvent*, *jadex.bdi.runtime.Plan*, *java.util.HashMap*, *java.util.Map*
- Let the no argument constructor print out the text “Created:”+this.
- Implement the plan’s body() method as infinite loop. At the beginning of this loop the plan should wait for translation requests using *IMessageEvent* *me = waitForMessageEvent(“request_translation”)*
- Instead of performing a database query let us use a simple hashmap for the word lookup. The creation and initialization of this word table with a few word pairs can already be done in the constructor. As result the plan should print **“Translating from English to German:”+eword+” - “+gword** or *“Sorry word is not in database:”+eword*. To get the content from the request-event use *me.getParameter(SFipa.CONTENT).getValue()*.

<p/>

Add the plan to the agent by putting it into the agent definition file:

- Therefore, a new plans section is introduced, in which all plans for the agent have to be declared. In this simple example only one plan named here “egtrans” is added. As part of the plan the Java class name for the plan body is stated. Additionally, the plan’s waitqueue is declared to handle all message events of type “request_translation”. This means that the plan has its own event waitqueue in which all matching events are dispatched,

even when the plan is busy and currently waits for other events. These events are collected in its queue till it calls a suitable *waitFor()* matching one of the collected events. In this case this collected event is directly dispatched to the plan.

- The plan should be started when the agent is born. For this purpose a configuration has to be declared within the ADF. It is sufficient in this case to define one configuration (named “default”) with an initial plan. The initial plan simply references the plan for which an instance should be created.
- Besides the introduction of the new plan we also need to make explicit what exactly a request_translation event means. For this purpose a new events section is introduced. In this section the request_translation event is declared being a message event with one parameter. This parameter specifies that its performative has the fixed value ‘request’. Whenever the agent receives a message it will search its declared events for the best matching event type. In this case all messages with performative request it will be treated as request_translation events.

```
<agent xmlns="http://jadex.sourceforge.net/jadex-bdi"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://jadex.sourceforge.net/jadex-bdi
    http://jadex.sourceforge.net/jadex-bdi-2.0.xsd"
  name="TranslationB1"
  package="jadex.bdi.tutorial">

  <plans>
    <plan name="egtrans">
      <body class="EnglishGermanTranslationPlanB1"/>
      <waitqueue>
        <messageevent ref="request_translation"/>
      </waitqueue>
    </plan>
  </plans>

  <events>
    <messageevent name="request_translation" direction="receive" type="fipa">
      <parameter name="performative" class="String" direction="fixed">
        <value>jadex.bridge.fipa.SFipa.REQUEST</value>
      </parameter>
    </messageevent>
  </events>

  <properties>
    <property name="debugging">>false</property>
```

```

</properties>

<configurations>
  <configuration name="default">
    <plans>
      <initialplan ref="egtrans"/>
    </plans>
  </configuration>
</configurations>
</agent>

```

Starting and testing the agent\
 <p/>

Create a translation agent via the Control Center and observe the standard output, if the initial plan is created at startup. Use the Conversation Center (in new Jadex versions the conversation center needs to be activated via popup menu on the toolbar, see Tool Guide) to send a translation request to the TranslationAgent by setting the performative to *request* and the content to some word to translate. Observe the TranslationAgent’s output on the console when it receives the request.

Exercise B2 - Passive Plans

In contrast to the last exercise we will now use a passive plan to react on translation requests. To show the difference between the two forms of plans we now modify the service plan slightly to become a passive plan. Create the files EnglishGermanTranslationPlanB2.java and TranslationB2.agent.xml by copying the files from exercise B1.

Modify the copied file TranslationPlanB2.java:\

- Replace all occurrences of “B1” in the Plan with “B2”
- In contrast to the initial plan, the passive plan’s body method is only invoked, when an event matches the plan’s trigger. So use the method *getReason()* to retrieve the event that caused the execution. Because we know that only certain messages activate the plan the event can directly be cast to type *jadex.bdi.runtime.IMessageEvent* and the content can be retrieved. The infinite loop in the body should be discarded, because for each event a new plan instance is created, which only handles a single message.\

<p/>

Modify the copied file TranslationB2.agent.xml\

- Replace all occurrences of “B1” in the ADF file with “B2”

- Modify the plan declaration in the ADF by removing the configurations section. Additionally a passive plan needs a trigger, that specifies under what circumstances a new plan instance is created. Therefore remove the waitqueue statement and add a new statement for the plan trigger:

```
<trigger>
  <messageevent ref="request_translation"/>
</trigger>
```

Starting and testing the agent\

Start the agent as explained in the preceding exercise. Observe that a new instance of the translation plan is created everytime an appropriate event arrives. The passive plan is instantiated and each instance processes a different message event. Many different plan instances may remain active while processing their triggers.

Exercise B3 - Plan Parameters

In this exercise we will use plan parameters to supply the plan with arguments. Plan parameters can directly be accessed from within the plan body via the `getParameter("paramname")` and `getParameterSet("paramsetname")` methods. Generally parameters can have the directions *in*, *out* and *inout* describing parameters that are used for supplying values or resp. gathering return values from the plan. Plan parameters can be supplied with fixed values via the `<value>` or `<values>` tags. More interestingly parameter values can be mapped from and to the triggers by using parameter mappings. If a plan could be activated by more than one trigger (e.g. two different messages, or a message and a goal, etc.) multiple goal mappings (one for each trigger type) have to be used to unify the plans view on its arguments.

Create the files `EnglishGermanTranslationPlanB3.java` and `TranslationB3.agent.xml` by copying the files from exercise B2. Apply the same replacements B2->B3 as in the previous exercise.

<p/>

Modify the EnglishGermanTranslationPlanB3.java

- Instead of using the `getReason()` method to retrieve the English word, we use the the statement: `String eword = (String)getParameter("eword").getValue();`\

<p/>

Modify the copied file `TranslationB3.agent.xml` to include the new plan parameter

- Add the new plan parameter with a message event mapping to the ADF:

```

<plan name="egtrans">
  <parameter name="eword" class="String">
    <messageeventmapping ref="request_translation.content"/>
  </parameter>
  <body class="EnglishGermanTranslationPlanB3"/>
  <trigger>
    <messageevent ref="request_translation"/>
  </trigger>
</plan>

```

Starting and testing the agent\

Test and verify that the agent behavior is the same as in the last exercise.

Exercise B4 - Plan Selection

In this exercise we will use plan priorities to establish a plan selection order. Create the files EnglishGermanTranslationPlanB4.java and TranslationB4.agent.xml by copying the files from exercise B2. Apply the same replacements B2->B4 as in the previous exercise.

Create a new plan file named SearchTranslationOnlineB4.java

- This plan should be used when the agent cannot find the word in its (currently very small) dictionary. In this case the on-line search plan will try to connect to a web dictionary and report the found translations. The address of a simple English-German dictionary is <http://wolfram.schneider.org/dict/dict.cgi> (you may use any other dictionary for this purpose if you are not afraid of parsing the result HTML page). To issue a query against this online database you need to create a URL and read the data from there as outlined below. You will have to add code for fetching the “eword” that should be translated and a try/catch block when creating the URL, i.e. if the dictionary is not available.

```

URL dict = new URL("http://wolfram.schneider.org/dict/dict.cgi?query="+eword);
BufferedReader in = new BufferedReader(new InputStreamReader(dict.openStream()));
String inline;
while((inline = in.readLine())!=null)
{
  if(inline.indexOf("<td>")!=-1 && inline.indexOf(eword)!=-1)
  {
    try

```

```

    {
        int start = inline.indexOf("<td")+4;
        int end = inline.indexOf("</td", start);
        String worda = inline.substring(start, end);
        start = inline.indexOf("<td", start);
        start = inline.indexOf(">", start);
        end = inline.indexOf("</td", start);
        String wordb = inline.substring(start, end==-1? inline.length()-1: end);
        wordb = wordb.replaceAll("<b>", "");
        wordb = wordb.replaceAll("</b>", "");
        System.out.println(worda+" - "+wordb);
    }
    catch(Exception e)
    {
        System.out.println(inline);
    }
}
}
in.close();

```

Modify the EnglishGermanTranslationPlanB4 having a static dictionary

- Make the variable for the dictionary static and initialize it in a static block instead of in the constructor:

```

static {
    wordtable = new HashMap();
    wordtable.put("coffee", "Kaffee");
    wordtable.put("milk", "Milch");
    wordtable.put("cow", "Kuh");
    wordtable.put("cat", "Katze");
    wordtable.put("dog", "Hund");
}

```

- Provide a public static method for testing if a word is contained in the dictionary:

```

public static boolean containsWord(String name) {
    return wordtable.containsKey(name);
}

```

Modify the copied file TranslationB4.agent.xml to include the new plan

- Add the new online search plan to the plan declarations using a low priority:


```

<plan name="searchonline" priority="-1">
  <body class="SearchTranslationOnlineB4"/>
  <trigger>
    <messageevent ref="request_translation"/>
  </trigger>
</plan>

```

- Add an imports section and the import statement for the jadex.bridge.fipa classes to the imports section:

```

<imports>
  <import>jadex.bridge.fipa.*</import>
</imports>

```

- Modify the applicability of the translation plan by introducing a precondition:

```

<plan name="egtrans">
  <body class="EnglishGermanTranslationPlanB4"/>
  <trigger>
    <messageevent ref="request_translation"/>
  </trigger>
  <precondition>
    EnglishGermanTranslationPlanB4.containsWord((String)$event.getParameter(SFipa.CONTENT) .g
  </precondition>
</plan>

```

Starting and testing the agent\

<p/>

When the agent receives translation request it searches applicable plans to handle this request. If the word is contained in the dictionary both plans are applicable and the one with the higher priority is chosen (in this case it is the egtrans plan because the standard priority is 0). When the word is not contained in the dictionary only the searchonline plan is applicable and will be used.

Exercise B5 - BDI Debugger

The Jadex debugging perspective is conceived to support you in the debugging of agents and helps you to understand what happens inside an agent. You could use it for the agents from the previous excercises, e.g. to grasp the differences between B1 and B2.

<p/>

Using the Jadex debugger tool to control the execution of an agent

- Choose the agent model you want to debug in the starter (e.g. B4) and check the checkbox “start suspended”. You will notice after starting that the agent symbol in the agent tree (at the bottom left) shows the agent in suspended state (zzzz icon). The agent will only process events when the execution is manually requested in the debugger tool. Note that you can also freeze and resume the execution of the translation agent by setting execution mode to “step” and “run” in the tool.
- As an alternative you can prepare the agent debugging by setting the debugging flag in a new properties section (at the end of the file) of the ADF to true.

```
<properties>
  <property name="debugging">true</property>
</properties>
```

- Start the translation agent of the last exercise from the Control Center.
- Switch to the debugger perspective in the Control Center and activate a debugger view for the translation agent by double clicking the agent in the agent tree on the left hand side.
- Use the Conversation Center to send some translation requests to the translation agent (as in B4).
- The information panel of the debugger on the right hand side has two tabs. Use the “agent inspector” tab to view the internal state of a bdi agent and the “rule engine” to see the agent logic in form of rules. The “rule engine” tab also provides a view for the execution history (already executed activations).
- Press the “step” button several times in the debugger and observe in the rule engine tab how an activation (rule) from the agenda is executed.
- The debugger also offers a breakpoint view (on the left). Try out working with breakpoints by selecting some of them and pressing “run”. In case the selected breakpoint (rule) is activated the debugger will automatically set the agent to step mode so that it can be easily inspected and executed step by step.

Exercise B6 - Log-Outputs

In this exercise we will use log-outputs instead of directly printing console outputs. The log outputs will typically be printed to the System.err stream, which is displayed in red color in eclipse. Create the files EnglishGermanTranslationPlanB6.java and TranslationB6.agent.xml by copying the files from exercise B2.

Modify the copied file TranslationPlanB6.java

- Replace all occurrences of System.out.println(..) to getLogger().info(..).

<p/>

Modify the copied file TranslationB3.agent.xml

- Add an imports section and the import statement for the java.logging classes to the imports section:

```
<imports>
  <import>java.util.logging.*</import>
</imports>
```

- Introduce a properties section at the bottom of the ADF to specify the logging behavior. Insert the following code:

```
<properties>
  <property name="logging.level">Level.INFO</property>
  <property name="logging.useParentHandlers">true</property>
</properties>
```

These properties can be used to control the agent logging. The log-level decides what kind of log-outputs shall be considered for logging, according to the java.util.logging level hierarchy. Increasing the level value, e.g. to warning means, that only log-outputs at this or a higher level are considered by the logger. The useParentHandlers property can be used to turn on or off the standard console logging handler (per default it is set to true).

<p/>

Starting and testing the agent\

Start the translation agent. Send a translation request to the translation agent and watch the console and logger output. To turn off the console output simply set the property useParentHandlers in the ADF to false.

END MACRO: html

1 Chapter 4. Using Beliefs

An agent's beliefbase represents its knowledge about the world. The beliefbase is in some way similar to a simple data-storage, that allows the clean communication between different plans by the means of shared beliefs. Contrary to most PRS-style BDI systems, Jadex allows to store arbitrary Java objects as beliefs in its beliefbase. In Jadex between two kinds of beliefs is distinguished. On the one hand there are beliefs that allow the user to store exactly one fact and on the other hand belief sets are supported that allow to store a set of facts. The use of beliefs and belief sets as primary storage capacities for plans is strongly encouraged, because from its usage the user benefits in several ways. If it is necessary to retrieve a cut out of the stored data this is supported by a declarative OQL-like query language. Furthermore, it is possible to monitor single beliefs with respect to their state and cause an event when a corresponding condition is satisfied. This allows to trigger some action when e.g. a fact of a belief set

is added or a belief is modified. It is also possible to wait for some complex expression that relates to several beliefs to become fulfilled.

1.1 Exercise C1 - Beliefs

From this point the copying and renaming of files is not explicitly stated anymore. Furthermore, from now on we use a syntax in the request format that looks like this:

```
*\<action\> \<language(s)\> \<content\>*
```

To translate a word we have to send a request in the form:

```
*translate english_german \<word\>*
```

To add a new word pair to the database we have to send a request in the format:

```
*add english_german \<eword\> \<gword\>*
```

In this first exercise we will use the beliefbase for letting more than one plan having access to the word table by using a belief for storing the word table.

Modify the existing plan to support the request format and introduce a new plan for adding word pairs

- Create a new EnglishGermanAddWordPlanC1 as passive plan, that handles add-new-wordpair requests. In its body method, the plan should check whether the format is correct (using a `~java.util.StringTokenizer~`). If it is ok, it should retrieve the hashtable containing the word pairs via: `~Map words = (Map)getBeliefbase().getBelief("egwords").getFact();~` Assuming that the belief for storing the wordpairs is named "egwords". Now the plan has to check if the English word is already contained in the map (using `~words.containsKey(eword)~`) and if it is not contained, it should be added (using `~words.put(eword, gword)~`).
- Modify the EnglishGermanTranslationPlanC1 so, that it uses the word table stored as single belief in the beliefbase. Additionally the plan has to check the newly introduced request format by using a `~java.util.StringTokenizer~`.
- Add a static `getDictionary()` method to the EnglishGermanTranslationPlanC1. This method should return a hashmap with some wordpairs contained in it. Besides the static method you also need to declare a static variable for storing the dictionary:

```
{code:java}
protected static Map dictionary;
public static Map getDictionary()
{
    if(dictionary==null)
    {
        dictionary = new HashMap();
        dictionary.put("milk", "Milch");
    }
}
```

```

        dictionary.put("cow", "Kuh");
        dictionary.put("cat", "Katze");
        dictionary.put("dog", "Hund");
    }
    return dictionary;
}
{code}

```

Update the ADF to incorporate the new plan and the new belief

The updated version of the translation agent ADF is outlined in the following code snippet. Note that the agent now has two plans named “addword” for adding a word pair to the database and “egtrans” for translating from English to German. The belief declaration is enclosed by a beliefs tag that denotes that an arbitrary number of belief declarations may follow. The ADF defines the beliefs and belief sets of an agent, optionally with default fact(s). The belief for storing the wordtable is named “egwords” and typed through the class attribute to `~java.util.Map~`. The tag of this element is set to `<belief>` (in contrast to `<beliefset>`) denoting that only one fact can be stored. Further it is necessary to clarify which kinds of events trigger the plans. Therefore, the events section is extended to include a new `~request_addword~` event type which also matches request messages. To be able to distinguish between both kinds of events they are refined to match only messages that start with a specific content string. The match expressions use a logical AND (`&\&`), that has to be written a little bit awkwardly with the xml entities `\&\&`.

```

{code:xml}
<agent xmlns="http://jadex.sourceforge.net/jadex "(http://jadex.sourceforge.net/jadex)"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance "(http://www.w3.org/2001/XMLSchema-
instance)"
    xsi:schemaLocation="http://jadex.sourceforge.net/jadex "(http://jadex.sourceforge.net/jadex)
        http://jadex.sourceforge.net/jadex-bdi-2.3.xsd "(http://jadex.sourceforge.net/jadex-
bdi-2.3.xsd)"
    name="TranslationC1"
    package="jadex.bdi.tutorial">
    <imports>
        <import>java.util.logging.*</import>
        <import>java.util.*</import>
        <import>jadex.bridge.fipa.*</import>
    </imports>
    <beliefs>
        <belief name="egwords" class="Map">
            <fact>EnglishGermanTranslationPlanC1.getDictionary()</fact>
        </belief>
    </beliefs>

```

```

<plans>
  <plan name="addword">
    <body class="EnglishGermanAddWordPlanC1"/>
    <trigger>
      <messageevent ref="request_addword"/>
    </trigger>
  </plan>

  <plan name="egtrans">
    <body class="EnglishGermanTranslationPlanC1"/>
    <trigger>
      <messageevent ref="request_translation"/>
    </trigger>
  </plan>
</plans>

<events>
  <messageevent name="request_addword" direction="receive" type="fipa">
    <parameter name="performative" class="String" direction="fixed">
      <value>SFipa.REQUEST</value>
    </parameter>
    <match>$content instanceof String &&& ((String)$content).startsWith("add
english_german")</match>
  </messageevent>

  <messageevent name="request_translation" direction="receive" type="fipa">
    <parameter name="performative" class="String" direction="fixed">
      <value>SFipa.REQUEST</value>
    </parameter>
    <match>$content instanceof String &&& ((String)$content).startsWith("translate
english_german")</match>
  </messageevent>
</events>
</agent>
{code}

```

Start and test the agent

Send several add-word and translation requests to the agent and observe, if it behaves well. In this example the belief is already created when the agent is initialized.

1.1 Exercise C2 - Beliefsets

Using a belief set for storing the word-pairs and employing beliefbase queries to look-up a word in the word table belief set. In this example each word pair is saved in a data structure called `~jadex.commonstuple` which is a list of entities similar to an object array. In contrast to an object array two tuples are

considered to be equal when they contain the same objects. *Of course, in belief sets arbitrary Java objects can be stored, not just Tuples.*

Modify the plans

- Modify the EnglishGermanTranslationPlanC2 so, that it uses a query to search the requested word in the belief set. Therefore use an expression defined in the ADF: `~this.queryword = getExpression("query_egword");~` (Assuming that the `~jadex.bdi.runtime.IExpression~ queryword` is declared as instance variable in the plan). To apply the query insert the following code at the corresponding place inside the plan's body method: `~String gword = (String)queryword.execute("$eword", eword);~`
- Modify the EnglishGermanAddWordPlanC2 so, that it also uses the same query to find out, if a word pair is already contained in the belief set. Apply the query before inserting a new word pair. When the word pair is already contained log some warning message. To add a new fact to an existing belief set you can use the method: `~getBeliefbase().getBeliefSet("egwords").addFact(new jadex.commons.Tuple(eword, gword));~`

Modify the ADF

- For checking if a word pair is contained in the wordtable and for retrieving a wordpair from the wordtable a query expression will be used. Insert the following code into the ADF below the events section:

```
{code:xml}
<expression name="query_egword">
  select one $wordpair.get(1)
  from Tuple $wordpair in $beliefbase.getBeliefSet("egwords").getFacts()
  where $wordpair.get(0).equals($eword)
</expression>
{code}
```

- We don't cover the details of the query construction in this tutorial. If you are interested in understanding the details of the Jadex OQL query language, please consult the [BDI User Guide>BDI User Guide.01 Introduction].
- Modify the ADF by defining a belief set for the wordtable. Therefore change the tag type from "belief" to "belief set" and the class from "Map" to "Tuple". Note that Tuple is a helper class that is located in `jadex.commons` and has to be added to the imports section if you don't specify the fully-qualified classname. Remove the old Map fact declaration and put in four new facts each surrounded by the fact tag. Put in the same values as before `~new Tuple("milk", "Milch")~` etc. for each fact.

Start and test the agent

Send several add-word and translation requests to the agent and observe, if it behaves well. Verify that it behaves exactly like the agent we built in exercise C1. This exercise does not functionally modify our agent.

1.1 Exercise C3 - Belief Conditions

In this exercise we will use a condition for triggering a passive plan that congratulates every 10th user.

Create and modify plans

- Create a new passive ThankYouPlanC3 that prints out a congratulation message and the actual number of processed requests. The number of processed requests will be stored in a belief called “transcnt” in the ADF. Retrieve the actual request number by getting the fact from the beliefbase with: `~int cnt = ((Integer)getBeliefbase().getBelief(“transcnt”).getFact()).intValue();~`
- Modify the EnglishGermanTranslationPlanC3 to count the translation requests: `~int cnt = ((Integer)getBeliefbase().getBelief(“transcnt”).getFact()).intValue(); getBeliefbase().getBelief(“transcnt”).setFact(new Integer(cnt+1));~`

Modify the ADF

- Modify the ADF by defining the new ThankYouPlanC3 as passive plan (with the name `thankyou` in the ADF) in the plans section. Instead of defining a triggering event for this passive plan we define a condition that activates the new ThankYouPlanC3. A condition has the purpose to monitor some state of affair of the agent. In this case we want to monitor the belief “transcnt” and get notified whenever 10 translations have been requested. Insert the code from the following snippet in the plan’s trigger. This condition consists of two parts: This first `transcnt>0` makes sure that at least one translation has been done and the second part checks if `transcnt modulo 10` has no rest indicating that $10 \cdot x$ translations have been requested. The two parts are connected via a logical AND (`&&`), that has to be written a little bit awkwardly with the xml entities `\&\&`.

```
{code:xml}
<condition>$beliefbase.transcnt>0 &amp;&amp; $beliefbase.transcnt%10==0</condition>
{code}
```

- Define and initialize the new belief in the ADF by introducing the following lines in the beliefs section:

```
{code:xml}
<belief name=“transcnt” class=“int”>
  <fact>0</fact>
</belief>
{code}
```

Start and test the agent

Send some translation requests and observe if every 10th time the congratulation plan is invoked and prints out its message.

1.1 Exercise C4 - Agent Arguments

In this exercise we will use agent arguments for the custom initialization of an agent instance.

- Use the translation agent C3 as starting point and specify an agent argument in the ADF. Arguments are beliefs for which a value can be supplied from outside during the agent start-up. For declaring a belief being an agent argument simply mark it as argument by using the corresponding belief attribute `~argument="true"~`. In this case we want the belief “transcnt” being the argument. Note that only beliefs not belief sets can be used as arguments.
- Use the Starter to create instances of the new agent model. The Starter automatically displays textfields for all agent arguments and also shows the default model value (if any) that will be used when the user does not supply a value. Try entering different values into the textfield: What happens if you enter e.g. a string instead of the integer value that is needed here?
- Start the agent with different argument values. Verify, that the agent immediately invokes the congratulation plan if the initial number of translation requests is e.g. 10.

1.1 Exercise C5 - Observing Agent State

In this exercise we will use the Jadex BDI introspector tool agent to view the beliefs of the agent.

Start the translation agent from the last exercise. Before sending requests to the translation agent start the Jadex BDI debugger and open the “agent inspector” tab.

- Use the Conversation Center to send translation or add-word requests to the translation agent.
- Observe the belief change of the translation count, whenever a translation request is processed.
- Observe the changes of the word pair belief set, whenever an add-word request is processed.
- Use the example from C1 to see the difference in the representation of the word table as belief and belief set.

1 Chapter 5. Using Capabilities

Different agents often need to use the same or similar functionalities that incorporate more than just plan behavior. Often private or shared beliefs and goals are part of a common functionality of one agent. These units of functionality are comparable to the module concept in the object oriented paradigm, but exhibit very different properties because of the use of mentalistic notions. For this reasons the capability concept was originally introduced [Busetta et al. 2000] and enhanced in [Braubach et al. 2005b] that allows for packaging a subset of beliefs plans and goals into an agent module and reuse this module wherever needed. The capability structure of an agent forms a tree. A superordinated (parent) capability may contain an arbitrary number of subcapabilities. All elements of a capability have per default private visibility and need to be explicitly made available for usage in a connected capability. For this purpose elements can be defined as abstract or exported enabling access from another capability.

1.1 Preparation

We use the functionality of the C2 Agent and build up a capability of its plans and beliefs. Therefore, it is necessary to copy and rename all files from C2 to D1. We slightly modify these plans to make the translation agent answer to a request with a reply message. Hence the following has to be done in both plans:

- Declare two variables at the beginning of the plans:

```
{code:java}
String reply; The message event type of the reply.
String content; The content of the reply message event.
{code}
```

- Set both variables with respect to the success of the translation. In the success case set (assuming that gword and eword are variables for the English and German word respectively):

```
{code:java}
reply = "inform";
content = gword;
{code}
```

- And in the failure case:

```
{code:java}
reply = "failure";
content = "Sorry, word could not be translated."+eword;
{code}
```

- Send an answer to the caller at the end of the event processing:


```
{code:java}
IMessageEvent replymsg = getEventbase().createReply((IMessageEvent)getReason(),
reply);
replymsg.getParameter(SFipa.CONTENT).setValue(content);
sendMessage(replymsg);
{code}
```
- Add the new message event types “inform” and “failure” to the ADF:

```
{code:xml}
<events>
...
<messageevent name="inform" direction="send" type="fipa">
  <parameter name="performative" class="String" direction="fixed">
    <value>SFipa.INFORM</value>
  </parameter>
</messageevent>
<messageevent name="failure" direction="send" type="fipa">
  <parameter name="performative" class="String" direction="fixed">
    <value>SFipa.FAILURE</value>
  </parameter>
</messageevent>
</events>
{code}
```

- Test the agent and verify that it answers to the request messages by sending an answer message (for correct as well as for incorrect requests).

1.1 Exercise D1 - Creating a Capability

In this exercise we will create a translation capability.

Create a new Capability ADF

- Create a new file TranslationD1.capability.xml with the skeleton code from the following code snippet. Now copy the definition of imports, plans, beliefs, events (including the newly defined ones from above) and expressions (in this case there are no goals) from TranslationC2.agent.xml into this file.

```
{code:xml}
<capability xmlns="http://jadex.sourceforge.net/jadex-bdi"(http://jadex.sourceforge.net/jadex-
bdi)"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance
```

```

](http://www.w3.org/2001/XMLSchema-instance)"
    xsi:schemaLocation="http://jadex.sourceforge.net/jadex-bdi
](http://jadex.sourceforge.net/jadex-bdi)
    http://jadex.sourceforge.net/jadex-bdi-2.0.xsd
](http://jadex.sourceforge.net/jadex-bdi-2.0.xsd)"
    name="TranslationD1"
    package="jadex.bdi.tutorial">
...
</capability>
{code}

```

- Modify the agent ADF (TranslationD1.agent.xml) by removing all plan and belief definitions. Instead insert a new section for using the new capability.

```

{code:xml}
<capabilities>
  <capability name="transcap" file="TranslationD1"/>
</capabilities>
{code}

```

- Note that here the type name is employed, but absolute and relative paths to (the model name of) the XML file can also be used.

Start and test the agent

Load the agent model in the RMA and start the agent. Test the agent with add word and translate requests. It should behave exactly like the Agent from C2. Use the debugger agent to view the new internal structure of the agent.

1.1 Exercise D2 - Exported Beliefs

In this exercise we will extend the translation agent by making it capable to find synonyms for English words. Therefore we extend the agent from D1 with a new find synonyms plan which will directly be contained in the agent description. Because the plan needs to access the dictionary from the translation capability, the egwords belief will be made usable from external.

Create a new plan

- Create the file FindEnglishSynonymsPlanD2.java as a passive plan which reacts on messages with performative type request and starts with "find_synonyms english". Therefore, you need to introduce the new message event type "find_synonyms" that fits for request messages that start with "find_synonyms english".
- Create one query (called query_translate here) in the constructor for translating an English word (the query expression can be copied from

the EnglishGermanTranslationPlanD2). Create another query (called query_find here) with the purpose to find all English words that match exactly a German word and are unequal to the given English word.

```
{code:java}
String find = "select $wordpair.get(0)"
+ "from Tuple $wordpair in $beliefbase.egwords"
+ "where $wordpair.get(1).equals($gword) && !$wordpair.get(0).equals($eword)";
this.queryfind = createExpression(find, new String[]{"$gword", "$eword"},
new Class[]{String.class, String.class});
{code}
```

- In the body method, search for synonyms when the message format is correct, what means that the request has exactly three tokens. Use a `~StringTokenizer~` to parse the request and apply the translation query on the the given English word. When a translation was found, use the result to apply the query find for searching for synonyms. Create a reply and send back the found synonyms as an inform message in the success case and a failure message with a failure reason in the error case. The following code snippet outlines how the second query can be realized (eword is the English word for which synonyms are searched, gword is the German translation of the given English word):

```
{code:java}
List syms = (List)queryfind.execute(new String[]{"$gword", "$eword"}, new
Object[]{gword, eword});
{code}
```

Create a new Capability ADF

- Create a new file TranslationD2.capability.xml by copying the capability from exercise D1.
- Modify the belief set declaration of "egwords" by setting the belief set `~type="exported"~`. Add some facts to the belief "egtrans" to have some synonyms present.

```
{code:xml}
<beliefset name="egwords" class="Tuple" exported="true">
  <fact>new Tuple("milk", "Milch")</fact>
  <fact>new Tuple("cow", "Kuh")</fact>
  <fact>new Tuple("cat", "Katze")</fact>
  <fact>new Tuple("dog", "Hund")</fact>
  <fact>new Tuple("puppy", "Hund")</fact>
  <fact>new Tuple("hound", "Hund")</fact>
  <fact>new Tuple("jack", "Katze")</fact>
  <fact>new Tuple("crummie", "Kuh")</fact>
```

```
</beliefset>
{code}
```

- Also use the exported attribute to make the “inform” and “failure” messages accessible, as we want to use these in the new synonyms plan.

Create a new TranslationD2 Agent ADF

- Create a new file TranslationD2.agent.xml by copying the file from D1. Extend this definition by adding the new plan to the plans section and adding a referenced belief, that relates to the egwords belief from the capability. Note that the name of the referenced belief can be chosen arbitrarily (in this case we name it egwords, too). Additionally change the capability reference to the newly created TranslationCapabilityD2 and add the new message event type request_findsynonyms.

```
{code:xml}
<beliefs>
  <beliefsetref name="egwords">
    <concrete ref="transcap.egwords" />
  </beliefsetref>
</beliefs>

<plans>
  <plan name="find_synonyms">
    <body class="FindEnglishSynonymsPlanD2"/>
    <trigger>
      <messageevent ref="request_findsynonyms"/>
    </trigger>
  </plan>
</plans>

<events>
  <messageevent name="request_findsynonyms" direction="receive"
type="fipa">
  <parameter name="performative" class="String" direction="fixed">
    <value>SFipa.REQUEST</value>
  </parameter>
  <parameter name="content-start" class="String" direction="fixed">
    <value>"find_synonyms english"</value>
  </parameter>
</messageevent>

  <messageeventref name="inform">
    <concrete ref="transcap.inform"/>
  </messageeventref>
```

```

    <messageeventref name="failure">
      <concrete ref="transcap.failure"/>
    </messageeventref>
  </events>
  {code}

```

Start and test the agent

Start the agent and send it some find synonyms requests, e.g. “find_synonyms english dog”. When your agent works ok, you should be notified that the synonyms for dog are hound and puppy. Use the bdi debugger to understand what the belief set reference means.

1 Chapter 6. Using Goals

Goal-oriented programming is one of the key concepts in the agent-oriented paradigm. It denotes the fact that an agent commits itself to a certain objective and maybe tries all the possibilities to achieve its goal. A good example for a goal that ultimately has to be achieved is the safe landing of an aircraft. The agent will try all its plans until this goal has succeeded, otherwise it will not have the opportunity to reach any other goal when the aircraft crashes. When talking about goals one can consider different kinds of goals. What we discussed above is called an *~achieve goal~*, because the agent wants to achieve a certain state of affairs. Similar to an achieve goal is the *~query goal~* which aims at information retrieval. To find the requested information plans are only executed when necessary. E.g. a cleaner agent could use a query goal to find out where the nearest wastebin is. Another kind is represented through a maintain goal, that has to keep the properties (its maintain condition) satisfied all the time. When the condition is not satisfied any longer, plans are invoked to re-establish a normal state. An example for a maintain goal is to keep the temperature of a nuclear reactor below some specified limit. When this limit is exceeded, the agent has to act and normalize the state. The fourth kind of goal is the perform goal, which is directly related to some kind of action one wants the agent to perform. An example for a perform goal is an agent that as to patrol at some kind of frontier.

1.1 Exercise E1 - Subgoals In this exercise we will use a subgoal for translating words. Extend the translation agent C2 to have a second translation plan for translations from English to French. Introduce a `ProcessTranslationRequestPlanE1` that receives all incoming translation requests and uses a subtask triggered by an achieve goal to perform the translation.

Remove, create and modify plans

- Remove the `EnglishGermanAddWordPlan` to keep the agent simple.
- Create a new initial `ProcessTranslationRequestPlanE1` that reacts on all incoming messages with performative type request and creates subgoals for all (correctly formatted) requests. Because we are using a service plan implement the body method with an infinite loop and start waiting for

a message to process. Assuming that the plan has extracted the action (translate), the language direction (english_german or english_french) and the word(s) from an incoming message the following code can be used to create, dispatch and wait for a subgoal:

```

IGoal goal = createGoal("translate");
goal.getParameter("direction").setValue(dir);
goal.getParameter("word").setValue(word);
try
{
    dispatchSubgoalAndWait(goal);
    getLogger().info("Translated from "+goal+" "+
word+" - "+goal.getParameter("result").getValue());
}
catch(GoalFailureException e)
{
    getLogger().info("Word is not in database: "+word);
};

```

- After the goal returns successfully, read the result from the goal and log some translation message.
- Modify the EnglishGermanTranslationPlanE1 so that it can handle a translation goal with direction="english_german". Therefore, the body method has to be adapted so that it extracts the word from the plan parameter mapping (using `getParameter("word").getValue()`). After having performed the query on the wordtable, set the result using `getParameter("result").setValue(gword)`. When no translation could be retrieved, the plan has failed and this should be indicated calling the `fail()` method (which throws a plan failure exception).
- Create a new EnglishFrenchTranslationPlanE1 as a copy of the EnglishGermanTranslationPlanE1 and make sure to work on a new wordtable belief efwords. Modify the `query_word` and the body method accordingly.

Modify the ADF

- Add the ProcessTranslationRequestPlanE1 to the ADF as initial plan with a waitqueue for translation requests. Add an configurations section and declare a configuration with an initial plan for the ProcessTranslationRequestPlanE1.
- Adapt the plan head declarations of both plans to include plan parameters and the new triggers. The plan parameters are directly mapped to the corresponding goal parameters so that the input as well as the result are automatically transferred from resp. to the goal. In addition, both translation plans should handle exactly suitable translation goals. In the following the modified plan head for the EnglishGermanTranslationPlanE1 is depicted:


```

<plan name="egtrans">
  <parameter name="word" class="String">
    <goalmapping ref="translate.word"/>
  </parameter>
  <parameter name="result" class="String" direction="out">
    <goalmapping ref="translate.result"/>
  </parameter>
  <body class="EnglishGermanTranslationPlanE1"/>
  <trigger>
    <goal ref="translate">
      <match>"english_german".equals($goal.getParameter("direction").getValue())</mat
    </goal>
  </trigger>
</plan>

```

- Introduce a new goals section and declare the achieve goal for translations:

```
{code:xml}          {code}
```

- Modify the ADF by adjusting the plan declarations to include the new EnglishFrenchTranslationPlanE1 and exclude the add word plan. Additionally a new belief efwords has to be declared in the beliefs section:

```
{code:xml}  new Tuple("milk", "lait")  new Tuple("cow", "vache")  new
Tuple("cat", "chat")  new Tuple("dog", "chien") {code}
```

- Introduce a second query for the new belief efword in the expressions section of the ADF:

```
{code:xml}  select one $wordpair.get(1)  from Tuple $wordpair in $belief-
base.efwords  where wordpair.get(0).equals(efword) {code}
```

Start and test the agent

Start the agent and supply it with some translation requests. Observe which plans are activated in what sequence and how the goal processing is done. Change the translation direction in the requests and check if the right plan is invoked.

1.1 Exercise E2 - Retrying a Goal

Using the BDI-retry mechanism for trying out different plans for one goal. This can be useful, for example if there are several plans for one specific goal, but all plans work under different circumstances. With the retry mechanism, all plans will be tried until one plan lets the goal succeed or any plan has been tried.

Modify the following

- Modify the trigger of both translating plans so, that they react on every translation goal by removing the lines:

```
{code:xml} “english_german/english_french”.equals($goal.getParameter(“direction”).getValue())  
{code}
```

- Having done this causes the translation plans to react on every translation goal, even when they can’t handle the translation direction or language.
- Introduce a new plan parameter for the translation direction and supply it with a corresponding goal mapping:

```
{code:xml} {code}
```

- Modify the translation plans so that they check the translation direction in the body method before translating. When the direction cannot be handled, they should indicate that they failed to achieve the goal by calling the `fail()` method. Additionally the plans should log or print some warning message, when they fail to process a goal:

```
{code:java}  
if(“english_french”.equals(getParameter(“direction”).getValue())) print out some message and fail() if this is not the english-french plan {code}
```
- You need not explicitly set the BDI-retry flag of the goal, because in the standard configuration for goals all BDI-mechanisms (retry, exclude and meta-level reasoning) are enabled. This means, that a failed goal will be retried by different plan candidates until it succeeds or all possible candidates have failed to handle the goal and it is finally failed. *Start and test the agent*

Provide the agent with some translation work and watch out how the goal processing is done this time. Observe by changing the translation direction of the request how different plans are scheduled to handle a goal. 1.1 Exercise E3 - Maintain Goals

Using a maintain goal to keep the number of wordtable entries below a specified maximum value. For this exercise we will use the TranslationAgentD2 as starting point.

Create a new file RemoveWordPlanE3.java

- This plan has the purpose to delete an entry from the wordtable.
- In the body method use the following code to delete one entry from the set:

```
{code:java} Object[] facts = getBeliefbase().getBeliefSet(“egwords”).getFacts();  
getBeliefbase().getBeliefSet(“egwords”).removeFact(facts[0]); {code} Create the TranslationE3.agent.xml as copy from the TranslationD2.agent.xml
```

- Remove the FindSynonymsPlanD2 from the plans section and remove the event section completely.
- Add the RemoveWordPlanE3 to the plans section:

```
{code:xml}
```

```
$beliefbase.egwords.length > 0 {code}
```

- Add the following beliefs to the belief section:

```
{code:xml}      8 {code}
```

- Add a new maintain goal declaration to the goals section:

```
{code:xml}      $beliefbase.egwords.length <= $beliefbase.maxstorage  
{code}
```

- Create a configurations section with one configuration that creates a maintain goal instance on startup.

```
{code:xml}      {code}
```

Start and test the agent

Start the translation agent and add new wordpairs to the dictionary. The maintain condition is violated and the goal should be activated. This leads to a subsequent removal of entries in the belief set, until the condition holds again.

1 Chapter 7. Using Events

Communication takes place at two different abstraction levels in Jadex. The so called intra-agent communication is necessary when two or more plans inside of an agent want to exchange information. They can utilize several techniques to achieve this. The encouraged possibility is to use beliefs (and conditions). Beliefs in Jadex are containers for normal Java objects, but they are a specially designed concept for agent modelling and therefore using beliefs has several advantages. One advantage is that they allow the usage of conditions to trigger events depending on belief states (e.g. creating a new goal when a new fact is added to a belief set). A further advantage is that using the beliefbase one is able to formulate queries and retrieve only entities that correspond to the query-expression. Another possibility of internal communication is to use explicit internal events. In contrast to goals, events are (per default) dispatched to all interested plans but do not support any BDI-mechanisms (e.g. retry). Therefore the originator of an internal event is usually not interested in the effect the internal event may produce but only wants to inform some interested parties about some (important) occurrence.

On the other hand inter-agent communication describes the act of information exchange between two or more different agents. The inter-agent information exchange in Jadex is based on asynchronous message event passing. Each message event in Jadex has a dedicated `~jadex.bridge.MessageType~` which constrains the allowed parameters and the parameter types of the message event. Currently, only the [FIPA message type><http://fipa.org/specs/fipa00061/SC00061G.html>] (<http://fipa.org/specs/fipa00061/SC00061G.html>) is supported. It equips a message type with all possible FIPA parameters such as sender, receivers, performative, content, etc. Besides the underlying message type (which is normally not of very much importance for agent programmers) in the ADF user defined message event types are specified, such as the `request_translation` message event we already encountered in earlier exercises. Note that the message event types are only locally visible and each agent uses its own message event

types for sending and receiving messages. Hence, when an agent receives a message it has to decide which local message event type will be used to represent this message. The details of this process will be outlined in one of the following exercises. In this tutorial we will only show how a basic communication between two agents is implemented, when one agent offers a service that the other one seeks. The supplier therefore has to register its services by the Directory Facilitator (DF) and is further on available as service provider. Another agent seeks a service by asking the DF and receives the providers address which it subsequently uses for the direct communication with the provider.

1.1 Exercise F1 - Internal Events

In this exercise we will use internal events to broadcast information. We extend the simple translation agent from exercise C2 with a plan that shows the processed requests in a gui triggered by an internal event.

Create a new GUI class named TranslationGuiF1.java as an extension of a JFrame

- This class has the purpose to show the already performed actions in a table. As member variable the table model is needed to be able to refresh the data in response to update notifications.

```
{code:java}
protected DefaultTableModel tadata;
{code}
```

- In the constructor the table and its model should be created and added to the frame:

```
{code:java}
tadata = new DefaultTableModel(new String[]{"Action", "Language", "Content",
"Translation"}, 0);
JTable tatable = new JTable(tadata);
JScrollPane sp = new JScrollPane(tatable);
this.getContentPane().add("Center", sp);
this.pack();
this.setLocation(SGUI.calculateMiddlePosition(this));
this.setVisible(true);
{code}
```

- Finally, for updating the gui a method is needed:

```
{code:java}
public void addRow(final String[] content)
{
    SwingUtilities.invokeLater(new Runnable()
    {
```

```

    public void run()
    {
        tadata.addRow(content);
    }
    });
}
{code}

```

Modify the plans

- Create a new GUIPlanF1 plan that has the purpose to create the gui and update it accordingly. The plan should create the gui in its constructor. In its body method it should wait in an endless loop for internal events of type gui_update:

```

{code:java}
InternalEvent event = waitForInternalEvent("gui_update");
{code}

```

- Whenever such an event occurs the plan has to invoke the addRow() method of the gui whereby the update information is contained in a parameter named content

```

{code:java}
event.getParameter("content").getValue();
{code}

```

- In addition the plan's aborted method can be used to close the gui automatically when the agent is terminated:

```

{code:java}
public void aborted(){
    SwingUtilities.invokeLater(new Runnable()
    {
        public void run()
        {
            gui.dispose();
        }
    });
}
{code}

```

- Modify the EnglishGermanTranslationPlanF1 so that it produces an internal event after translation processing:

```

{code:java}
InternalEvent event = createInternalEvent("gui_update");
event.getParameter("content").setValue(new String[]{action, dir, eword,
gword});
dispatchInternalEvent(event);

```

```
{code}
```

Modify the ADF

- The addword plan and event declarations are not used and can be removed for clarity.
- Modify the ADF so, that it contains the declaration for the new gui plan without specifying a trigger:

```
{code:xml}  
<plan name="gui">  
  <body class="GUIPlanF1"/>  
</plan>  
{code}
```

Introduce the declaration of the new gui_update event within the events section:

```
{code:xml}  
<internalevent name="gui_update">  
  <parameter name="content" class="String[]"/>  
</internalevent>  
{code}
```

Add an configurations section with one configuration that creates an initial gui plan:

```
{code:xml}  
<configurations>  
  <configuration name="default">  
    <plans>  
      <initialplan ref="gui"/>  
    </plans>  
  </configuration>  
</configurations>  
{code}
```

Start and test the agent

Start the agent and send several translation requests to the agent. Observe if the gui displays all the translation requests.

1.1 Exercise F2 - Receiving Messages

This exercise will explain how the mapping of received messages to the agent's message events works. Whenever an agent receives a message it has to decide which message event will internally be used for representing the message. This mapping is very important because any agent behavior such as e.g. plan triggers may only depend on the interpreted message event type. In general the event mapping works automatically and an agent designer does not have to worry about the mappings. Nevertheless, there are situations in which more than

one mapping from a received message to different message events are available (normally this is undesirable and should be avoided by using more specific message event declarations). In such situations the agent rates the alternatives by specificity that is simply estimated by the number of parameters used for the declaration and chooses the one with the highest specificity. If more than one alternative has the same specificity the first one is chosen, although this case indicates an implementation flaw and might lead to undesired behavior when the wrong mapping is chosen. In any case the developer is informed with a logging message whenever more than one mapping was found by the agent.

As starting point for this exercise we take agent B2, which only has one passive translation plan which reacts on request messages. Other kinds of messages are simply ignored by the agent. To improve this situation and let the agent answer on all incoming messages we use the ready to use `NotUnderstoodPlan` from the Jadex plan library. Additionally, instead of the original B2 translation plan we take the enhanced one from section 5.1 which sends back inform/failure messages to the requesting agent.

Modify the copied file `TranslationF2.agent.xml` to include the new not-understood plan

- Add the following to the imports section:

```
{code:xml}
<imports>
  <import>jadex.bdi.planlib.*</import>
  <import>jadex.base.fipa.*</import>
  <import>java.util.logging.*</import>
</imports>
{code}
```

- Add the new not-understood plan to the plan declarations:

```
{code:xml}
<plan name="notunderstood">
  <body class="NotUnderstoodPlan"/>
  <trigger>
    <messageevent ref="any_message"/>
  </trigger>
</plan>
{code}
```

- Add the new `any_message` event which matches all kinds of messages and the not understood message that will be sent by the `NotUnderstoodPlan` to the event declarations:

```
{code:xml}
<messageevent name="any_message" direction="receive" type="fipa"/>
<messageevent name="not_understood" direction="send" type="fipa">
  <parameter name="performative" class="String" direction="fixed">
```

```

    <value>SFipa.NOT_UNDERSTOOD</value>
  </parameter>
</messageevent>
{code}

```

- Besides the any_message for receiving arbitrary kinds of messages and the not_understood message which will be send by the not understood plan, we also need message declarations for the other messages to be sent be our agent. Here we need inform and failure messages that will be used by the modified translation plan:

```

{code:xml}
<messageevent name="inform" direction="send" type="fipa">
  <parameter name="performative" class="String" direction="fixed">
    <value>SFipa.INFORM</value>
  </parameter>
</messageevent>
<messageevent name="failure" direction="send" type="fipa">
  <parameter name="performative" class="String" direction="fixed">
    <value>SFipa.FAILURE</value>
  </parameter>
</messageevent>
{code}

```

Start and test the agent

The added plan provides the agent with the ability to react on arbitrary messages. When the agent receives a message with performative request, both message events match and the request_translation event is chosen due to its higher specificity. Other messages are directly mapped to the any_message event type and hence, the agent will repond with a not understood message. Send the agent different messages and observe if it invokes the right plans.

1.1 Exercise F3 - Service Publication

We make the services of our translation agent publicly available by registering its service description at the Directory Facilitator (DF).

Use the translation agent D1 as starting point and extend its copied ADF by performing the following steps

- Include the DF capability in the ADF to be used:

```

{code:xml}
<capabilities>
  <capability name="dfcap" file="jadex.bdi.planlib.df.DF"/>
  <capability name="transcap" file="jadex.bdi.tutorial.TranslationD1"/>
</capabilities>
{code}

```


- Create a reference for the df_keep_registered goal to make it locally available:

```
{code:xml}
<goals>
  <maintaingoalref name="df_keep_registered">
    <concrete ref="dfcap.df_keep_registered"/>
  </maintaingoalref>
</goals>
{code}
```

- Create a configurations section with one configuration. In this configuration an initial goal for the df registration should be provided. The agent description that is used for the registration is provided as initial value of the "description" parameter of the df_keep_registered goal:

```
{code:xml}
<configurations>
  <configuration name="default">
    <goals>
      <initialgoal ref="df_keep_registered">
        <parameter ref="description">
          <value>
            ((IDF)$scope.getServiceContainer().getService(IDF.class))
              .createDFComponentDescription(null, ((IDF)$scope.getServiceContainer().getService(IDF.class))
                .createDFServiceDescription("service_translate", "translate en-
english_german", "University of Hamburg"))
          </value>
        </parameter>
        <parameter ref="leasetime">
          <value>20000</value>
        </parameter>
      </initialgoal>
    </goals>
  </configuration>
</configurations>
{code}
```

Start and test the agent

Start the agent and open the DF GUI. Observe if an entry for the agent exists. Use the DF GUI to deregister your agent (via popup-menu) and observe what happens after a while. Note that when you want to register the agent at a remote df, you only need to slightly modify your initial goal description by adding parameter values for the DF IComponentIdentifier and address.

1.1 Exercise F4 - A Multi-Agent Scenario

As a little highlight we now extend our scenario from F3 to become a real multi-agent system. The conceptual scenario is depicted in the following figure. The user wishes to translate a sentence and sends its requests via the conversation center to the user agent. The user agent searches for a translation service at the DF and subsequently sends for each word from the sentence, a translation request to the translation agent. The user agent collects the translated words and sends back the translated sentence to the message center, where it is visible for the user.

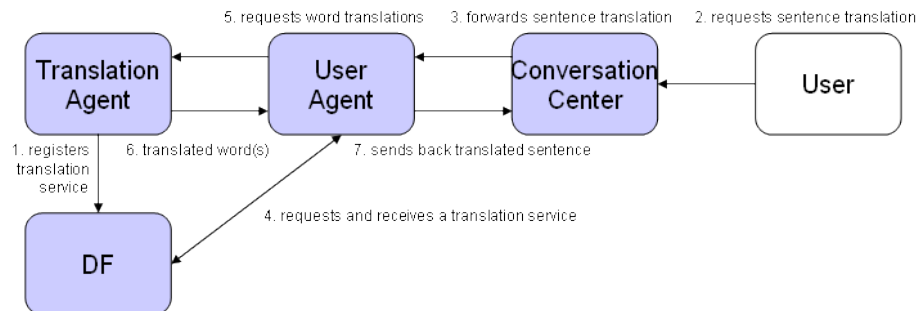


Figure 46:

~Figure 1 The multi-agent scenario~

Create a new UserAgentF4.agent.xml

- Create a new agent called UserAgent by creating an ADF and one plan called EnglishGermanTranslateSentencePlanF4. In the ADF define the translate sentence plan with an appropriate waitqueue that handles message events of the new type request_translatesentence. The new request_translatesentence message event should be declared to match request messages that start with “translate_sentence english_german”. Additionally incorporate the DF and Protocols capabilities in the capabilities section and create references for the rp_initiate (request protocol initiate) and df_search goals. This agent uses the df search goal to find a translation agent and the request goal to communicate in a similar standard way with the translation agent.

```

{code:xml}
<agent xmlns="http://jadex.sourceforge.net/jadex-bdi "(http://jadex.sourceforge.net/jadex-
bdi)"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance "(http://www.w3.org/2001/XMLSchema-
instance)"
  xsi:schemaLocation="http://jadex.sourceforge.net/jadex-bdi "(http://jadex.sourceforge.net/jadex-
bdi)
                                http://jadex.sourceforge.net/jadex-bdi-2.0.xsd "(http://jadex.sourceforge.net/jadex-
bdi-2.0.xsd)"

```

```

name="UserF4"
package="jadex.bdi.tutorial">

<imports>
  <import>jadex.planlib.*</import>
  <import>jadex.base.fipa.*</import>
  <import>java.util.logging.*</import>
</imports>

<capabilities>
  <capability name="procap" file="jadex.bdi.planlib.protocols.request.Request"/>
  <capability name="dfcap" file="jadex.bdi.planlib.df.DF"/>
</capabilities>

<goals>
  <achievegoalref name="rp_initiate">
    <concrete ref="procap.rp_initiate"/>
  </achievegoalref>

  <achievegoalref name="df_search">
    <concrete ref="dfcap.df_search"/>
  </achievegoalref>
</goals>

<plans>
  <plan name="egtrans">
    <body class="EnglishGermanTranslateSentencePlanF4"/>
    <waitqueue>
      <messageevent ref="request_translatesentence"/>
    </waitqueue>
  </plan>
</plans>

<events>
  <messageevent name="request_translatesentence" direction="receive"
type="fipa">
  <parameter name="performative" class="String" direction="fixed">
    <value>SFipa.REQUEST</value>
  </parameter>
  <match>$content instanceof String &amp;&amp; ((String)$content).startsWith("translate_sentence
english_german")</match>
  </messageevent>

  <messageevent name="inform" direction="send" type="fipa">
    <parameter name="performative" class="String" direction="fixed">
      <value>SFipa.INFORM</value>
    </parameter>
  </messageevent>

```

```

<messageevent name="failure" direction="send" type="fipa">
  <parameter name="performative" class="String" direction="fixed">
    <value>SFipa.FAILURE</value>
  </parameter>
</messageevent>
</events>

<configurations>
  <configuration name="default">
    <plans>
      <initialplan ref="egtrans"/>
    </plans>
  </configuration>
</configurations>
</agent>
{code}

```

- The body method of this plan should adhere to the following basic structure:

```

{code:java}
public void body()
{
  protected IComponentIdentifier ta;
  ...

  while(true)
  {
    Read the user request.
    IMessageEvent mevent = (IMessageEvent) waitForMessageEvent("request_translatesentence");

    Save the words of the sentence.
    Process the message event here.
    ...
    String[] words = ...;
    String[] twords = ...;
    ...

    Search a translation agent.
    while(ta==null) ta is the instance variable for the translation agent
    {
      Create a service description to search for.
      You can use the ServiceDescription from the ADF of exercise F3.
      Use a df-search subgoal to search for a translation agent
      Save the translation agent in the variable ta
      If no translation agent could be found waitFor() some time and try again
    }

    Translate the words.
    for(int i=0; i<words.length; i++)

```


Exercise G1 - Socket Communication

We extend the simple translation agent from exercise C2 with a plan that sets up a server socket which listens for translation requests. Whenever a new request is issued (e.g. from a browser) a new goal containing the client connection is created and dispatched. The translation plan handles this translation goal and sends back some HTML content including some text and the translated word.

Create a new file for the ServerPlanG1

- Declare the ServerSocket as attribute within the plan

```
protected ServerSocket server;
```

- Create a constructor which takes the server port as argument and creates the server within it:

```
try
{
    this.server = new ServerSocket(port);
}
catch(IOException e)
{
    throw new RuntimeException(e.getMessage());
}
getLogger().info("Created: "+server);
```

- Additionally create a close method that can be used for shutting down the server socket:

```
public void close()
{
    try
    {
        getExternalAccess().getLogger().info("Closing: "+server);
        server.close();
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }
}
```

- In the body simply start a new thread that will handle client request in the run method. Additionally add an agent listener that gets invoked when the agent will be terminating. In this case the server is shut down:

```

new Thread(this).start();
getScope().addAgentListener(new IAgentListener()
{
    public void agentTerminating(AgentEvent ae)
    {
        close();
    }
    public void agentTerminated(AgentEvent ae)
    {
    }
});

```

- In the threads run method create and dispatch goals for every incoming request.

The external access's scheduleStep() method assures that modifications to the agent's goal base happen on the component step:

```

while(true)
{
    final Socket client = server.accept();
    getExternalAccess().scheduleStep(new IComponentStep()
    {
        public Object execute(IInternalAccess ia)
        {
            IBDIInternalAccess scope = (IBDIInternalAccess)ia;
            IGoal goal = scope.getGoalbase().createGoal("translate");
            goal.getParameter("client").setValue(client);
            scope.getGoalbase().dispatchTopLevelGoal(goal);
            return null;
        }
    });
}

```

Modify the EnglishGermanTranslationPlanG1 to handle translation goals

- Extract the socket from the goal and read the English word:

```

Socket client = (Socket)goal.getParameter("client").getValue();
BufferedReader in = new BufferedReader(new InputStreamReader(client.getInputStream()));

```

```
String request = in.readLine();
// Read the word to translate from the input string
```

- Translate the word as usual by using the query
- Send back answer to the client:

```
PrintStream out = new PrintStream(client.getOutputStream());
out.print("HTTP/1.0 200 OK\r\n");
out.print("Content-type: text/html\r\n"); out.println("\r\n");
out.println("<html><head><title>TranslationG1 - "+eword+"</title></head><body>");
out.println("<p>Translated from english to german: "+eword+" = "+gword+".");
out.println("</p></body></html>");
client.close();
```

Create a file **TranslationG1.agent.xml** by copying **TranslationC2.agent.xml**

- The addword plan and event declarations are not used and can be removed for clarity.
- Introduce the translation goal type:

```
<achievegoal name="translate">
  <parameter name="client" class="java.net.Socket"/>
</achievegoal>
```

- Introduce the new plan for setting up the server and start the plan initially:

```
<plan name="server">
  <parameter name="port" class="int">
    <value>9099</value>
  </parameter>
  <body class="ServerPlanG1"/>
</plan>
...
<configurations>
  <configuration name="default">
    <plans>
      <initialplan ref="server"/>
    </plans>
  </configuration>
</configurations>
```

- Modify the trigger of the translation plan to react on translation goals and add a parameter for the client:


```

<plan name="egtrans">
  <parameter name="client" class="Socket">
    <goalmapping ref="translate.client"/>
  </parameter>
  <body class="EnglishGermanTranslationPlanG1"/>
  <trigger>
    <goal ref="translate"/>
  </trigger>
</plan>

```

Starting and testing the agent\ Start the agent and open a browser to issue translation request. This can be done by entering the server url and appending the word to translate, e.g. `http://localhost:9099/dog.`](`http://localhost:9099/dog.`) The result should be printed out in the returned web page.

Chapter 9. Conclusion

We hope you enjoyed working through the tutorial and now are equipped at least with a basic understanding of the Jadex BDI reasoning engine. Nevertheless, this tutorial does not cover all important aspects about agent programming in Jadex. Most importantly the following topics have not been discussed:

Ontologies

Ontologies can be used for describing message contents. In more complex applications you usually want to transfer objects instead of simple strings. In Jadex for this purpose you could use arbitrary Java beans in connection with the SFipa.JADEX_XML language. If this language is specified for a message event the an xml encoder/decoder (from the Protégé .

Protocol Capabilities

The protocol capabilities, which belongs to the Jadex planlib provide ready-to-use implementations of some common interaction protocols. In Exercise F4 you could already see how to use the initiator side of the FIPA request protocol. For more details on the protocol capabilities, please have a look at the BDI User Guide . Conceptual details about goal-oriented protocols can be found in [Braubach et al. 2007].

Goal Deliberation

This tutorial only mentions the different goal types available in Jadex (perform, achieve, query and maintain). It does not cover aspects of goal deliberation, i.e. how a conflict free pursuit of goals can be ensured. Jadex offers the built-in *Easy Deliberation* strategy for this purpose. The strategy allows to constrain the *cardinality* of active goals. Additionally, it is possible to define *inhibition links* between goals that allow to establish an ordering of goals. Inhibited goals are suspended and can be reactivated when the reason for their inhibition has vanished, e.g. another goal has finished processing. Please refer also to the BDI User Guide for an extended explanation. Background information is available in the paper [Pokahr et al. 2005a].

Plan Deliberation

If more than one plan is applicable for a given goal or event the Jadex interpreter has to decide which plan actually will be given a chance to handle the goal resp. event. This decision process called plan deliberation can be customized with *meta-level reasoning*. This means that a custom defined meta-level goal is automatically raised by the system in case a plan decision has to be made. This meta goal can be handled by a corresponding meta-level plan which has the task to select among the candidate plans. Further details about meta-level reasoning can be found in the BDI User Guide and by looking into the source code of the “puzzle” agent included in the Jadex release.

Jadex BDI Architecture

During some of the exercises you may have used the Jadex debugger for executing Jadex agents step-by-step. But what makes-up one such step in the debugger? All steps represent BDI rules meaning that they are not at the application-level but on the architecture level. Examples for such BDI rules are “selecting plans for a given event”, “executing a plan step”, “creating a new goal” and many more. Basically, the Jadex interpreter selects one BDI rule after another and executes them when they are applicable in the current situation. This new architecture makes the Jadex framework efficient and also extensible as new rules can be added to the system easily. Details about the architecture are described in the BDI User Guide and the papers [Pokahr et al. 2005b], [Pokahr et al. 2009].

1 Bibliography

[Bauer et al. 2001] B. Bauer, J. Müller, and J. Odell. Agent UML: A Formalism for Specifying Multiagent Interaction. P. Ciancarini and M. Wooldridge. Proceedings of the First International Workshop on Agent-Oriented Software Engineering (AOSE 2000). Springer. Berlin, New York. 2001. pp.91-103.

- *\ [Bellifemine et al. 2007\]* F. Bellifemine, G. Caire, and D. Greenwood. Developing Multi-Agent Systems with JADE. John Wiley & Sons. New York, USA. 2007.
- *\ [Bratman 1987\]* M. Bratman. Intention, Plans, and Practical Reason. Harvard University Press. Cambridge, MA, USA. 1987.
- *\ [Braubach et al. 2004\]* L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf. Goal Representation for BDI Agent Systems. R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. Proceedings of the Second Workshop on Programming Multiagent Systems: Languages, frameworks, techniques, and tools (ProMAS04). Springer. Berlin, New York. 2004. pp.9-20.
- *\ [Braubach et al. 2005a\]* L. Braubach, A. Pokahr, and W. Lamersdorf. Jadex: A BDI Agent System Combining Middleware and Reasoning. R. Unland, M. Klusch, and M. Calisti. Software Agent-Based Applications, Platforms and Development Kits. Birkhäuser. 2005. pp.143-168.
- *\ [Braubach et al. 2005b\]* L. Braubach, A. Pokahr, and W. Lamersdorf. Extending the Capability Concept for Flexible BDI Agent Modularization. R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. Proceedings of the Third International Workshop on Programming Multi-Agent Systems (ProMAS'05). 2005. pp.99-114.
- *\ [Braubach et al. 2007\]* Lars Braubach, Alexander Pokahr. Goal-Oriented Interaction Protocols, Fifth German conference on Multi-Agent System TEchnologieS (MATES-2007).
- *\ [Busetta et al. 2000\]* P. Busetta, N. Howden, R. Rönquist, and A. Hodgson. Structuring BDI Agents in Functional Clusters. N. Jennings and Y. Lespérance. Intelligent Agents VI, Proceedings of the 6th International Workshop, Agent Theories, Architectures, and Languages (ATAL) '99. Springer. Berlin, New York. 2000. pp.277-289.
- *\ [Hindriks et al. 1999\]* K. Hindriks, F. de Boer, W. van der Hoek, and J.-J. Meyer. Agent Programming in 3APL. N. Jennings, K. Sycara, and M. Georgeff. Autonomous Agents and Multi-Agent Systems. Kluwer Academic publishers. 1999. pp. 357-401.
- *\ [Huber 1999\]* M. Huber. JAM: A BDI-Theoretic Mobile Agent Architecture. O. Etzioni, J. Müller, and J. Bradshaw. Proceedings of the Third Annual Conference on Autonomous Agents (AGENTS-99). ACM Press. New York. 1999. pp. 236-243.
- *\ [Lehman et al. 1996\]* J. F. Lehman, J. E. Laird, and P. S. Rosenbloom. A gentle introduction to Soar, an architecture for human cognition. Invitation to Cognitive Science Vol. 4. MIT press. 1996.
- *\ [McCarthy et al. 1979\]* J. McCarthy. Ascribing mental qualities to machine. M. Ringle. Philosophical Perspectives

in Artificial Intelligence. Humanities Press. Atlantic Highlands, NJ. 1979. pp. 161-195.

[Pokahr et al. 2005a] A. Pokahr, L. Braubach, and W. Lamersdorf. A Goal Deliberation Strategy for BDI Agent Systems. T. Eymann, F. Klügl, W. Lamersdorf, M. Klusch, and M. Huhns. In Proceedings of the third German conference on Multi-Agent System TEchnologieS (MATES-2005). Springer-Verlag. Berlin Heidelberg New York. 2005.

[Pokahr et al. 2005b] A. Pokahr, L. Braubach, and W. Lamersdorf. A Flexible BDI Architecture Supporting Extensibility. A. Skowron, J.P. Barthes, L. Jain, R. Sun, P. Morizet-Mahoudeaux, J. Liu, and N. Zhong. Proceedings of The 2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT-2005). IEEE Computer Society. 2005. pp. 379-385.

[Pokahr et al. 2005c] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI Reasoning Engine. R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. Programing Multi-Agent Systems. Kluwer Academic Publishers. 2005. pp.149-174.

[Pokahr et al. 2009] A. Pokahr, L. Braubach From a Research to an Industrial-Strength Agent Platform: Jadex V2 in: 9. Internationale Tagung Wirtschaftsinformatik 2009

[Rao and Georgeff 1995] A. Rao and M. Georgeff. BDI Agents: from theory to practice. V. Lesser. Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95). The MIT Press. Cambridge, MA, USA. 1995. pp.312-319.

[Shoham 1993] Y. Shoham. Agent-oriented programming. D. G. Bobrow. Artificial Intelligence Volume 60. Elsevier. Amsterdam. 1993. pp.51-92.

[Winikoff 2005] M. Winikoff. JACK Intelligent Agents: An Industrial Strength Platform. R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. Programing Multi-Agent Systems. Kluwer Academic Publishers. 2005. pp.175-193.

Chapter 1 - Introduction

The Active Components project aims at providing programming and execution facilities for distributed and concurrent systems. The general idea is to consider systems to be composed of components acting as service providers and consumers. Hence, it is very similar to the Service Component Architecture (SCA) approach and extends it in the direction of agents. In contrast to SCA, components are always active entities, i.e. they possess autonomy with respect to what they do and when they perform actions making them akin to agents. In contrast to agents communication is preferably done using service invocations.

This user guide provides a detailed description of the available functionalities of Jadex Active Components. In particular, the following topics are covered in the upcoming chapters:

- Chapter 02 Active Components describes the underlying conceptual foundations.
- Chapter 03 Asynchronous Programming gives an introduction to asynchronous methods using futures.
- Chapter 04 Component Specification illustrates how services can be found and invoked.
- Chapter 05 Services explains how service invocations work.
- Chapter 06 Web Service Integration describes how web services can be offered and used.
- Chapter 07 Platform Awareness explains how platforms can automatically find each other.
- Chapter 08 Security introduces the main security concepts of Jadex.
- Chapter 09 Auto Update describes services for spawning new platforms and updating running platforms to new versions.

Chapter 2 - Active Components

The roots of active components lie within the area of multi-agent systems and in the same way help constructing complex distributed systems composed of potentially concurrent interacting entities. Experience from many software projects has shown that agent concepts are valuable but cannot be applied easily in practice due to several inherent difficulties of agent technology. One important reason is the restriction of interaction means to asynchronous messages, which are a rather low-level concept and require programmers to learn details of rather complex message formats, speech acts and think in terms of protocols (the allowed sequences of messages). Messages and protocols have their merits in scenarios requiring negotiations or other advanced coordination activities but are a too complex scheme for many rather simple scenarios. Often a service based worldview (SOA, service oriented architecture) is sufficient, which distinguishes between service providers and users and model interactions simply in the form of service invocations. An existing drawback of SOA per se consists in the tendency to build up networks of many services without an overall architecture of the target system. This problem has been recently addressed by the service component architecture (SCA), a new software engineering approach that has been proposed by several major industry vendors including IBM, Oracle and TIBCO. SCA combines in a natural way the service oriented architecture with component orientation by introducing SCA components communicating via services. This allows modelling a system in terms of components acting as

service providers and consumers facilitating application design by offering means for assembling and especially by hierarchically (de)composing components and services.

Active components build on SCA and extend it in the direction of software agents. The general idea is to transform passive SCA components into autonomously acting service providers and consumers in order to better reflect real world scenarios which are composed of various active stakeholders. In the figure below an overview of the synthesis of SCA and agents to active components is shown.

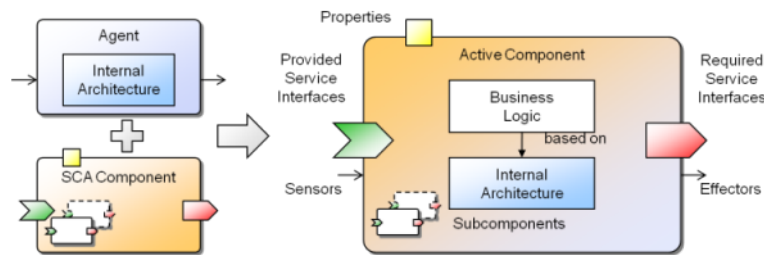


Figure 47: 02 Active Components@ac.png

Active Component Structure

The figure presents on the right hand side the structure of an active component. It yields from conceptually merging an agent with an SCA component (shown at the left hand side). An agent is considered here as an autonomous entity that is perceiving its environment using sensors and can influence it by its effectors. The behavior of the agent depends on its internal reasoning capabilities ranging from rather simple reflex to intelligent goal-directed decision procedures. The underlying reasoning mechanism of an agent is described as an agent architecture and determines also the way an agent is programmed. On the other side an SCA component is a passive entity that has clearly defined dependencies with its environment. Similar to other component models these dependencies are described using required and provided services, i.e. services that a component needs to consume from other components for its functioning and services that it provides to others. Furthermore, the SCA component model is hierarchical meaning that a component can be composed of an arbitrary number of subcomponents. Connections between subcomponents and a parent component are established by service relationships, i.e. connecting their required and provided service ports. Configuration of SCA components is done using so called properties, which allow values being provided at startup of components for predefined component attributes. The synthesis of both conceptual approaches is done by keeping all of the aforementioned key characteristics of agents and SCA components. On the one hand, from an agent-oriented point of view the new SCA properties lead to enhanced software engineering capabilities as hierarchical agent composition and service based interactions become possible. On the other hand, from an

SCA perspective internal agent architectures enhance the way how component functionality can be described and allow reactive as well as proactive behavior.

Active Components by Example

This section illustrates several example applications that are part of the Jadex distribution. The purpose of this section is twofold. First, it aims at improving the understanding of the active components approach by using concrete scenarios. In this respect, it tries to highlight the advantages of the approach with respect to typical challenges of distributed systems. Second, it tries to encourage using the existing examples as a source for documentation and inspiration when developing active components programs. If descriptions in this user guide do not appear clear, it is often helpful to check how the features have been used in some of the examples.

Chat

The chat is a peer-to-peer application that allows users exchanging simple text messages. In the distributed scenario, each user will host a Jadex platform and start a chat component on her local computer. The Jadex platforms will discover each other automatically without configuration efforts using the built-in awareness mechanisms (Platform Awareness) therefore allowing the chat components to interact with each others services. In the peer-to-peer design, each component acts as a service provider (for receiving chat messages for its user) and a service consumer (for sending chat messages to other users). The application design is shown in the following figure.

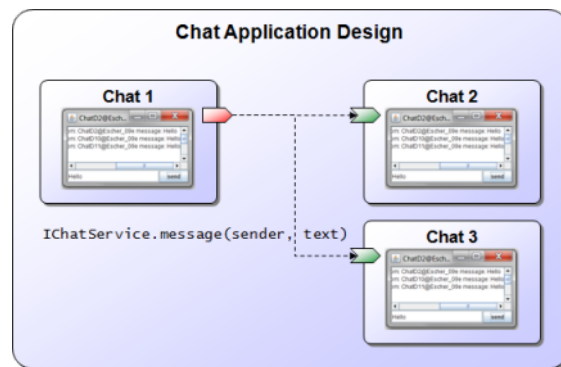


Figure 48: 02 Active Components@chatdesign.png

Chat application design

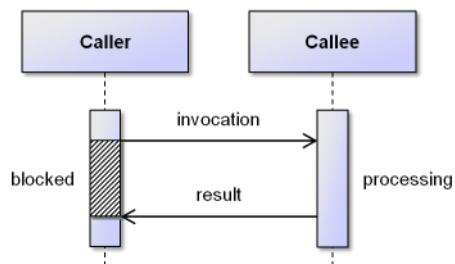
The figure shows the interaction between three chat components. The interaction is specified by the Java service interface *IChatService*, which declares

the *message(sender, text)* method. For sending a chat message Chat1 acts as a service consumer, i.e. it searches for chat services that are provided by the other chat components (Chat2, Chat3) and invokes their *message()* methods. The implementation of the chat components is described in depth as a running example in the Jadex Active Components Tutorial .

Mandelbrot

Disaster Management

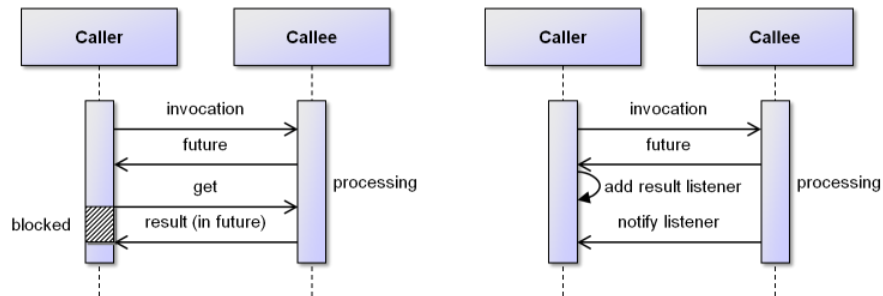
Chapter 3 - Asynchronous Programming



Synchronous call

Understanding the difference between the synchronous and the asynchronous programming style is fundamental for using active components. The meaning of a typical synchronous call is shown in the figure above. It can be seen that a *Caller* is calling e.g. a method on a *Callee*. The result of this call is computed by the callee and delivered to the caller as result afterwards. It is important to note here that the caller is *blocked* while the callee is calculating the result of the call. Such blocking is uncritical when calls are very fast and the blocking does not hinder the caller to perform other tasks in the meantime. In distributed systems using the synchronous invocation scheme has a severe drawback besides the performance loss. The scheme is inherently deadlock prone as call graphs may lead to cycles so that no participant has a chance to leave its waiting state and continue processing. For this reason asynchronous calls should be used (The asynchronous call scheme has successfully been introduced in other areas such as in the context of the web with HTTP with AJAX or as programming model in the Google AppEngine).

Asynchronous Call Concepts

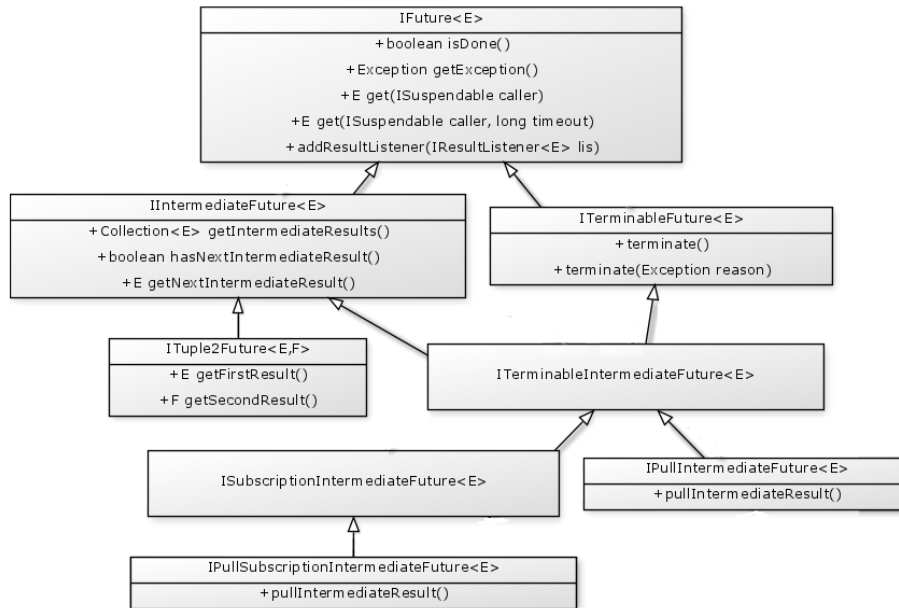


a) Asynchronous call with wait by necessity
 call with listener notification

b) Asynchronous

An asynchronous call is simply a call that does not block the caller while the callee is serving the request. Hence, the question arises how the caller can determine if the processing of the callee has been finished and how it can fetch the result of the call. For this purpose the *future* abstraction has been introduced (In Jadex it is represented by *jadex.commons.future.IFuture*). A future represents the result of an asynchronous call and can be seen as a container for the real result of the call. The underlying idea is that the callee delivers immediately a future object to the caller and uses the future itself to store the real result after having finished its processing. A future allows the caller to check if the result is already available without blocking (*isDone()*). Furthermore, if the caller cannot proceed further without knowing the result of the call, it can decide to wait for the result in a possibly blocking manner (*get()*). If the call blocks depends on whether the result has already been provided by the callee or not. Fetching the result in this way is called *wait by necessity*. The blocking variant is shown in the figure above. As there is a chance for blocking in *wait by necessity* using this scheme should also be avoided in the distributed case. A purely non-blocking invocation scheme can be derived when the callee takes over the responsibility for notifying the caller when processing has finished. This means that the caller installs a result listener (in Jadex the *jadex.commons.future.IResultListener*) on the future and this result listener is called in the moment the callee provides the result value. This scheme is shown in figure b) and is the basis of the Jadex active components programming model.

Programming Futures and Listeners



Future types

As can be seen in the Figure above, in Jadex several different future types can be used, which all extend the common base interface *jadex.commons.future.IFuture*. On the one hand *ITerminableFutures* allow for aborting an ongoing asynchronous method execution from the caller side by calling *terminate* at any time and on the other hand *IIntermediateFutures* can be used when more than one result value is expected. Intermediate futures can return these result values one by one allowing a caller to continue processing as soon as the first result becomes available. Of course, also a terminable version of an intermediate future exists. Finally, very similar to terminable intermediate futures are *ISubscriptionFutures*, which only relax result storage, i.e. a subscription future immediately forgets intermediate results as soon as they have been passed to one subscriber (listener). It follows a brief introduction of all currently available future types:

- **IFuture<E>**: Basic type of future that represents a container for a value of type E. Allows for blocking wait with *get()* and listener callbacks using *addResultListener()*. The corresponding listener type is *IResultListener* and has two methods: *resultAvailable()* is called with the result value as parameter and *exceptionOccurred()* is invoked in case of an invocation error.
- **IIntermediateFuture<E>**: The intermediate future allows for receiving multiple result values of the same type (E). Again receiving can be done in a blocking fashion (using *getNextIntermediateResult()*) and via listeners. For this kind of futures with *IIntermediateResultListener* a specific listener

type exists that supports reacting to individual results (using `intermediateResultAvailable(E result)`). Furthermore, it is signalled via `finished()` that no all values have been received.

- **ITerminableFuture<E>**: A terminable future allows the caller to cancel the task at any point in time by called `terminate()`. This termination will reach the called entity which may react to the request by stopping its activities regarding the invocation (the callee is not forced to do so). At callee side a termination command can be used to state what should be done when a call is cancelled.
- **ITuple2Future<E,F>**: The tuple2 future can be used if exactly two result values should be returned. The tuple2 future is strongly typed by using type E for the first and type F for the second result value. With the `ITuple2ResultListener` a listener type is available that offers two corresponding methods `firstResultAvailable()` and `secondResultAvailable()` which are called when the results are available. It has to be noted that the results need not necessarily be provided in order, i.e. `setSecondResult()` can be called also before `setFirstResult()`. The same order is then perceived at the receiver side, i.e. `first secondResultReceived()` is invoked in that case.
- **ITerminableIntermediateFuture<E>**: This is the terminable version of an intermediate future.
- **ISubscriptionIntermediateFuture<E>**: A subscription future can be used to establish a **publish subscribe relationship** between the caller and callee. It is only slightly different with respect to an intermediate future. The main difference is that it does not save the intermediate values but uses a fire and forget semantics, i.e. after a value has been propagated it will not be stored in the internal result collection. This also means that a listener that is added with some delay will not receive the earlier results. An exception to this rule is that the future saves previous values until the first listener is added.
- **IPullIntermediateFuture<E>**: A pull future allows for realizing an **iterator relationship** between caller and callee. This means that is this case the caller can decide when it wants to receive the next result by calling `pullIntermediateResult()`. The functionality that should be executed in case of a pull of the caller is supplied as command by the callee.
- **IPullSubscriptionIntermediateFuture<E>**: The subscription version of the pull future.

The detailed interface of the `jadex.commons.future.IFuture` is also shown in the figure above. It can be seen that a future has a generic type, which allows for having method signatures that also unveil the real return value of the method. As an example consider the example `sayHello()` method signature below. It differs from a synchronous call only in the declared return type (here a String within the future).

```
public interface IFuture<E>
{
```

```

public static final IFuture<Void> DONE = new Future<Void>((Void)null);

public boolean isDone();
public E get(ISuspendable caller);
public E get(ISuspendable caller, long timeout);
public void addResultListener(IResultListener<E> listener);
}

```

The methods of the *IFuture* have already been introduced in the context of the general programming concept above and are therefore only briefly explained:

- *isDone()* can be used to check whether the callee has already supplied the real result.
- *get()* methods initiate possibly a blocking call and should therefore only be used in exceptional cases. The *get* methods expect as parameter a so called *jadex.commons.future.ISuspendable* and optionally a timeout. The interface *suspendable* abstracts the way how the future should suspend and resume the caller in case blocking is needed. Hence, with *ThreadSuspendable* a default implementation exists that simply suspends and resumes the current thread. An example usage that operates on a future return from the *sayHello()* method is *String text = future.get(new ThreadSuspendable())*. If the timeout variant is employed and the callee does not provide the result within the given deadline a *jadex.commons.concurrent.TimeoutException* is thrown.
- *addResultListener()* allows for adding a result listener to the future. This listener is notified *once* when the result is available. If the result is already set the listener is notified right after its addition. Arbitrary many listeners can be added to a future.

The *IResultListener* interface is shown below. The interface is also generic and allows for creating generically typed listeners. The general contract is that a result listener will only be called once (or not at all if the result is never set by the callee). In case everything has worked as expected the *resultAvailable()* method will be invoked and the result will be passed as parameter. Otherwise, in case of an abnormal operation, the *exceptionOccurred()* method is executed and the corresponding exception is made accessible via a parameter.

```

public IFuture<String> sayHello();

```

In the following we will revisit the *sayHello()* example and use the listener style to retrieve the result. It can be seen that an anonymous Java class is added as listener to the future. This listener implementation just prints the results of the invocation.

```

public interface IResultListener<E>
{
    public void resultAvailable(E result);
}

```

```

    public void exceptionOccurred(Exception exception);
}

```

Programming Intermediate Futures and Listeners

In certain situations an asynchronous call might not only return a single result but a collection of result values. Of course, for this purpose a future of type collection could be used (for example *IFuture<Collection<String>>*), but if the result values are computed independently of each other this can lead to the severe drawback that the caller has to wait until the last value has been provided till it can continue processing. In case that the caller wants to use the values as soon as they are available an alternative result value scheme has to be used. This scheme is based on *IIntermediateFuture<E>* and *IIntermediateResultListener<E>* both from the package *jadex.commons.future*.

The interface definition of the *IIntermediateFuture<E>* is shown below. It can be seen that the intermediate future class extends the normal future class and defines its generic type as collection of the future type (*IFuture<Collection <E>>*). Furthermore, it only adds one method signature called *getIntermediateResults()* that allows to non-blockingly fetch the currently available intermediate results as generically typed collection.

```

IFuture<String> future = obj.sayHello();
future.addListener(new IResultListener<String>()
{
    public void resultAvailable(String result)
    {
        System.out.println("Say hello call result is: "+result);
    }
    public void exceptionOccurred(Exception exception)
    {
        System.out.println("An exception occurred: "+exception);
    }
});

```

More interesting than the intermediate future interface is the *IIntermediateResultListener<E>* specification. It can be seen that the intermediate result listener also extends the normal result listener and redefines its generic type to collection. Hence, it offers the same methods as the normal result listener (*resultAvailable()* and *exceptionOccurred()*) plus two new methods. The general idea of letting the intermediate versions of the future and listener extend the normal versions is that it should be possible to chain listeners and futures independently of their type, i.e. if one e.g. does not care about intermediate results it is possible to add a normal listener to an intermediate future. For these reasons the contract of an intermediate listener has been defined as follows:

- Either the *resultAvailable()* method is called once (e.g. when an intermediate result listener is added as a listener to a non-intermediate future)
- Or the *intermediateResultAvailable()* method is called once for each result and after the last value has been provided the *finished()* method is called once.
- In case of an exception during intermediate result reporting the *exceptionOccurred()* method is called once and no further intermediate results are generated.

```
public interface IIntermediateFuture<E> extends IFuture<Collection <E>>
{
    public Collection<E> getIntermediateResults();
}
```

Example

Let us assume we want to use the intermediate listener to search for different chat service providers sitting on arbitrary nodes in a network. Each chat service has a method *message()* that can be invoked to send a text message to the corresponding chat partner. The example intermediate listener implementation below shows that the *resultAvailable()* method as well as the intermediate counterparts *intermediateResultAvailable()* and *finished()* are used. With this technique the listener reacts failsafe with respect to the implementation of the callee, i.e. it will work if the callee truly provides intermediate results or just one bulk result. Hence, if the implementation of the callee is unknown both result listener methods should be implemented. In the example, it can also be seen that if processing of results is independent from the mode they are delivered, it is convenient to call *intermediateResultAvailable()* in *resultAvailable()* for each partial result value.

```
public interface IIntermediateResultListener<E> extends IResultListener<Collection <E>>
{
    public void intermediateResultAvailable(E result);
    public void finished();
}
```

Library Support for Listeners

| Task | Called on Thread | Intermediate | Class |
|--|------------------|--------------|---|
| Only log exception | Swing | no | DefaultResultListener<E> |
| | Swing | yes | IntermediateDefaultResultListener<E>
SwingDefaultResultListener<E>
SwingIntermediateDefaultResultListener<E> |
| Delegate result and exception
(For cascading future of same generic type) | Swing | no | DelegationResultListener<E> |
| | Swing | yes | IntermediateDelegationResultListener<E>
SwingDelegationResultListener<E>
SwingIntermediateDelegationResultListener<E> |
| Delegate exception
(For cascading future of different generic type) | Swing | no | ExceptionDelegationResultListener<E, T> |
| | Swing | yes | IntermediateExceptionDelegationResultListener<E, T>
SwingExceptionDelegationResultListener<E, T>
SwingIntermediateExceptionDelegationResultListener<E, T> |
| Listeners Supporting Delegation | | | |
| Enforce timeout for call | | no | TimeoutResultListener<E> |
| | | yes | TimeoutIntermediateResultListener<E> |
| Execute on component thread | Component | no | ComponentResultListener<E> |
| | Component | yes | IntermediateComponentResultListener<E> |
| Execute on Swing thread | Swing | no | SwingResultListener<E> |
| | Swing | yes | SwingIntermediateResultListener<E> |
| Count n results and propagate when complete | | no | CounterResultListener<E> |
| Collect n results and propagate when complete | | no | CollectionResultListener<E> |

Result listeners

In Jadex there are several ready to use listeners that help with recurring use cases. These ready to use listeners are described in the following. Using these listeners can speed up development and simplify the code. For most of the listener implementations also intermediate versions exist. An overview of the currently supported listener types is given in the figure above. It was made to help you finding the right listener class quickly by reading it from left to right and taking a choice in each column. The first column describes the task that is achieved by the listener, the second column reveals on which thread the listener call will be performed (if not explicitly stated no thread switch will be done) and the third column distinguishes between single result and intermediate listener versions. Finally, in the fourth column you find the concrete Java class that can be used. It has additionally to be noted that listeners in the lower part of the table (below the heading ‘Listeners Supporting Delegation’) allow for an easy chaining of multiple listeners. This is achieved by delegation to another listener which has to be provided to the new listener (in most cases as constructor argument). To show you how the table can be used consider the case you want to create a listener that signals a single result to the user in a Swing user interface and adds a timeout to the underlying call. In this case one could chain a *SwingResultListener* with a *TimeoutResultListener*.

DefaultResultListener<E>

If exceptions cannot be handled or need not to be handled default result listeners can be used. They implement the exception occurred method by logging a severe message. The logger to be used can be either supplied via the listener constructor or a default logger with the name “default-result-listener” will be

used. As a default listener already implements *exceptionOccurred()* only the *resultAvailable()* method needs to be implemented by the developer. In the example code snippet below it can be seen how a default result listener can be used in context of using a service.

```
IIntermediateFuture<IChatService> fut = ia.getServiceContainer().getRequiredServices("chatservice");
fut.addListener(new IIntermediateResultListener<IChatService>()
{
    public void resultAvailable(Collection<IChatService> result)
    {
        for(Iterator<IChatService> it=result.iterator(); it.hasNext(); )
        {
            IChatService cs = it.next();
            intermediateResultAvailable(cs);
        }
    }
    public void exceptionOccurred(Exception exception)
    {
        exception.printStackTrace();
    }
    public void intermediateResultAvailable(IChatService result)
    {
        result.message("Hugo", "Hi chat partner");
    }
    public void finished()
    {
    }
});
```

DelegationResultListener<E>

In Java methods can either use try-catch blocks to handle exceptions themselves or they can declare to throw exceptions in order to let the caller of a method handle potentially occurring exceptions. The same styles of programming are useful in the asynchronous listener based case as well. In order to let an exception be treated by a caller, delegation listener can be used. They forward *resultAvailable()* and *exceptionOccurred()* to a specified future. When using the generically typed version of the delegation listener this means that the future and the listener have to be of the same type. As an example consider a method that calls another method and wants to forward exceptions of the invoked method to the caller. The method itself uses a delegation result listener that overrides the *customResultAvailable()* method to modify the result retrieved from calling the *sayHello()* method. Should *sayHello()* raise an exception, the delegation listener will set this exception automatically on the future of the *sayHello2()* method.


```

IFuture<IClockService> fut = agent.getServiceContainer().getRequiredService("clockservice");
fut.addListener(new DefaultResultListener<IClockService>()
{
    public void resultAvailable(IClockService cs)
    {
        System.out.println("Current time is: "+new Date(cs.getTime()));
    }
});

```

ExceptionDelegationResultListener<E, T>

In order to allow also chaining of futures and listeners with different generic types the exception variant of the delegation listener can be used. The generic types <E, T> denote the type of the received and forwarded result respectively. In contrast to the delegation listener the exception delegation listener only implements the *exceptionOccurred()* method so that the *customResultAvailable* method always has to be implemented by the developer himself. The example below illustrates how a result type can be transformed from String to String[] using the exception delegation listener. The method *sayHellos()* returns a String[] but gets a String from the *sayHello()* method. Therefore, it overrides the *customResultAvailable()* to set a result of of type String[].

```

public IFuture<String> sayHello2()
{
    final Future<String> ret = new Future<String>();
    sayHello().addListener(new DelegationResultListener<String>(ret)
    {
        public void customResultAvailable(String result)
        {
            ret.setResult(result+"2");
        }
    });
    return ret;
}

```

TimeoutResultListener<E>

For writing robust programs it is often essential not to wait indefinitely for a result. Instead one often assumes that an error occurred, if the result is not made available after some predefined timeout. The *TimeoutResultListener* captures this assumption. It is created with a target result listener and a timeout value in milliseconds. If the result is made available before the timeout, it is immediately forwarded to the target listener. When the timeout passes without a result being

available, the *exceptionOccurred()* method is called with a timeout exception. In the latter case, the result will never be forwarded to the target listener, even if it becomes eventually available after the timeout.

The timeout result listener uses any available clock service for measuring the timeout. Therefore it needs to be created with an *IExternalAccess* object used to lookup the clock service. The constructor signature is as follows:

```
public IFuture<String[]> sayHellos()
{
    final Future<String[]> ret = new Future<String[]>();
    sayHello().addResultListener(new ExceptionDelegationResultListener<String, String[]>(ret)
    {
        public void customResultAvailable(String result)
        {
            ret.setResult(new String[]{result, result});
        }
    });
    return ret;
}
```

The following snippet shows how the timeout listener can be used:

```
public TimeoutResultListener(long timeout, IExternalAccess exta, IResultListener<E> listener)
```

SwingDefaultResultListener<E>

In many cases it is of importance on which thread the listener methods are called. In context of Java AWT or Swing GUI programming it needs to be ensured that widgets are only accessed from the Swing thread. This can always be achieved by calling *SwingUtilities.invokeLater()* and providing a *Runnable* that is executed on the Swing thread later on. To avoid manually calling invoke later in the listener methods the Swing variants of the listeners can be used (package *jadex.base.gui*). They ensure that listener methods are only called on the Swing thread.

The *SwingDefaultResultListener* extends the *DefaultResultListener* and implements

additional behavior for handling exceptions. If a gui component is supplied in the constructor, any exception is alerted to the user using a simple message dialog. Otherwise the exception is logged as before.

The example below shows how the result of a *sayHello()* call is set in a *JTextField*.

```
sayHello().addResultListener(new TimeoutResultListener<String>(3000, agent.getExternalAccess
```

```

    new IResultListener<String>()
{
    public void resultAvailable(String msg)
    {
        System.out.println("Hello: "+msg);
    }
    public void exceptionOccurred(Exception exception)
    {
        if(exception instanceof TimeoutException)
        {
            System.out.println("No answer received");
        }
    }
}
});

```

SwingDelegationResultListener<E>

The *SwingDelegationResultListener* is the swing variant of the *DelegationResultListener*.

It allows forwarding the result or exception to another future. In addition, the listener methods are invoked on the swing thread. The common use case is to only override the *customResultAvailable* method, e.g., for doing gui stuff with a result, but keeping the *customExceptionOccurred* implementation for forwarding of errors.

```

JPanel panel;
...
sayHello().addResultListener(new SwingDefaultResultListener<String>(panel)
{
    public void customResultAvailable(String result)
    {
        mytextfield.setText(result);
    }
});

```

ExceptionSwingDelegationResultListener<E, T>

Represents the exception variant of the *SwingDelegationResultListener<E>*. Also ensures that all listener calls are performed on the Swing thread. Behaves in the same way as the *SwingDelegationResultListener<E>* so that no further explanation is given here.

CounterResultListener<E>

The counter result listener is useful when a known number of calls needs to be done and after all calls have been finished processing should continue. A common use case is a loop with asynchronous calls inside. In this case, the loop is sequential but the calls itself may be executed concurrently. The counter result listener needs to be initialized with the expected number of calls and a target listener that is called once all results have been received. Using the *ignorefailures* flag it can be customized how exceptions of single calls are treated. Using the default behaviour (*ignorefailures*=false) an exception immediately leads to calling *exceptionOccurred()* on the delegation listener. If *ignorefailures* is set to true, always *resultAvailable()* is called on the target listener, after all calls have terminated but independently of how many exceptions occurred. Please note that the counter result listener does not collect results of the single calls and thus expects a result listener of type void as delegation listener. To collect the result the collection result listener can be used (see below). If intermediate results or exceptions are of interest two corresponding methods can be overridden (*intermediateResultAvailable()* and *intermediateExceptionOccurred()*).

```
Future<String> ret = new Future<String>();
sayHello().addResultListener(new SwingDelegationResultListener<String>(ret)
{
    public void customResultAvailable(String result)
    {
        mytextfield.setText(result);
        ret.setResult(result);
    }
});
```

The example code below uses a counter listener to start a hello world agent on every platform it knows. The known platforms are represented by a list of component management services (cmslist) that has been retrieved by some other code (e.g. a global scoped search). Afterwards the counter result listener is defined with a delegation listener that prints out a message after all components have been created. Last part of the code example is the loop which fetches a cms from the list and instructs it to create a hello world agent. The same counter listener is added to all asynchronous calls. If an exception occurs within a call the counter listener aborts (*ignorefailures* is false) and calls *exceptionOccurred()* on the default listener (which will just print a log message that an exception occurred).

```
public CounterResultListener(int num, boolean ignorefailures, IResultListener<Void> target),
```

CollectionResultListener<E>

The collection result listener is similar to the counter result listener but collects the intermediate results in contrast to the latter. It needs to be supplied with the expected number of calls and a target result listener of type collection. Additionally, the *ignorefailures* flag can be used to customize error behavior in the same way as described above in the context of the counter result listener. The constructor of the listener is also shown below for convenience. Using a collection listener for a known number of asynchronous calls allows for concurrent call processing and afterwards downstream handling of the collected results. Please note that the number of results is equal to the number of calls only if *ignorefailures* is false. Otherwise, the result collection may contain fewer result values.

```
List<IComponentManagementService> cmslist = ...;

CounterResultListener<IComponentIdentifier> lis = new CounterResultListener<IComponentIdentifier>
    (cmslist.size(), false, new DefaultResultListener<Void>())
{
    public void resultAvailable(Void result)
    {
        System.out.println("Created all components.");
    }
};

for(int i=0; i<cmslist.size(); i++)
{
    cmslist.get(i).createComponent(null, "jadex.micro.examples.helloworld/HelloWorldAgent.class")
        .addResultListener(lis);
}
```

The example is nearly identical to the example explained above in the context of the counter result listener. Instead of a counter listener a collection listener is used with “*ignorefailures*” set to true. This means that the *resultAvailable()* method of the listener is invoked after all calls have finished. The listener finally prints the list of successfully created agents.

```
public CollectionResultListener(int num, boolean ignorefailures, IResultListener<Collection>
```

ComponentResultListener<E>

The component result listener is similar to the swing result listeners, as it can also be used to execute listener methods on another thread. Here, the listener methods are called on the thread of a specific component. Executing on the right component thread is important to establish state consistency: When always

accessing the internal state of a component from the same thread, no race conditions can occur and inconsistent modifications are effectively prevented.

In many cases, the Jadex framework takes care of executing code on the right component thread. By default, calls to service implementations as well as results from service searches and subsequent service calls are automatically executed on the component thread. E.g. in the example below, an agent searches for and invokes the hello service and stores the result in a field. This is safe, because both *resultAvailable* methods are called on the agent's thread.

```
List<IComponentManagementService> cmslist = ...;

CollectionResultListener<IComponentIdentifier> lis = new CollectionResultListener<IComponentIdentifier>(
    cmslist.size(), true, new DefaultResultListener<Collection<IComponentIdentifier>>()
{
    public void resultAvailable(Collection<IComponentIdentifier> result)
    {
        System.out.println("Created components: "+result);
    }
});

for(int i=0; i<cmslist.size(); i++)
{
    cmslist.get(i).createComponent(null, "jadex.micro.examples.helloworld/HelloWorldAgent.class")
        .addResultListener(lis);
}
```

Typical cases, where the automatic thread management of listener methods is not in control is when using the static methods of the *SServiceProvider* helper class and when scheduling steps on other components using *IExternalAccess.scheduleStep()*. In these cases, a component result listener should be created manually. The component result listener needs to be supplied with another result listener and an external access of the component on which the listener should be executed. The corresponding constructor is shown below. The result listener that is passed as arguments to the component result listener is used for delegation, i.e. this listener should contain the domain logic to be executed. In many component kernels there is a convenience method called *createResultListener(IResultListener listener)*, which creates a component result listener behind the scenes and hides passing the external access of the component.

The example code assumes that the body method of a micro agent is executed and the component management service of the platform has been fetched to a variable named "cms" using the *SServiceProvider* helper class. On the component management service the method *createComponent()* is utilized to create a new component instance of type *HelloWorldAgent*. After creation has finished successfully, the listener prints out that the component identifier. This print statement is executed on the component thread of the micro agent.

```

@Agent
MicroAgent agent;

String msg;

@AgentBody
public void body()
{
    IFuture<IHelloService> fhello = agent.getServiceContainer().getRequiredService("hello");
    fhello.addListener(new DefaultResultListener<IHelloService>()
    {
        public void resultAvailable(IHelloService hello)
        {
            hello.sayHello().addListener(new DefaultResultListener<String>()
            {
                public void resultAvailable(String msg)
                {
                    // Safe to access component state.
                    HelloClientAgent.this.msg = msg;
                }
            });
        }
    });
}

public ComponentResultListener(IResultListener<E> listener, IExternalAccess access)

```

Sequential Asynchronous Loops

In the previous examples loops were used in which the asynchronous calls have been processed potentially concurrently. In some cases a sequential loop containing asynchronous calls is needed. This can be achieved by using a method that recursively calls itself. The code example shown below again creates hello world agents at all known platforms. But in contrast to the variants shown before, this time the loop is realized with a method that calls itself as long as the iterator has more elements. The method calls the asynchronous method *createComponent()* and adds an exception delegation listener to the future. In case an exception occurs it is delegated to the future that is returned by the method. Otherwise it calls recursively itself and uses a delegation result listener to chain the results. If the iterator has no more elements the method returns a future with result null. This will cause the recursion to end and the chain of listeners and futures to be finished as well so that the outermost caller of the method is notified.

```

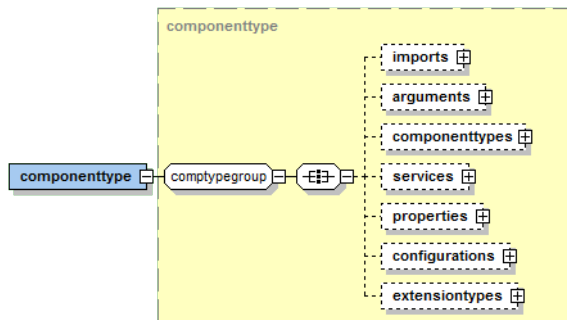
@AgentBody
public void body()
{
    // ... fetching cms omitted
    cms.createComponent(null, "jadex.micro.examples.helloworld/HelloWorldAgent.class", null, null, null)
        .addResultListener(createResultListener(new DefaultResultListener<IComponentIdentifier>()
        {
            public void resultAvailable(IComponentIdentifier cid)
            {
                System.out.println("Created component: "+cid);
            }
        }
        )
    )
}

```

Chapter 4 - Component Specification

In Jadex all component types (e.g. micro agents, BDI agents, BPMN workflows) share the same active component characteristics. Depending on the type of the component the definition is based on Java annotations or XML elements. The following explanations make extensive use of XML-based structure diagrams, but they are valid for annotations as well.

As can be seen in the figure below an active component specification is composed of *imports*, *arguments*, *componenttypes*, *services*, *properties*, *configurations* and *extensiontypes*. These elements will be explained in detail in the next subsections.

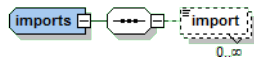


Component type

Imports

The imports can be used in the same way as in Java to specify the classes and packages that should be used for class and resource loading. In addition, these imports are helpful if Java expressions are used because the expressions

are evaluated taking into account the defined imports. It has to be noted that normal Java imports cannot be used for expression evaluation as the import statements are not preserved within the class file. For this reason the annotation *@Imports* can be employed. As usual, single classes are defined per name and packages using the *-notation.



Imports

In the examples below the package *java.util.** and the class *java.net.URL* are imported.

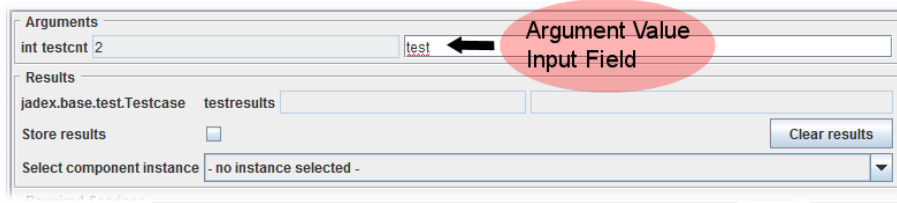
```
<imports>
  <import>java.util.*<import>
  <import>java.net.URL<import>
</imports>
```

```
@Imports(
{
  "java.util.*",
  "java.net.URL"
})
```

Arguments

Active components can have arguments and results. The arguments are supplied at startup of the component and the results can be fetched after the component has terminated. This allows to use components in a functional way, i.e. a component can be started and given argument values. The functional component computes something, sets its result values and terminates itself. The component creator is notified that the component has ended and can read the results for further processing. In order to use arguments and results at runtime it is necessary in the component type to declare the allowed argument and result types. For each argument and result type the following details can be specified:

- *name*: The name of the argument or result.
- *class*: The Java type of the argument or result.
- *defaultvalue*: (Optional) expression for the argument or result value if nothing else is supplied. The default value is defined as Java expression that is evaluated once.



Arguments

The example code snippets shows how two arguments with the names “number” and “obj” and one result called “res”. The type of the first is *Integer* and the type of the latter is *Customer* (if *Customer* has a package it has to be declared either fully qualified or it has to be imported). Both arguments have default values, which are *10* and *new Customer(“Sparky”)* respectively. The result is of type *boolean* and has no default value.

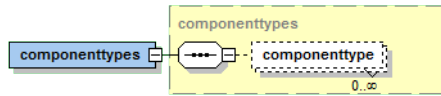
```
<arguments>
  <argument name="number" class="Integer">10</argument>
  <argument name="obj" class="Customer">new Customer("Sparky")</argument>
  <result name="res" class="boolean"/>
</arguments>
```

```
@Arguments(
{
  @Argument(name="number", clazz=Integer.class, defaultvalue="10"),
  @Argument(name="obj", clazz=Customer.class, defaultvalue="new Customer(\"Sparky\")")
})
@Results(@Result(name="res", clazz=boolean.class))
```

Component Types

Active components allow for hierarchical decomposition, i.e. an active component may consist of an arbitrary number of subcomponents. The types of potentially created subcomponents should be declared within the component types section. The declaration of a subcomponent type is done using the following parameters:

- *name*: The local name of the component type. This name can be used at other places to refer to the component type. Especially in the components section to create component instances of a given type.
- *filename*: The filename of the referenced active component type. The filename can either include the package structure or contain only the name itself if the package is imported.



Component types

As example the declaration of a heatbug as subcomponent type is illustrated. It has the name *Heatbug* and refers to the file *jadex/micro/examples/heatbugs/HeatbugAgent.class*.

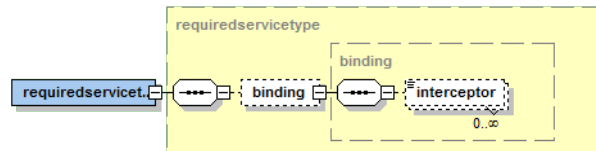
```
<componenttypes>
  <componenttype name="Heatbug" filename="jadex/micro/examples/heatbugs/HeatbugAgent.class"/>
</componenttypes/>
```

```
@ComponentTypes(@ComponentType(name="Heatbug", filename="jadex/micro/examples/heatbugs/HeatbugAgent.class"))
```

Services

Active components realize component orientation in the same way as traditional component approaches. In order to ensure a high degree of self-containedness of components each component type has explicitly to declare which functionalities it uses and needs. These aspects are defined in terms of *required* and *provided* services.

Required Services



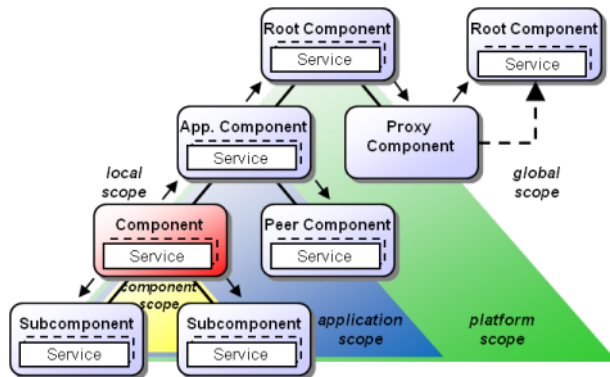
Required service type

A required service represents a needed functionality of another component. The specification of a required service is composed of two aspects. First the service definition describes what kind of service is needed and secondly the binding describes how a service can be found.

A required service is defined using the following properties:

- *name*: The name that can be used to fetch the service programmatically. A required service can be fetched using the *service container* of a component.
- *class*: The type of the required service defined by an interface.
- *multiple*: If multiple is set to true, not one but all services of the given interface are searched and will be returned when requested.

The optional service *binding* has the following options:

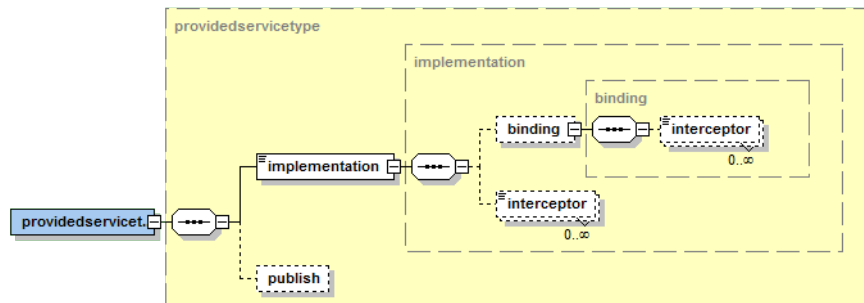


Search scopes

- *scope*: The scope of the search (cf. also the figure above). Static constants for search scopes are available via the class `RequiredServiceInfo`. Currently several predefined scopes are available (*application* being the default):
 - *local*: Consider only service within the component itself (the one that issues the search).
 - *component*: Includes services of the component itself and all subcomponents.
 - *application*: It is assumed that applications are directly started on the platform (not as subcomponents). The application scope includes services of all components up to the uppermost application component (the platform is excluded). Please note that currently application scope does not support distributed applications. If a distributed search is needed scope global has to used.
 - *platform*: This scope includes all components of the platform the component is running on.
 - *global*: Global scope extends platform scope towards connected remote platforms. These connected remote platforms are represented as proxy components on the platform. (If platform awareness is enabled these proxies are automatically created and delete as platforms are discovered or leave. Without awareness such proxy objects can be created programmatically or via the connect button in the Starter of the JCC).
 - *parent*: (not shown in figure) Parent scope refers to services of the parent component only. (This scope is also used for static component service connections, e.g having a component A with two subcomponents B and C, in A it should be defined (or overridden) that B uses a service of C. Then B can define a binding with parent scope and component name C to state that the service should be picked from C of its parent. An example is shown in `jadex.micro.testcases.semiautomatic.compositeservice`).
 - *upwards*: The upwards only searches upwards from the component towards the platform, i.e. root node.

- *dynamic*: If declared as dynamic, static is default, each call to a *getRequiredService(s)* method of the service container will cause a new search. If the binding is not dynamic, the result from the first search will be cached and returned to subsequent invocations, too. In order to issue a new search for a static binding *getRequiredService(s)* can be called with the flag *rebind* set to true.
- *proxytype*: The proxytype defines how calls are handled on the caller side with respect to interceptors. The possible values are *raw* for direct call routing and no interceptors and *decoupled* (default) for including the decoupling interceptor. If *direct* is specified the interceptor chain is built but the decoupling interceptor is not used. Decoupling means that a service call result is shifted back to the component thread that called the service.
- *create*: If enabled a component is created when no service of the given type could be found. The create flag requires the *componenttype* property to be specified in order to know what kind of component need to be started.
- *recover*(experimental): If enabled each service invocation is tried to be recovered in specific error cases. Currently, the *ComponentTerminatedException* and *ServiceInvalidException* are caught and transparently a new search is performed. The call is automatically tried again on the new service. This feature is experimental because there are some known issues with it. For example, the search currently can return the failed service again.
- *componentname*: The component name is used to directly reference a service of a known component, e.g. a subcomponent. Using the componentname and parent or local scope it is possible to link to services of peer or subcomponents.
- *componenttype*: The component type is currently used for two purposes (will be changed). First, it is used as search type in the same way as the component name. This allows for type level binding to peer or subcomponents. Furthermore, it is currently used as type for component creation if the create property is enabled.
- *interceptors*: The interceptors are used for performing actions before and after service calls. The provider as well as the consumer side may have their own interceptor chains. If custom interceptors are defined they are positioned after the default interceptors, which are the decoupling (turn off via proxytype=direct) and recover (turn on via recover=true) interceptors.

Provided Services



Provided service type

Provided services describe the functionalities offered by a component. The specification of a provided service consists of a service definition and an implementation definition.

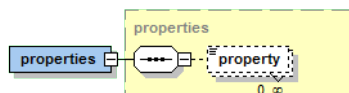
A provided service is defined using the following properties:

- *name*: The optional name can be used to refer to the provided service e.g. to override its implementation in a configuration.
- *type*: The interface type of the service.

In most cases a service definition should include an implementation because otherwise the component cannot create and provide the service at startup. It is possible to omit the implementation and supply it within a configuration. Instead of a direct implementation a provided service can also delegate the implementation to another component by using a binding. The implementation has the these properties:

- *class*: The implementation class. Needs to have an empty constructor.
- *expression*: Can be used if the implementation class cannot have an empty constructor, e.g. because it needs arguments. The creation expression can be an arbitrary Java expression.
- *proxytype*: The proxytype of the service. It has the same meaning as in the binding explained above. The possible values are *decoupled*, *direct* and *raw*. The default interceptors are the *DecouplingInterceptor* to execute the service call on the client component, the *ValidationInterceptor* to check if the service is initialized (returns a *ServiceInvalidException* otherwise), the *ResolveInterceptor* to route service calls if it is a Pojo service and the *MethodInvocationInterceptor* to finally perform the call.

Properties



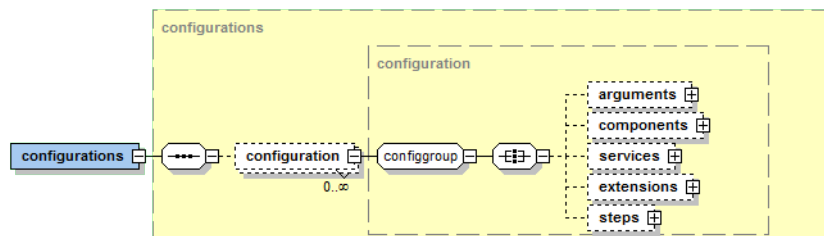
Properties

Properties can be used to define specific aspects of components. A property is only evaluated once at startup of the component. If the type of a property is *IFuture* the component will evaluate it to the underlying value, i.e. the component will wait for the property to be initialized.

A property has the following attributes:

- *name*: The property name.
- *class*: The property type. Please note the special handling of future properties described above.
- *expression*: The property value Java expression.

Configurations



Configurations

Configurations can be used to define different runtime settings of a component (a component may define an arbitrary number of different configurations). Each configuration has a name and at startup this name can be used to start the component in the underlying configurations. Configurations may differ in all runtime aspects from the type specification as can be seen also in the figure below.

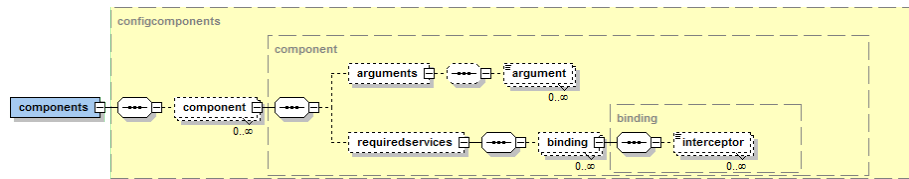
The properties of a configuration are as follows:

- *name*: The configuration name that can be used to refer to this specific setting. This name needs to be provided at startup of a component to activate the given configuration.
- *suspend*: If enabled the component will be started in suspended mode. This is e.g. helpful for debugging purposes.
- *master*: Starts the component as master. If a master component is terminated this causes the parent of the master also to terminate.
- *daemon*: The daemon setting is used in combination with *autoshtutdown*. If a component is started as daemon it does not prevent the parent from being terminated after the last non-daemon subcomponent has terminated.
- *autoshtutdown*: In enabled, automatically terminates the component when the last child (non-daemon) has been killed.

Arguments

- *arguments*: The arguments section allow for defining new values for arguments and results that possibly override the default values specified at the type level. If arguments are passed from the outside to the component these values always have precedence over configuration or type level values. An argument or result type is referenced via its name.

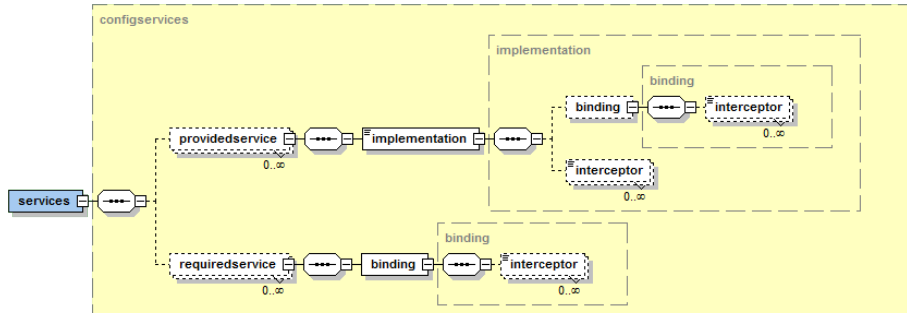
Components



Configuration components

- *components*: The components section allows for creating component instances of specified subcomponent types. These component instances can be customized by the follow properties:
 - *type*: The local component type that is one of the names of the declared subcomponent types.
 - *name*: The optional instance name for the component. The name can be an expression. If a number of agents is created (see next property) using $\$n$ can be used as predefined expression variable that contains the current number of the instance.
 - *number*: The number of components that should be started. The number is allowed to be a Java expression.
 - *configuration*: The configuration name the subcomponent will be started with.
 - *suspend*: If set to true the subcomponent will be started in suspended mode.
 - *master*: If set to true the subcomponent will be started as master. If a master subcomponent is terminated the parent will be terminated as well.
 - *daemon*: If set to true the subcomponent will be started as daemon. A daemon subcomponent will not hinder the autoshutdown of the parent component (if the parent has autoshutdown set to true).
 - *shutdown*: If autoshutdown is true the component counts the number of subcomponents. If the last subcomponent is terminated the parent component will be shutdown.
 - *arguments*: The arguments for the subcomponent can be defined.
 - *required services*: Using the name of the required services the corresponding binding can be overridden. The is e.g. helpful to statically link subcomponent services with parent or peer services.

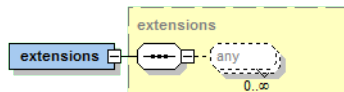
Services



Configuration Services

In the configuration of a component also the service details can be changed. With respect to provided services the implementation and with regard to required services the binding can be changed. The specifications require that a required or provided service is identified via its type name. The implementation or binding details can be redefined for the referenced service.

Extensions

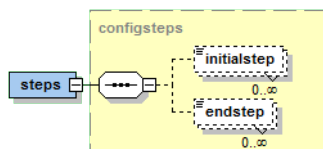


Extensions

Extensions can be used to add extension instance elements to an active component. As can be seen in the figure the concrete extension elements depend on the extension type used. Examples for extensions that make use of the extension mechanism are the virtual environment EnvSupport used in many example applications and AGR (the agent-group-role model). Please also refer to the Extension Types section below for further explanations.

(Current limitation: extensions cannot be used in annotation based Java components)

Steps



Steps

Steps can be used to execute custom behavior at component creation or termination. The first can be achieved using *initial steps* and the latter using *end steps*.

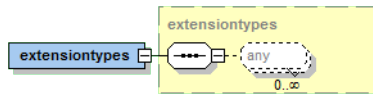
Each step is defined via:

- *class*: The class is a reference to a Java class that has to implement the generic *jadex.bridge.IComponentStep* interface. Each component has to realize a method called *execute* that should contain the corresponding domain logic. In order to access the component internals, the *IInternalAccess* interface of the component is passed as parameter value to the *execute* method. The internal access allows unprotected access to the internals. This is possible because it is ensured by the runtime infrastructure that steps are always executed on the component thread itself. The method must return a future that indicates when the step has been executed. The generic type of the step class can be used to adjust its concrete return type.

```
public interface IComponentStep<T>
{
    public IFuture<T> execute(IInternalAccess ia);
}
```

(Current difference: Micro agents have lifecycle methods instead of steps to execute behavior at creation and deletion time)

Extension Types



Extension types

Extensions can be used to add new functionality to an active component. An extension consists of the following aspects:

- An loader extension for the component factory that is provided by the *IComponentFactoryExtensionService* interface from package *jadex.bridge.service.types.factory*.
- An extension model instance elements that implement the *IExtensionInfo* interface from package *jadex.bridge.modelinfo*.
- The extension logic as class that implements the *IExtensionInstance* interface from package *jadex.bridge.modelinfo*.

A component factory searches for all component factory extension services and asks them for factory dependent loader information. This information (in case of XML-based components typically a set of *TypeInfo* Objects for the Jadex XML

reader) that is used by the factory to load the extension specific parts of the model. When creating a component instance the component interpreter expects that the instance elements defined in the configurations implement the interface *IExtensionInfo*. The interpreter use the method *createInstance()* on the objects to asynchronously create an extension instance of type *IExtensionInstance*. This instance is managed by the interpreter under the name defined in the extension info, i.e. *getExtension()* can be called on the interpreter to fetch an extension instance per name. The interface of an extension instance only allows to terminate the running extension.

```
public interface IComponentFactoryExtensionService
{
    public IFuture getExtension(String componenttype);
}
```

The component factory extension service is used by component factories to fetch extension loader functionality. The *getExtension()* method is called by a component factory to retrieve loader functionality for the extension. The component type parameter describes the type of component the factory is responsible for. To activate an extension mechanism a component instance has to be created that provides the extension service.

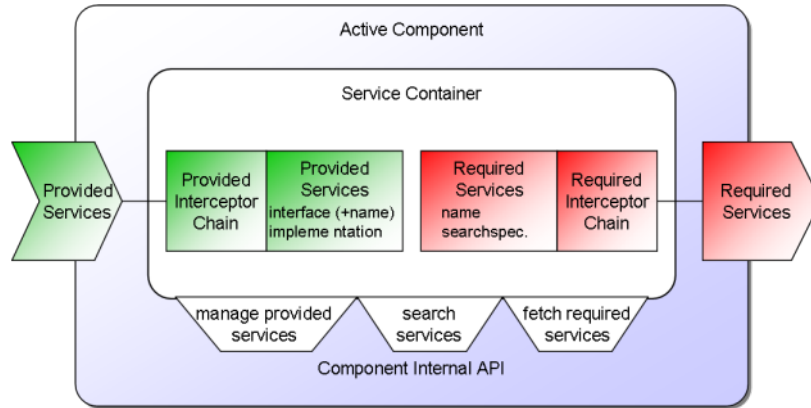
```
public interface IExtensionInfo
{
    public String getName();
    public IFuture<IExtensionInstance> createInstance(IExternalAccess access, IValueFetcher f
}
```

The extension info interface allows for creating an extension instance of a given name.

```
public interface IExtensionInstance
{
    public IFuture<Void> terminate();
}
```

An extension instance has a method that terminates the extension.

Chapter 5 - Services



Service container

The management of services is crucial for understanding how active components work. Each component can be seen as an autonomous service provider that offers services for other components. In case it needs services it can either use statically bound or dynamically searched services. Each component has a *service container* that should be used for all service management activities. It allows for fetching required services, explicitly searching for services at runtime and also for adding, removing or exchanging provided services. From within a component the service container can be fetched using `getServiceContainer()`. The interface of the container is `jadex.bridge.service.IServiceContainer`. The most important methods of the service container are shown in the following three code snippets:

```
public interface IServiceContainer
{
    public IFuture<Void> addService(IInternalService service, ProvidedServiceInfo info);
    public IFuture<Void> removeService(IServiceIdentifier sid);
    public IService getProvidedService(String name);
    public IService[] getProvidedServices(Class clazz);

    public <T> IFuture<T> getRequiredService(String name);
    public <T> IIntermediateFuture<T> getRequiredServices(String name);
    public <T> IFuture<T> getRequiredService(String name, boolean rebind);
    public <T> IIntermediateFuture<T> getRequiredServices(String name, boolean rebind);

    public <T> IFuture<T> getService(Class<T> type, IComponentIdentifier cid);
    public <T> IFuture<T> searchService(Class<T> type);
    public <T> IFuture<T> searchService(Class<T> type, String scope);
    public <T> IIntermediateFuture<T> searchServices(Class<T> type);
    public <T> IIntermediateFuture<T> searchServices(Class<T> type, String scope);
}
```

```

    public void addInterceptor(IServiceInvocationInterceptor interceptor, Object service, int
    public void removeInterceptor(IServiceInvocationInterceptor interceptor, Object service);
    public IServiceInvocationInterceptor[] getInterceptors(Object service);
}

```

Provided Services

The first block of methods allows for fetching, adding and removing provided services. It has to be noted that, despite it is possible to use add or remove services at runtime, one should be aware that it is often better from a software engineering point of view when components comply to their component type specifications and do not change their exposed behavior dynamically. If you want to add or remove provided services to a component it might also be easier to use corresponding methods directly on the component interpreter (cf. the `DynamicServiceAgent` example in package `jadex.micro.testcases.semiautomatic`). The methods of the container expect a readily configured service instance which is typically not the same as a user created service instance.

Required Services

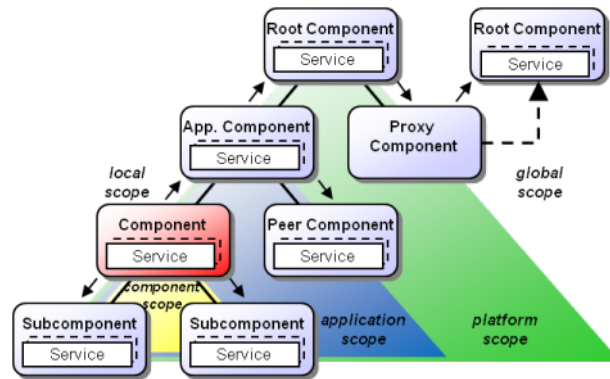
The second block of methods shows how required services can be used. There is a method to fetch a single service `getRequiredService()` and another one for fetching multiple services (those that have been defined with `multiple=true`) `getRequiredServices()`. Both methods come in two variants. The first variant only requires the `name` as argument. The second variant includes a `rebind` flag that allows for initiating a fresh binding by searching again the services according to the required service definition. This is only required if the service definition is static, i.e. `dynamic=false` which is the default.

Service Search

The third block contains methods that can be used to dynamically locate methods at runtime. The first `getService()` method is a convenience method to fetch a service of a given interface type from one exactly known other component. Hence the parameters are the interface type and the component identifier of the target component. The following two `searchService()` methods can be used to search for a service via its interface and an optional search scope. If no search scope is given, the default is application scope, i.e. all sub and super components within the application are included within the search. The same kinds of methods are available for multi services via the next two `searchServices()` methods.

The search scope defines the area of the search and is per default set to application. This means that only components within the started application are considered

within the search. In the figure below a visual representation of the most relevant search scopes is given. Static constants for search scopes are available via the class *RequiredServiceInfo*.

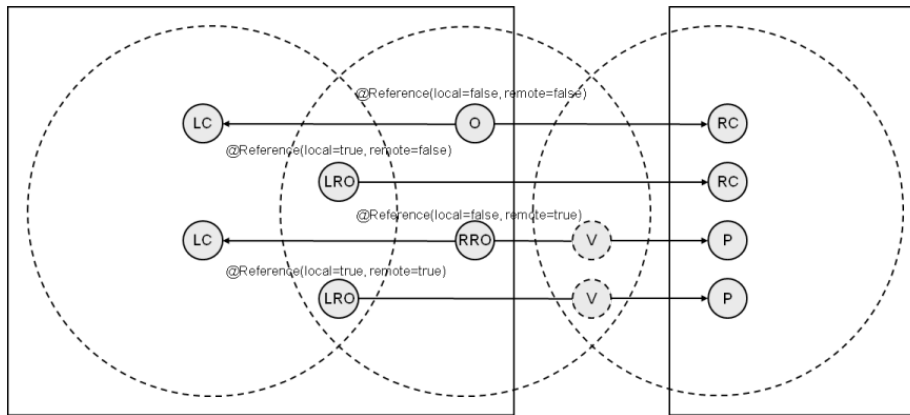


Search scopes

Interceptor Handling

The fourth block is meant for runtime interceptor management of service. These methods can be used to add, remove and inspect interceptors for specific services. It has to be noted that interceptors are not supported for all required and for provided services which are not declared to have *proxytype raw*. Such provided services are directly invoked from the caller without running through the interceptor chain. The first method *addInterceptor()* can be used to add an interceptor at a specific position into the interceptor chain. If the interceptor should be placed at the end of the chain (before the real invocation occurs) the position can be set to -1. The second method *removeInterceptor()* can be used to remove a specific interceptor from the chain and the third method *getInterceptors()* deliver the interceptor chain for a service as array. These array can be inspected e.g. to decide at which position a new interceptor should be placed.

Parameter Passing



Parameter passing semantics

An important aspect of active components is their isolation regarding state and execution, i.e. one component should not interfere with any other component. In order to realize isolation regarding the component state, parameter passing in service calls has a specific semantics. To avoid data inconsistencies due to concurrent access on data, all data is passed with **call-by-copy** semantics regardless if the invoked service is on the same or another platform. As call-by-copy induces an unwanted performance penalty it is also possible to adjust the parameter semantics in a fine-granular way. On the one hand the semantics can be defined based on the underlying parameter types and on the other hand also method specific parameter declarations can be used. In any case the `@Reference` annotation is used (package `jadex.bridge.service.annotation`). The annotation allows for defining the local and remote invocation semantics, i.e. you can e.g. define a class to be handed over per reference at local service calls and be copied at remote service calls (`@Reference(local=true, remote=false)`).

In the Figure above the four different cases of reference settings are illustrated from the viewpoint of three different active components (circles). The middle component invokes a service locally (left) and remotely (right) and it is shown how the data in the middle is treated.

- **@Reference(local=false, remote=false)** In this case the parameter object is copied locally and remotely leading to two clones (LC=local copy, and RC=remote copy). This case is the default.
- **@Reference(local=true, remote=false)** In this case the parameter object is copied only remotely and locally a reference to the same object is used (LRO=locally reference object and RC=remote copy). This case is useful for speeding up handling of immutable objects.
- **@Reference(local=false, remote=true)** Here, the parameter object is copied locally and remotely reference semantics is used. This is only possible if a remote proxy of the object can be created. It has to imple-

ment an interface that extends the `jadex.commons.IRemotable` marker interface. The proxy will be created according to that remotable interface. As a result a local copy and a virtually remotely shared object are created (LC=local copy, RRO=remotely referenced object, V=virtual object, P=proxy object).

- **@Reference(local=true, remote=true)** In this case the parameter object is not copied. Thus, it acts as local reference for the left and middle component and as remote reference for the middle and right component (LRO=local reference object, V=virtual object, P=proxy object). The semantics of this case also applies to active component services.

Hints:

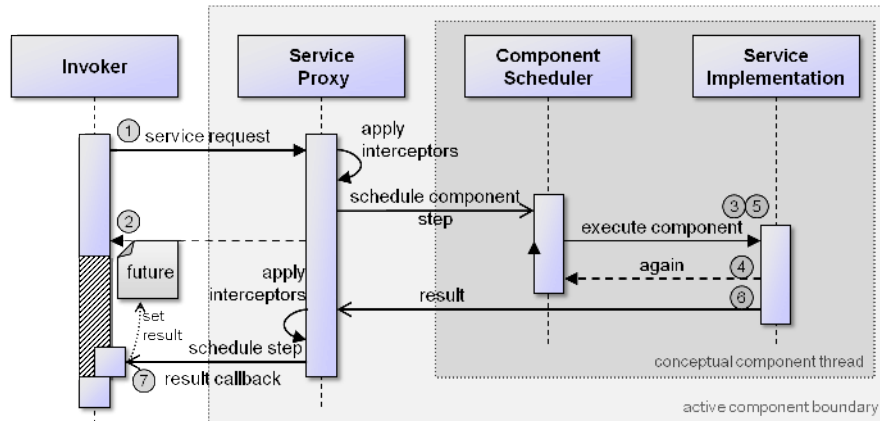
- When an object is immutable per definition (example an invoice that will never change in future), it should be declared as local reference to avoid unnecessary copying.
- In most cases remote references should be avoidable. The most common use case is a remote listener object that automatically notifies the listeners on the caller side. In the Jadex programming model such listeners in many cases can be replaced by using subscription futures as return values (subscription futures do not save intermediate results).
- **The caller of a service call needs not to be passed explicitly as parameter value.** Instead at the beginning of the service method (i.e. before other services have been invoked or component steps have been performed), *ServiceCall.getInstance().getCaller()* can be used to retrieve the component id of the caller component.

Concurrency

The concurrency model of active components ensures that every component and thus also each service of a component is called in a thread-safe manner, i.e. a service developer can always assume that the service is exclusively executed by the component without interference of other concurrent calls. This assumption means that it is normally not necessary to use any synchronization mechanisms like monitors or synchronized methods within service implementations. If you intentionally do not want a thread-safe service you can configure with ‘proxy-type=raw’, which means that the implementation class will be directly called. In this case the developer has to ensure that concurrent access will not harm the service and/or component state consistency. Another important aspect is that a service implementation might use asynchronous methods until it completes and returns the result. This implies that a service call may be executed interleaved with other service calls regarding the steps its consists of.

Whenever a (non-raw) service is called from another component thread switching is automatically performed, i.e. the call is first decoupled from the caller and executed on the thread of the receiving component. When the callee finishes

processing and ends the method by setting the return value again a thread switch is automatically performed to ensure that the result listener is notified on the thread of the original caller. This scheme is changed if specific thread switching listeners are used, e.g. if the user uses a `SwingDefaultResultListener`, the call will be received on the swing thread.



Invocation scheme

The general service invocation scheme is also visualized in the figure above. It can be seen how a service invocation is processed step by step. First, the service request reaches the service proxy (that looks like the original service for the service user), which applies the interceptors one by one. The invocation is scheduled as a new component step on the component scheduler. Thereafter, the original call returns to the invoker and presents a future to it. The future represents the placeholder object for the real result that is made available via the callee. The scheduler possibly awakens the component by initiating its execution. The component then may process several steps until the service call result could be determined and is handed over to the service proxy, which in turn applies all interceptors in the opposite direction. Finally, the service proxy will schedule a step on the original invoker to set the callback result in the listener. This ensures that the user will perceive the result on the invoker's component thread.

Contract Oriented Programming

Programming by contract is a well-known approach that focusses on the relationships between suppliers and clients. In the active components approach contracts between service providers and service consumers can be defined and automatically monitored. The basic idea is to allow **pre-** and **postconditions** to be stated for service interfaces. These conditions can be expressed in terms of different kinds of annotations attached to method parameters and/or the return value and are automatically checked by the Jadex infrastructure, i.e. the

conditions need not be manually evaluated at the beginning or end of a service call. Jadex simply uses specific condition interceptors before and after a service call to ensure that conditions hold. If that is not the case a runtime exception is raised and returned to the caller. Currently, the following types of conditions are supported:

- **@CheckNotNull:** As the name suggests, this annotation ensures that the corresponding parameter or result value is different from null.
- **@CheckIndex:** Precondition for checking if the argument is a valid index. This annotation needs a further value that states in which parameter the collection is given, to which the index should be checked against. The annotation works for all kinds of elements that can be somehow iterated over (e.g. arrays or collections)
- **@CheckState:** Allows a Java expression to be evaluated. Reserved variables are \$arg for the current argument and \$arg0 - \$argn for other arguments. In case of a post condition the result is available via \$res and intermediate results via \$res[0], \$res[-1], etc.

An example application (from package `jadex.micro.testcases.prepostconditions`) further illustrates how conditions can be specified:

```
public interface IContractService
{
    public @CheckNotNull @CheckState("$res>0 && $res<100") IFuture<Integer> doSomething(
        @CheckNotNull String a,
        @CheckState("$arg>0 && $arg<100") int x,
        @CheckState("$arg>0") int y);

    public IFuture<String> getName(@CheckIndex(1) int idx, @CheckNotNull List<String> names);

    public @CheckState(value="$res[-1] < $res", intermediate=true) IIntermediateFuture<Integer>
}
}
```

The example interface definition shows how the different conditions can be used:

- The first method `doSomething()` takes three parameters (a, x, y) and expects that a never nulls, x is between 0 and 100 and y is greater 0. Furthermore, the result value of the method must never be null.
- The second method `getName()` works with an index (idx) and a collection (names). The precondition ensures that the index is valid according to the referenced collection (the 1 argument in `@CheckIndex(1)`), i.e. `idx>=0` and `idx<size of the collection`.
- The third method `getIncreasingValue()` shows how postconditions can be used with intermediate results. In the example it is safeguarded that the method delivers only monotonically increasing values (`$res[-1] < $res`).

Streams

Streams are a convenient and versatile programming concept in Java. Basically, the idea is that a stream can be opened and data can be either written to it (output stream) or read from it (input stream) step by step, i.e. the data producer and data consumer are logically separated. This concept is ideal also for distributed systems, in which a data producer may exist on another host than the consumer. Jadex offers a dedicated streaming API that allows for high-level programming with streams. Originally, it was planned to directly support the Java streaming API with new classes for the distributed case. Regrettably, this was not easily possible, because a) Java does not introduce interfaces for streams and more importantly b) has a blocking API, which does not fit well to principally non-blocking active components. For these reasons, the Jadex streaming API is based on a new set of interfaces (and classes) shown in the figure below. It can be seen that those interfaces resemble the original Java classes (*InputStream* / *OutputStream*) to a high degree, but additionally take into account the new possibilities by future based return values.

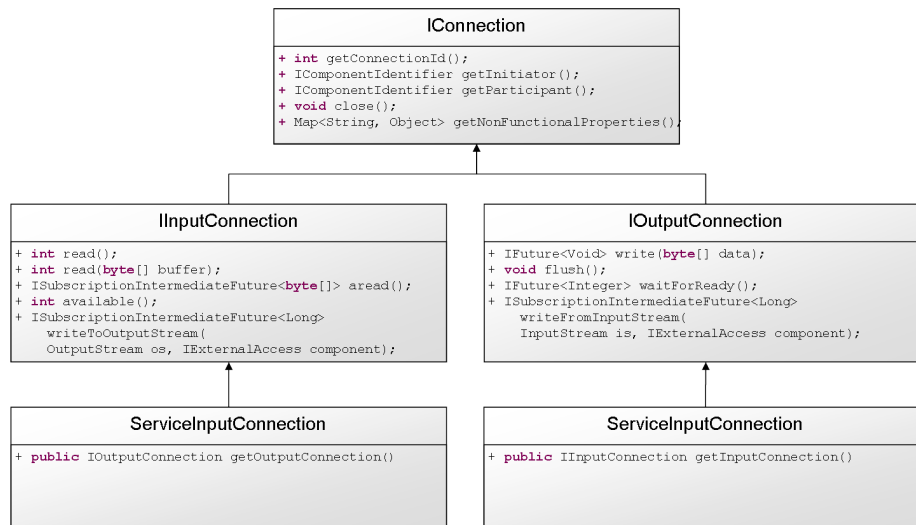


Figure 49:

Overview of important stream interfaces and classes

The base interface for all types of Jadex streams is ***IConnection***, which mainly provides methods to get general information about the stream (such as the connection id, the initiator and participant component ids). From this interface ***IInputConnection*** and ***IOutputConnection*** are derived. These coarsely correspond to the Java *InputStream* and *OutputStream* and offer functionalities to read and write data. Finally, the concrete classes ***ServiceInputConnection*** and ***ServiceOutputConnection*** implement those interfaces and additionally

add one important method to get a corresponding output for input connection and vice versa. The idea is that a service connection can be created within a service implementation of a component. The original service connection can then be used to e.g. write or read data and the derived opposite connection can be passed as parameter value in a method call or can be passed back as return value. In this way two services can communicate via streams. In Jadex streaming support is based on two different APIs, a low and a high level one as described next.

Low-Level Streaming API

The low-level streaming API directly makes use of the *IMessageService* of the platform. Besides sending single messages, the message service also allows for creating **virtual streams**. The streams exist between two component instances and can be used to transport binary data between them. The streams are virtual as different real connections may be used to bring the data from the source to the target. The binary data will be handled in small packets in the same way as ordinary messages, i.e. depending on the destination different transport mechanisms can be used to send the data. The message service hence multiplexes the binary streams in time and space leading to the following benefits:

- The platform does not need to open any extra ports for handling streaming data
- The platform can switch transports if connection settings change during the transmission, i.e. an underlying aborted tcp connection does not harm the stream if another connection to the target exists

The low-level API provides two basic functionalities that allow for creating a stream, and retrieving a stream from another component. Creating a stream can be done by calling one of the following two methods on the *IMessageService*:

```
public IFuture<IOutputConnection> createOutputConnection(IComponentIdentifier sender,  
    IComponentIdentifier receiver, Map<String, Object> nonfunc);
```

```
public IFuture<IInputConnection> createInputConnection(IComponentIdentifier sender,  
    IComponentIdentifier receiver, Map<String, Object> nonfunc);
```

The component that receives the stream is notified about the new stream arrival. How this is exactly performed depends on the concrete component type and the way the interpreter forwards the (streamArrived()) call.

Micro Agent

In case of a micro agent notification is done via a simple callback method. In case a pojo agent is used, the method that is annotated with *@AgentStreamArrived* is called with the *IConnection* as method parameter as shown below.

```

@AgentStreamArrived
public void newStream(IOutputConnection con)
{
    System.out.println("received new connection: "+con);
}

```

In case the agent extends `MicroAgent` directly, the following method can be overridden to place custom code that is executed when a new stream arrives:

```

public void streamArrived(IConnection con)
{
    System.out.println("received new connection: "+con);
}

```

BDI Agent

A BDI agent can receive in the same way as a message. Internally, the interpreter just creates a new FIPA message map and sets the sender and receiver ids to the received stream sender and receiver. Furthermore, it puts the stream as content in the fake message. To be able to receive a stream at the application level an appropriate message event template has to be declared in the agent (or capability). In order to distinguish a stream message from other kinds of messages a match expression can be used that tests if the content is a connection (using e.g. `$content instanceof IConnection`). The message event template can then be used as trigger for plans as usual.

BPMN Process

A bpmn process can receive a stream also very similar to a message using a corresponding catching message event. In this case the wait filter of the catching event has to be programmed to react to a connection. After having received a stream via a message event element, `$event` can be used to fetch the stream in the next activity. A corresponding filter might look like the following:

```

IFilter fil = new IFilter()
{
    public boolean filter(Object obj)
    {
        return obj instanceof IConnection;
    }
}

```

Other component types

XML component types do not possess behavior specifications so that they do not possess a way to react to an incoming stream directly. Yet, service

implementations of XML components (as well as any other component types) may use the high-level streaming API described next.

High-Level Streaming API

The high-level streaming API deals with streams as parameters and return values of service calls. This enables to pass a stream directly to another component, so that one side can write and the other one read data.

In general the high-level streaming API allows to:

- Pass *IInputConnection* and *IOutputConnection* as parameters of services
- Pass *IInputConnection* and *IOutputConnection* as return value of services
- Offers *ServiceInputConnection* and *ServiceOutputConnection* to create pipes between services (using `getInputConnection()`, `getOutputConnection()`)
- Makes reading/writing data from/to Java streams very easy by creating pipes (`writeFromInputStream()`, `writeToOutPutStream()`)

In order to make this scheme more transparent a small example will be explained. The basic assumption is that there is a service that needs an input stream as parameter to fetch some data. The service could look like this:

```
public interface IExampleService
{
    public IFuture<Void> m1(IInputStream is);
}
```

This service is invoked from a service of some other component. This component wants to provide the content of a specific file as input stream to the service. To achieve this the following steps need to be performed:

- Create a service output connection using

```
ServiceOutputConnection ocon = new ServiceOutputConnection();
```

- Fetch the corresponding input connection using

```
IInputConnection icon = con.getInputConnection();
```

- Call the service method and provide the input connection

```
service.m1(icon)
```

- Start writing data to the output connection. This can be performed manually by calling

```
public IFuture<Void> write(byte[] data)
```

for each piece of data that should be transferred. In addition a service output method also offers a convenience method that allows to pipe all data from a Java input stream to the Jadex output stream. This comes very handy in the example where a file should be transferred. For this purpose it is sufficient to do the following.

```
FileInputStream fis = new FileInputStream(new File("somefile.data"));  
service.writeFromInputStream(fis, exta);
```

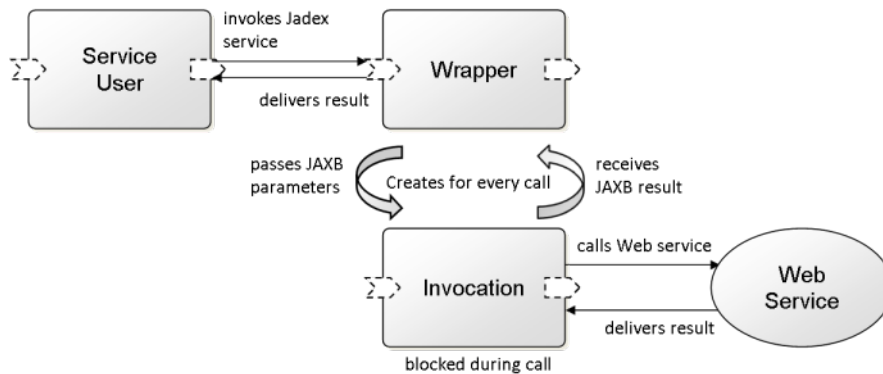
Chapter 6 - Web Service Integration

In this chapter it will be explained how existing standard web technologies can be used in concert with Jadex active components. Jadex allows for seamless usage of existing web services as well as publishing Jadex services with minimal effort in the web. Classical WSDL-based web services as well as RESTful web services are supported. In order to use the web service support the Jadex module *jadex-platform-extension-webservice* has to be included. Example applications that make use of the presented techniques can be found in the package *jadex-applications-webservices*. In order to publish WSDL or RESTful services the platform has to provide corresponding publish services. The default publish implementations can be automatically started at startup of the platform by using the arguments *-wspublish true* and *-rspublish true* respectively.

Integration Concept

The integration of Jadex components with web services includes two directions. The first deals with using existing web services with Jadex applications in a transparent manner. The latter considers how Jadex services can be made available as web services also for non-Jadex users.

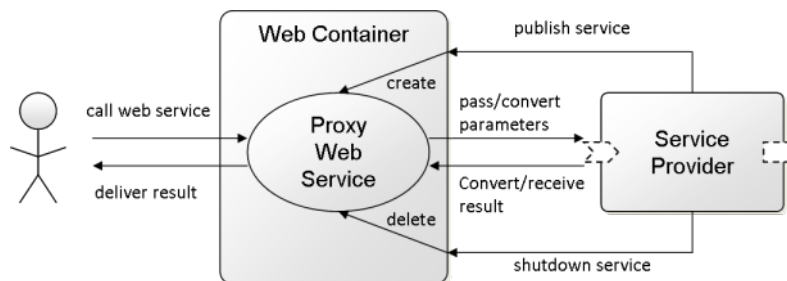
Integrating Existing Web Services



Web Service Invocation Architecture

The objective of the web service call integration consists in integrating an external web service as normal Jadex component service within the Jadex platform. This allows for using the web service in the same way as other Jadex services, i.e. it can be used as required service or dynamically searched in other components. The general approach is sketched in the figure above. It can be seen that a Jadex wrapper agent is used to offer the web service as Jadex service. This Jadex service offers an asynchronous variant of the synchronous web service. The wrapper agent forwards service calls to the web service and returns the results to the caller. As the web service call is synchronous and therefore blocking it has to be ensured that the wrapper agents can handle multiple calls concurrently. This is achieved by creating a new invocation component for each call. This invocation component only perform the web service invocation, is blocked during the call, and afterwards returns the results to the wrapper component and terminates.

Publishing Jadex Services as Web Services



Web Service Publish Architecture

Publishing Jadex services as web service has the underlying idea that one can easily make available Jadex selected services in a standards compliant manner. The Jadex publishing mechanism is inspired by the same mechanism in SCA. For each provided service within a component it can be specified if and how this service should be published. After starting a service (usually at component

startup) this publishing information is inspected and a corresponding publishing service (according to the publishing type) is invoked in order to publish the service. In the same way, at shutdown of the service the publish service is requested to end publishing. Publishing a service as web service typically means that some kind of web container is instructed to listen at a given url and handle web service requests with a created web service implementation. When using the default WSDL or RESTful publishing mechanisms a proxy service component is created within the web container, which hands over requests towards the backend Jadex service.

At the moment of publishing the Jadex interpreter searches for all available *IPublishServices*. In order to add new kinds of publishing or use alternative publishing mechanisms just the available publish services at the platform need to be changed. This interface contains the following methods:

```
public interface IPublishService
{
    public IFuture<Boolean> isSupported(String publishtype);

    public IFuture<Void> publishService(ClassLoader cl, IService service, PublishInfo pi);

    public IFuture<Void> unpublishService(IServiceIdentifier sid);
}
```

All these methods are automatically called by Jadex. The *isSupported()* method is used to check whether the publish service is able to publish the given type. If yes, the component interpreter will call the *publishService()* method in order to publish the service on the endpoint. Here the logic has to be filled in which knows the concrete endpoint type and can supply it with the necessary artifacts to make the web service available. The parameters include the current classloader, the Jadex service to which web service requests should be forwarded and a publish info object that represents just a struct with information about the publishing properties (publishtype, publishid and servicetype). The last method is called when the service is terminated, e.g. if the component is killed. Here the service identifier is passed as parameter in order to find the service and instruct the endpoint to stop the web service.

WSDL Web Services

Integrating Existing WSDL Web Services

A Jadex wrapper agent needs to be created that provides access to the external web service as a separate Jadex component service. The interfaces of both services differ with respect to the return value of the method signatures. The return value of the Jadex service interface always has to be a *IFuture<originaltype>* that contains the original type as generic part. This has to be done to make

the synchronous web service asynchronous. The input parameters of methods directly correspond to the generated parameter types coming from the Java included *wsimport* tool, which itself relies on *JAXB*. Service calls on the Jadex service will be forwarded from the wrapper agent to the original web service and the result will be returned to the caller. In order to avoid that the wrapper agent is blocked by the synchronous web service call and cannot process any further invocations, internally a subcomponent is created for each call. This subcomponent will be automatically deleted immediately after the call has returned.

Implementation steps:

- **The Java web service classes and data types have to be generated using *wsimport*** (usually available in the *bin* directory of the JDK). The most important options are: `-keep` for not deleting the source files, `-p` for specifying the target package (example: `wsimport -keep -p jadex.micro.examples.ws.geoip.gen http://www.websvicex.net/geopervice.asmx?WSDL` `](http://www.websvicex.net/geopervice.asmx?WSDL)`). The following block shows the service interface as generated by the *wsimport* tool. The tool automatically includes all the annotations required by the JAX-WS framework that allows publishing and invoking web services seamlessly from Java. Between all the annotations, you can see that the interface defined two methods: *getGeoIP(ipAddress)* and *getGeoIPContext()*.

```
@WebService(name = "GeoIPServiceSoap", targetNamespace = "http://www.websvicex.net/")
@XmlSeeAlso({ObjectFactory.class})
public interface GeoIPServiceSoap
{
    @WebMethod(operationName = "GetGeoIP", action = "http://www.websvicex.net/GetGeoIP")
    @WebResult(name = "GetGeoIPResult", targetNamespace = "http://www.websvicex.net/")
    @RequestWrapper(localName = "GetGeoIP", targetNamespace = "http://www.websvicex.net/", class = Object.class)
    @ResponseWrapper(localName = "GetGeoIPResponse", targetNamespace = "http://www.websvicex.net/", class = Object.class)
    public GeoIP getGeoIP(@WebParam(name = "IPAddress", targetNamespace = "http://www.websvicex.net/") String ipAddress);

    @WebMethod(operationName = "GetGeoIPContext", action = "http://www.websvicex.net/GetGeoIPContext")
    @WebResult(name = "GetGeoIPContextResult", targetNamespace = "http://www.websvicex.net/")
    @RequestWrapper(localName = "GetGeoIPContext", targetNamespace = "http://www.websvicex.net/", class = Object.class)
    @ResponseWrapper(localName = "GetGeoIPContextResponse", targetNamespace = "http://www.websvicex.net/", class = Object.class)
    public GeoIP getGeoIPContext();
}
```

- **The Jadex service interface has to be defined** on basis of the generated service interface class and WSDL. Return values have to be encapsulated into *IFuture* types. You can see below that the additional interface for the Jadex service declares the same methods as the interface generated from the WSDL, but changes the return types by wrapping the result

objects in corresponding futures.

```
public interface IGeoIPService
{
    public IFuture<GeoIP> getGeoIP(String ip);
    public IFuture<GeoIP> GetGeoIPContext();
}
```

- **The wrapper component has to be defined.** Therefore, e.g. a micro agent can be created, which extends *jadex.extension.ws.invoke.WebServiceAgent*. The new agent has to declare a provided service (using the *@ProvidedService* annotation). The implementation of the service should use the inherited method *createServiceImplementation(Class type, WebServiceMappingInfo mapping)*. As parameters the JAXB generated service class (*GeoIPService.class* in the example) and the name of the method for fetching the porttype (here *getGeoIPServiceSoap*) need to be passed. In case another component type should be used the provided service implementation should call the static method */createServiceImplementation(IInternalAccess agent, Class type, WebServiceMappingInfo mapping)* on *jadex.extension.ws.invoke.SWebService*. As first parameter the internal access of the agent needs to be passed. In xml type components it is available using the predefined *\$component* variable.

Example:

```
@Agent
@Imports({"jadex.extension.ws.invoke.*", "jadex.webservice.examples.ws.geoip.gen.*"})
@ProvidedServices(@ProvidedService(type=IGeoIPService.class,
    implementation=@Implementation(expression="$pojoagent.createServiceImplementation(
        IGeoIPService.class, new WebServiceMappingInfo(
            GeoIPService.class, \"getGeoIPServiceSoap\")"))))
public class GeoIPWebServiceAgent extends WebServiceAgent
{
}
```

Publishing Jadex Services as Web Services

It is also possible to make a Jadex component service accessible as Web Service. Jadex reuses existing publishing mechanisms for this purpose and encapsulates their functionality in services. For each provided service publishing information can be given by using the *@Publish* annotation (or in XML components the publish tag). This annotation allows for specifying:

- *publishtype*: The publish type helps identifying the way the service should be published. Currently, the only built-in publish type supported by Jadex

is *ws* for web service (in Java the constant `IPublishService.PUBLISH_WS` can be used).

- *publishid*: The id that is used to publish the service. In case of web service publishing the id should be the url under which the service will be made available in the infrastructure.
- *mapping*: The Java interface of the web service. This should be the synchronous variant of the Jadex service interface.

Publishing services works internally as follows. On service creation available platform services of type `IPublishService` from package `jadex.bridge.service.types.publish` will be searched and the first with a fitting publish type will be executed. This means that new custom publish services can be easily integrated by just starting components that offer new publish services implementing the aforementioned interface.

Steps for publishing a Jadex service:

- Implement a component service as usual including an asynchronous service interface and an implementation class. Additionally, the data classes for parameters have to be implemented. In the example shown below only the service interface and implementation are sketched:

```
public interface IBankingService
{
    public IFuture<AccountStatement> getAccountStatement(Request request);
}

@Service
public class BankingService implements IBankingService
{
    public IFuture<AccountStatement> getAccountStatement(Request request)
    {
        String[] data = new String[]{"Statement 1", "Statement 2", "Statement 3"};
        AccountStatement as = new AccountStatement(data, request);
        return new Future<AccountStatement>(as);
    }
}
```

- Write a synchronous variant of the Jadex service interface. This means that the web service return value types will change from `IFuture<type>` to just `type`. In the example it can be seen that that `IFuture<AccountStatement>` was adapted to `AccountStatement`.

```
public interface IWSBankingService
```

```

{
    public AccountStatement getAccountStatement(Request request);
}

```

- Add the `@Publish` annotation with values for the publishing url (publishid), the publishtype (`IPublishService.PUBLISH_WS` or just “ws”) and the servicetype (the interface created in the last step). In the example below the service is published at localhost with port 8080:

```

@Agent
@Imports({"jadex.extension.ws.publish.*", "jadex.webservice.examples.ws.offerquote.gen.*"})
@ProvidedServices(@ProvidedService(type=IBankingService.class,
    implementation=@Implementation(BankingService.class),
    publish=@Publish(publishtype=IPublishService.PUBLISH_WS, publishid="http://localhost:8080/"))
public class BankingAgent
{
}

```

- In order to test if publishing has work you can start the component and check if the WSDL is available at the publishing url with appended `?WSDL`. Again, `wsimport` can be used to generate the client sources. These can be utilized to invoke the web service and check if results are retrieved.

A current limitation is that only the Java internal web service endpoint can be used for publishing. Other publishing services that can deploy on other common infrastructures like e.g. Glassfish are not yet available. If a deployment is needed on another kind of endpoint a new publish service has to be created and provided by a component. Custom publish services need to implement the *IPublishService* already mentioned.

REST Web Services

Integrating Existing REST Web Services

In order to make existing REST web services usable inside of Jadex system, a wrapper agent needs to be defined, which offers a Jadex service that corresponds to the REST service. The wrapper agent uses a new invocation sub-agent for each incoming REST service call. It maps the Jadex service call to a suitable REST call and uses the invocation agent to execute the call. Parameters and the result value are converted if needed.

Implementation steps:

- **Jadex service interface specification.** Currently, the Jadex service interface has to be defined manually based on reading the documentation of the REST service that should be used. (It could also be an option to start with the Java `wadl2java` tool if the REST service offers a WADL

description and generate Java classes from it. A WADL description is the REST pendant to the WSDL description for SOAP based web services). The Jadex service interface should abstract away from the REST service syntax and use a clean object oriented style with typed parameters and return value (instead of only using String etc.). The reason is that the Jadex service should be designed in a way that makes it easy to use it from other Jadex components and services (these do not know that the Jadex service implementation is a web service).

As an example a small cutout of the Google chart API is used. In this case methods for creating images for bar, pie and line charts will be specified. The corresponding Jadex service interface is shown below. The reference annotation for result and color parameters is used to pass them via *call-by-reference* semantics. This is acceptable if parameters are considered immutable objects. Width and height describe the size of the resulting chart image in pixels, data represents the data values of possibly more than one data series, labels define description texts and colors specify how the different data series are visualized.

```
public interface IChartService
{
    public @Reference(local=true) IFuture<byte[]> getBarChart(int width, int height,
        double[][] data, String[] labels, @Reference(local=true) Color[] colors);

    public @Reference(local=true) IFuture<byte[]> getLineChart(int width, int height,
        double[][] data, String[] labels, @Reference(local=true) Color[] colors);

    public @Reference(local=true) IFuture<byte[]> getPieChart(int width, int height,
        double[][] data, String[] labels, @Reference(local=true) Color[] colors);
}
```

- **Defining the parameter and result mappings.** In the next step it has to be defined how the Jadex service calls are mapped to REST service calls. This is done by specifying a mapping file as Java interface. This interface should contain the same signatures as the original service interface but add annotations for the REST mappings to it. Currently the following annotations are supported (we have reused Jersey annotations as much as possible).
 - *@GET, @POST, @PUT, @HEAD, @DELETE, @OPTIONS*: The rest service type defines the http method that is used to perform the rest call.
 - *@Path*: The url path where the request should be targeted to. This annotation can be used at the class level as well as on individual methods. The target url is built from both.
 - *@Consumes*: The media type of the request parameters.
 - *@Produces*: The accepted media type the result should be delivered in.

- *@ParameterMapper, @ParametersMapper*: The Jadex service possibly has a completely different parameters compared to the REST web service. In order to convert parameters from Jadex to the required REST format parameter mappers can be used. These mappers allow for creating *n* (named) parameters from *m* incoming parameters. This is necessary because the REST service may require more or less parameters than the original one. The simplest way is to define a parameters mapper (*@ParametersMapper* at method) that is responsible for converting all parameters at once. This is simple but also incurs the drawback that the corresponding mapper class is highly application dependent and thus not very reusable. In order to be able to use simpler mappers also each parameter can be mapped on its own (*@ParameterMapper* at each parameter). It is also possible to create a parameter mapper that gets more than one input parameter. For this purpose the *source* attribute can be set to the numbers of the formal parameters that should be fed in. Furthermore, if parameters are needed in the REST call that are not present in the Jadex method interface, you can add one or more parameter mapper at the method itself. More than one additional parameter mapper have to be included into a *@ParameterMappers* container annotation.
- *@ResultMapper*: The result mapper annotation is used to map back the REST result to a parameter object. All parameter and result mappers have to implement the interface *IValueMapper* shown below:

```
public interface IValueMapper
{
    public Object convertValue(Object value) throws Exception;
}
```

In the chart example the following mapping is used:

```
@Path("https://chart.googleapis.com")
public interface IRSChartService
{
    @GET
    @Path("chart")
    @Produces(MediaType.APPLICATION_OCTET_STREAM)
    @ParameterMapper(value="cht", mapper=@Value("new ConstantStringMapper(\"bhs\")"))
    public @ResultMapper(@Value(clazz=ChartResultMapper.class)) IFuture<byte[]> getBarChart(
        @ParameterMapper(value="chs", mapper=@Value(clazz=SizeStringMapper.class), source={0,1}),
        @ParameterMapper(value="chd", mapper=@Value(
            "new IterableStringMapper(\"t:\",\"|\", null, new IterableStringMapper(\"|\",\"|\"))")) double[] data,
        @ParameterMapper(value="chl", mapper=@Value("new IterableStringMapper(\"|\",\"|\")) String[] labels,
        @ParameterMapper(value="chco", mapper=@Value(
            "new IterableStringMapper(\"|\",\"|\", new ColorStringMapper())")) Color[] colors);
```

...

It can be seen that the `getBarChart()` method is annotated to produce REST requests. The request URL is composed of the general path `https://chart.googleapis.com` (`https://chart.googleapis.com`) and the concrete method path `chart`. Furthermore, the http request type is set to `get` and the acceptable media type is set to binary. The rest of the mapping declaration deals with parameter conversion. Here, it is shown how a mapping per parameter can be done, i.e. for (most) of the parameters it is stated how they should be converted to a rest parameter. As the chart API requires the size to be given in the format `width:height` a chart specific `SizeStringMapper` class is used. Normally, a parameter mapper only gets its own parameter as input. If more are needed, these can be stated using the `source` attribute, which takes a set of integer values representing the absolute numbers of the parameters. The second parameter is used in the first mapper so that it does not declare its own mapper. The mapping of data series is a bit more complex. It requires a format that looks like the following example `t:s11,s12,s13/s21,s22/s31` The corresponding mapper is called `IterableStringMapper`, which iterates over some form of collection or array and concatenates the entries with a given separator. In addition a prefix (here `t:`) and postfix can be specified if needed. In order to map the two dimensional array the mapper can be equipped with a submapper, which it calls for each entry. In this way another mapper can be used to handle the inner series data. The labels are also mapped using an `IterableStringMapper`, which produces the format `label1/label2/label3` Finally, the colors are mapped to hex string values using a `ColorStringMapper`. It produces a format like `#rrggbb`, example `#FF2233`.

Using different mappers per parameter makes the mapper code simpler and facilitates reuse. On the other hand it leads to complex looking signatures. As an alternative also a parameters mapper can be used which takes all input parameters at once and produces the complete REST parameters. The equivalent method mapping of `getBarChart` is shown below:

```
@Path("https://chart.googleapis.com")
public interface IRSChartService
{
    @GET
    @Path("chart")
    @Produces(MediaType.APPLICATION_OCTET_STREAM)
    @ParametersMapper(@Value(clazz=ChartParameterMapper.class))
    @ResultMapper(@Value(clazz=ChartResultMapper.class))
    public IFuture<byte[]> getPieChart(int width, int height, double[] data, String[] labels)
    ...
}
```

- Specifying the wrapper component.


```

@Agent
@ProvidedServices(@ProvidedService(type=IChartService.class, implementation=@Implementation
    expression="$pojoagent.createServiceImplementation(IChartService.class, IRSChartService.c
public class ChartProviderAgent extends RestServiceAgent
{
}

```

The wrapper component is very simple and specifies the chart service as provided service. The implementation is an expression that takes the interface and mapping as arguments. Please note that the method *createServiceImplementation* is either available via the *RestServiceAgent* if it is extended or via the class *SRest* as static method. In this case the method takes the internal access of the component as further argument. After having started the component, its chart service can be searched and used as any other Jadex component service.

Publishing Jadex Services as REST Web Services

The publication of a Jadex service as RESTful web service can be used to make Jadex functionality available to external system users. As REST web service interfaces are quite different from object oriented service interfaces the publication can be customized to a high degree. The mapping from the Jadex service interface to the REST service interface can be done in the following ways:

- fully automatic: The REST publishing service will generate the REST interface directly from the Jadex service interface. It will use simple heuristics to decide about the mapping details.
- semi automatic: Additional mapping is specified using an annotated interface or annotated (abstract) class. The mapping information is annotated to the methods of the mapping interface and state how this method should be made available. The method signatures in the mapping file represent the REST view of the service, i.e. the parameters are expected REST input parameters. In case an interface is used, besides the REST call specifics it has to be specified to which Jadex service method the REST call should be delegated. If an (abstract) class is used the method body can also be provided directly so that no automatic service delegation needs to be done.
- manual: In manual mode the publishing service takes the mapping class directly as implementation and does not provide further delegation code.

The currently supported mapping annotations are (most of them are reused directly from Jersey JAX-RS):

- *@GET*, *@POST*, *@PUT*, *@HEAD*, *@DELETE*, *@OPTIONS*: The rest service type defines the http method that can be used to perform the rest call.
- *@Path*: The URL path to invoke the service method. This annotation can be used at the class/interface and at the method level. The complete path is retrieved by adding both parts.

- *@Consumes*: The consumable media types.
- *@Produces*: The produced media types.
- *@ParametersMapper*: The parameters mapper annotation is used to define how the REST input parameters should be transformed to Jadex service parameters. Please note that no single mapping of parameters is currently supported.
- *@ResultMapper*: The result mapper can be used to transform the result.

In addition to the annotation based mapping information the publish service can be instructed with several properties:

- *generate*: Boolean flag to state if the generator should add all methods from the Jadex interface to the REST interface. Default is true. If set to true, the generator will use the mapping information as provided.
- *generateinfo*: Boolean flag if an additional *getServiceInfo* method should be created. This REST methods return an auto generated HTML page with a section for each method. These methods can be directly invoked with parameters from the page. Default is true.
- *formats*: A string array that can be used to state the producible and consumable media types. This is helpful for automatic generation, i.e. if no mapping file is provided.

The steps to publish a Jadex service are as follows:

- Implement a component service as usual including an asynchronous service interface and an implementation class. Additionally, the data classes for parameters have to be implemented. In the example shown below only the service interface and implementation are outlined:

```
public interface IBankingService
{
    public IFuture<AccountStatement> getAccountStatement(Date begin, Date end);
}
```

The service implementation here is very simple and just returns some pseudo statements without considering the actual begin and end dates.

```
@Service
public class BankingService implements IBankingService
{
    protected List<String> data;

    @ServiceStart
    public void start()
    {
        data = new ArrayList<String>();
        data.add("Statement 1");
    }
}
```

```

        data.add("Statement 2");
    }

    public IFuture<AccountStatement> getAccountStatement(Date begin, Date end)
    {
        System.out.println("getAccountStatement(Date begin, Date end)");
        AccountStatement as = new AccountStatement(data.toArray(new String[data.size()]),
            new Request(begin, end));
        return new Future<AccountStatement>(as);
    }
    ...
}

```

- Create a mapping if the default mapping does not fit your needs:

```

public interface IRBankingService
{
    @GET
    @Path("getAS/")
    @Produces(MediaType.TEXT_HTML)
    @MethodMapper(value="getAccountStatement", parameters={Request.class})
    @ParametersMapper(@Value(clazz=RequestMapper.class))
    @ResultMapper(@Value(clazz=BeanToHTMLMapper.class))
    public String getAcci(String begin, String end);
    ...
}

```

This mapping defines a REST resource reachable under the base url plus *getAS*. This custom method uses its own request and result mappers and produces HTML output (generated by the BeanToHTMLMapper).

- Add the @Publish annotation with values for the publishing url (publishid), the publishtype (IPublishService.PUBLISH_RS or just "rs") and optional mapping information (the interface created in the last step). In the example below the service is published at localhost with port 8080:

```

@Agent
@Imports({"jadex.extension.ws.publish.*", "jadex.webservice.examples.ws.offerquote.gen.*"})
@ProvidedServices(@ProvidedService(type=IBankingService.class,
    implementation=@Implementation(BankingService.class),
    publish=@Publish(publishtype=IPublishService.PUBLISH_WS, publishid="http://localhost:8080/"))
public class BankingAgent
{
}

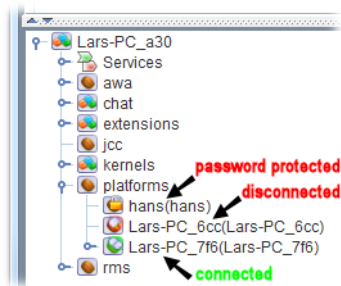
```

- In order to test if publishing has work you can start the component and check if the service info page is available. Per default the info page is reach-

able under the base url. Please note that due to some Jersey issues the base url has to contain a trailing slash. The example from above can be reached using `http://localhost:8080/banking/` (`http://localhost:8080/banking/`).

Chapter 7 - Platform Awareness

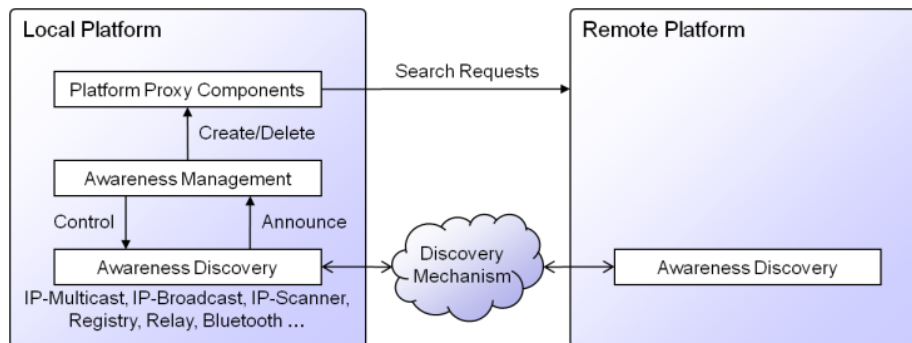
Platform awareness allows for automatic discovery of other platforms in the network. Depending on the enabled discovery mechanisms a platform will be capable of detecting others only within the same LAN or also globally. In Jadex, awareness is introduced by so called proxy components, which are local component representatives of a remote platform. Having a proxy component of another platform will integrate this remote platform transparently, i.e. given that security settings do not restrict access to the remote platform, its services can be discovered and used by local components. The concept of proxy components makes handling interactions with remote components rather simple. On the one hand these proxies can be created automatically by the awareness component and on the other hand a proxy can also be created manually given that the platform name and transport addresses are known.



Proxy components in Starter

In the screenshot of the JCC Starter shown above, it can be seen that platform proxy components are shown as subcomponents of the *platforms* component. In this case three other platforms have been detected called 'hans', "Lars-PC_6cc", and 'Lars-PC_76f'. The user interface also visualizes the state of the remote components. The first component is connected but password protected (lock icon) so that it cannot be directly accessed or inspected. The second platform was found but communication is disturbed, e.g. the other platform could have been terminated or a network or communication problem exists. The screenshot shows that a component named 'awa' exists, which is the default name for the awareness component.

Awareness Architecture



Awareness architecture

The awareness component is functionally split into a management and a discovery part. The management part is responsible for creating and deleting proxy components on the local platform whenever changes occur. These changes are detected by possibly different discovery components employing different kinds of discovery mechanisms. The discovery mechanisms have the task to distribute awareness infos of the own platform and also receive such infos from other platforms. Whenever a discovery component receives a new awareness info it will forward the info to the management component. The management component integrates the awareness infos of different sources and creates a new proxy if necessary. In addition, the management component uses lease times (contained in the awareness infos) to calculate the proxy validity. If no fresh awareness info is received after this timepoint has passed the management component will delete the proxy (with some delay tolerating latencies). Currently the following discovery mechanisms are available (base package `jadex.base.service.awareness.discovery` in module `jadex-platform-base`):

- **Broadcast discovery** (enabled per default, local network): Uses IP broadcast to announce awareness infos. It uses the default port '55670'. As IP broadcast is only available in IPV4 networks this mechanism will not work in pure IPV6 environments.
- **Multicast discovery** (enabled per default, local network): This mechanism uses IP multicasts to find other platforms. Per default it uses multicast address 224.0.0.0 and port 55667. As multicast requires receivers to register at the multicast address this mode does not send packets to other no Jadex nodes in the network.
- **IP-Scanner discovery** (disabled per default, local network): The scanner tries to find out the network type and send awareness infos to IP addresses within the network. The default port the scanner uses is 55668. Please note that ip scanning might conflict with the policies of your network administrator. First, the scanner floods the network with messages and second sending out messages to a bunch of network ips is sometimes considered to be an indication for virus behavior. For these

reasons the scanner is deactivated per default.

- **Registry discovery** (disabled per default, global network): The registry mechanism allows for using a dedicated registry platform at which all other platforms register at runtime (using the address argument). The registry distributes its entries to all platforms. If you want to use the registry you should provide it with a unique platform id (otherwise it will use some fallback that is not guaranteed to be online).
- **Message discovery** (enabled per default, global network): Message discovery is based on message receipt of other platforms. Whenever a message is received the message service will forward it to this discovery agent which subsequently announces a new platform. Using message discovery is especially beneficial in asymmetric network settings, in which one partner can find the other but not vice versa. This e.g. occurs with broadcast or multicast in virtual networks using NAT (e.g. VirtualBox or Android emulator).
- **Relay discovery** (enabled per default, global network): The relay discovery is based on a web server that is used as a common rendezvous point for the platforms, i.e. the web server distributes awareness infos among the currently connected nodes. Per default the relay uses a server at <http://jadex.informatik.uni-hamburg.de/relay/> (<http://jadex.informatik.uni-hamburg.de/relay/>) (<http://jadex.informatik.uni-hamburg.de/relay/>), but further servers are planned to be used. It is also possible to setup an own server using the relay WAR file from the downloads section, which allows for building up private platform networks.

Note: Experience has shown that the functioning of awareness mechanisms heavily depends of the concrete network infrastructure used. For this reason it is beneficial to enable multiple of them in order to get a robust setup.

Configuration

Awareness can be turned on/off with the argument:

-awareness true/false

This will start the platform with or without the awareness component. If you want to disable awareness at runtime it is sufficient to kill the 'awa' component. On the other hand it is also possible to enable awareness at runtime by starting the awareness component (*jadex.base.service.awareness.management.AwarenessManagementAgent*) contained in the module *jadex-platform-base*. In addition to the global awareness setting it is also possible to determine the awareness mechanisms that should be used. This can be done by using the argument:

`-awamechanisms new String[]{"Broadcast", "Multicast", "Message", "Relay"}`

At runtime the currently active awareness mechanisms can be seen by looking at the subcomponents of the awareness management component. To deactivate or activate mechanisms at runtime again subcomponents can be started or stopped. Furthermore, the delay between awareness announcements can be configured using the `-awadelay` argument, which per default is 20 seconds. This delay is propagated down to all awareness mechanisms at startup. At runtime the awareness settings can be further customized. For this purpose the awareness settings JCC plugin is available.

Chapter 8 - Security

Regarding security two different levels have to be distinguished: the **platform** and **application** level. The first is concerned with mechanisms for protecting a platform against unauthorized access and the latter deals with security aspects for services.

Platform Level Security



Figure 50: 08 Security@security.png

Platform security scheme

Platform level security is based on platform secrets, which are defined as platform passwords or networks. If no user defined password for a platform exists, the platform generates a random password at startup. This password can be changed later on using the security settings tool . The platform password as well as other secrets are saved in cleartext in the platform settings file located in the Jadex start folder. This implicates that this folder should also be secured and not be accessible for potential attackers.

Service communication between platforms is only possible if the service caller and the callee share a common secret. On the one hand, a shared secret can be established when the caller knows the platform password of the callee, on the other hand common secrets can be defined in the form of network names and (optional) passwords. If both platforms know at least one common platform name (password) combination communication is also possible. The network names may be completely virtual (such as 'mynetwork') or correspond to the IP network identifier (such as 134.11.100.0) for a class C IP4 network. Please note that collisions may occur with NAT based IP network names (such as

192.168.0.0) and for production environments, IP based network names should not be used without additional password, because these network names can be guessed easily by potential attackers.

The general communication scheme is depicted in the figure above. The service call of a **caller** from a platform to a **callee** on another platforms is processed in multiple steps. In a first step the call arrives (transparently for the caller) at a local **gateway** of the first platform. This gateway enhances the call with fingerprints of all currently known secrets, i.e. it computes salted hashcodes (currently SHA-384) of all the secrets and adds them to the call. On the receiving platform side the call reaches first again the gateway of this platform and will be checked. This is done by iterating over all known secrets and also computing the salted hashcode of them. It is then checked for each hashcode if it is contained in the set of fingerprints sent with the call. This scheme continues until the first matching fingerprint was found or no match could be identified. In the first case, the call is allowed and is forwarded to the intended callee component, which will subsequently process the call and return the result. In contrast, in the latter case it has been determined that no common secret could be found. This implicates that the service call is immediately rejected and a security exception is raised as result of the call.

Application Level Security

On application level different security settings can be used including access restrictions and security characteristics of services.

Service Access Restrictions

Per default all services are inaccessible from callers outside of the own platform that do not own a shared secret. In order to customize the access restrictions of a service as a whole or single methods of a service the *@Security* annotation can be used. The following settings can be made:

- ***@Security(Security.PASSWORD)***: Access to a service or service method is only possible if the caller knows a shared secret (as described above).
- ***@Security(Security.UNRESTRICTED)***: Access to a service or service method is possible in any case (no protection).

Note: Currently, per default a service interface is restricted. This means that service searches from callers with no shared secret will not proceed in that case even if single methods of the service are declared as unrestricted. To achieve an inclusion of a service in a search from an untrusted caller its interface has to be declared as unrestricted.

A small example code snippet is shown below for an artificial information service. This service offers a method to retrieve public information about a resource and another one for private info that is available for trusted platforms only. For this reason the service itself has been made unrestricted (in this way all components may find the service) but the *getPrivateInfo()* method has been made password protected.

```
@Security(Security.UNRESTRICTED)
public interface IInformationService
{
    public IFuture<String> getPublicInfo(String id);

    @Security(Security.PASSWORD)
    public IFuture<String> getPrivateInfo(String id);
}
```

Security Characteristics

Security characteristics of services refer to security objectives like confidentiality, integrity, and authentication. Currently, the following annotations are provided:

- **@SecureTransmission**: Ensures confidentiality and integrity of the data within the service call.
- **@Authenticated**: Ensures that only specific caller (platforms) can access a service method.

Secure Transmission

In order to ensure the confidentiality and integrity of a service call, the corresponding call is sent only via transport protocols that provide these characteristics. This means that the security objectives are annotated to the call and the available transport protocols are selected with respect to those requirements. Currently, the local transport (for calls within one platform) and the SSL transport offer a secure channel that provides these characteristics. (**Note:** The SSL transport is part of the Jadex pro version).

An example usage of the annotation is shown below:

```
public interface ISecurityService
{
    @SecureTransmission
    public IFuture<String> getLocalPassword();
}
```

In the example a small cutout of the *ISecurityService* of the Jadex platform is shown. In order to handle password changes from a user interface the interface has several methods that need to transfer the password via a service call. To make this call immune against eavesdropping the `@SecureTransmission` is annotated to the corresponding methods such as `getLocalPassword()`.

Authentication

Authentication can be used to ensure that only platforms with proven identity can access specific services and service methods. Authentication in Jadex relies on a **platform certificate** which is automatically generated as self-signed version if none is available. Jadex uses a local Java key store to store and retrieve its certificates, i.e. its own as well as certificates of other platforms. Certificate management can either be done automatically or manually by using the security settings plugin of the JCC. The `@Authenticated` annotation allows for specifying two parameters called *names* and *virtuals* both of type string array. As names an arbitrary number of trusted **platform prefix names** (i.e. the name without the autogenerated suffix “_xyz”) can be specified. The *virtuals* parameter **virtual platform names** can be given. These virtual names are mapped to real platform prefix names via a mapping table that needs to be supplied at platform startup. The virtual names have been introduced because otherwise the allowed platform names must be hardcoded within the Java code, which is not desirable e.g. if different installations of the same software need to be performed.

An example usage of the `@Authenticated` annotation is shown below (from :

```
public interface ITestService
{
    @Authenticated(names={"alphaplat", "betaplat"}, virtuals="testuser")
    public IFuture<String> getDBPassword(String user);
}
```

In this example the method `getDBPassword()` is equipped with an authenticated annotation that restricts access to platforms with the name “alphaplat” and “betaplat” and all platforms that can be mapped from the virtual name “testuser”. Please note, that it is also possible to just declare the `@Authenticated` annotation in the interface without giving any names. This keeps the interface totally clean of implementation details. The service implementation then has to provide the annotation again and provide the names that are permitted. In case a call is received that could not be authenticated a security exception is raised. In case virtual names are used the name mapping can be provided as map via the platform start parameter **virtualnames** as follows:

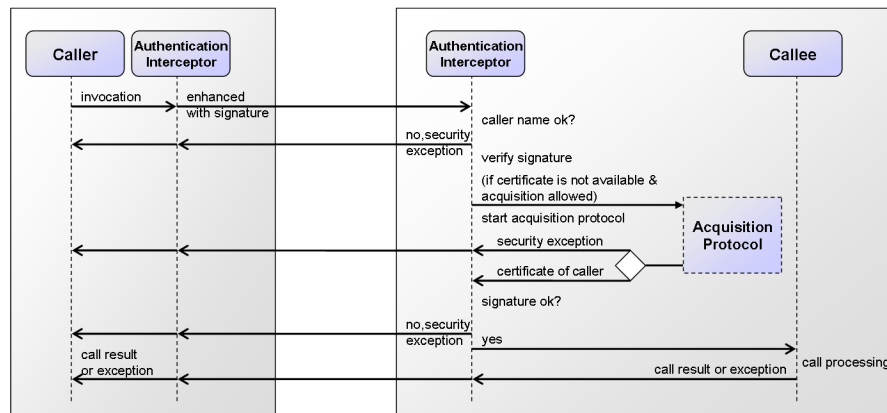
-virtualnames “jadex.common.SUtil.createHashMap(new String[]{\”testuser\”},

```
new Object[] {jadex.commons.SUtil.createHashSet(new String[] {\"willi\",
\"hans\"})}
```

Here the methods `createHashMap()` and `createHashSet()` of Jadex commons are used, but of course any other method call could also be employed that creates a map of the form `Map<String, Set<String>>`. In the example, the virtual name “testuser” is mapped to the two real platform names “willi” and “hans”. Please note, that the text above encloses the argument in quotation marks and escape the contained quotation marks to ensure that it is parsed as one element.

Authentication Protocol

The protocol that realizes the authentication is described next. It relies on the communication scheme shown below:



Authentication mechanism

The protocol shows how an authenticated call is performed between a caller and a callee. It has to be noted that the invocation mechanism is completely transparent to the caller and callee, except for the fact that the caller may receive a security exception in case it could not be authenticated or is not allowed to call the service. The invocation is first caught by the caller’s authentication interceptor to compute and add the signature to the call. It is then forwarded to the caller and processed by the authentication interceptor of the callee side. The interceptor will first check if the caller’s platform prefix name fits to one of the names stated within the annotation of `can be mapped` via the virtual names. If this is not the case the call will be rejected immediately and return with a security exception. Otherwise, the interceptor begin with the verification of the call. For this purpose it needs the public certificate of the caller’s platform. If it is not available via the local key store, depending on the security settings, a certificate acquisition protocol will be initiated. This protocol has the aim to fetch the caller’s certificate in a secured way and add it to the store. In acquisition has been disabled or the acquisition failed the authentication process is terminated with a security exception. Otherwise the certificate is used to check

the signature and depending on the result the call is either rejected again with an exception or forwarded to the callee for service processing. The call result (a normal result or an application level exception) is subsequently returned to the caller.

Certificate Acquisition Protocols

In case of installations consisting of multiple Jadex platforms the question arises how trusted platform certificates can be distributed among the platforms. The most secure way is to export the platform certificates to files (e.g. via the security settings in the JCC) and install them manually on each other platform. To reduce the amount of manual work also other automatic distribution schemes can be used. Currently, Jadex comes with the following two different acquisition mechanisms:

- **Decentralized acquisition:** The decentralized protocol asks all currently available platforms for a certificate of a platform. It collects the results and checks whether they are equal. In the protocol the number of received answers before it will install a certificate, can be adjusted. If the number is set to e.g. three, the protocol will wait for three certificate answers and compare those. If it is set to one, the mechanism will install the certificate without comparing it to any other answer. Such a scheme is insecure if performed in open networks, but if used in an intra net for an initial certificate distribution, this setting can be very helpful. In production, certificate acquisition could then be set to a higher number or it could be disabled completely.
- **Trusted third party (TTP) acquisition:** This mechanism relies on a dedicated trusted third party platform, which knows the certificates of all platforms in the own network environment. In case a client platform requires a certificate it will just ask the TTP for it. To make this scheme completely secure the communication with the TTP should be authenticated itself. This means that each client platform has to know two certificates (its own and the TTP certificate). The TTP certificate can be either manually installed or it can be directly fetched via the security settings gui in the JCC, given that client and TTP platform are online.

Jadex BDI is an agent-oriented reasoning engine for writing rational agents with XML and the Java programming language. Thereby, Jadex represents a conservative approach towards agent-orientation for several reasons. One main aspect is that no new programming language is introduced. Instead, Jadex agents can be programmed in the state-of-the art object-oriented integrated development environments (IDEs) such as [eclipse><http://www.eclipse.org/>](<http://www.eclipse.org/>)]. The other important aspect concerns the middleware independence of Jadex. As Jadex BDI is loosely coupled with its underlying middleware, Jadex can be used in very different scenarios on top of agent platforms as well as enterprise systems such as J2EE.

Similar to the paradigm shift towards object-orientation agents represent a new conceptual level of abstraction extending well-known and accepted object-oriented practices. Agent-oriented programs add the explicit concept of autonomous actors to the world of passive objects. In this respect agents represent active components with individual reasoning capabilities. This means that agents can exhibit reactive behavior (responding to external events) as well as pro-active behavior (motivated by the agents own goals).

This tutorial will cover the following topics, which correspond coarsely to the available Jadex language elements:

- [Chapter 2, BDI Concepts>02 Concepts] describes the basic BDI concepts.
- [Chapter 3, Agent Specification>03 Agent Specification] describes how an agent can be programmed.
- [Chapter 4, Imports>04 Imports] describes how elements can be imported.
- [Chapter 5, Capabilities>05 Capabilities] describes how agents modules (capabilities) can be used.
- [Chapter 6, Beliefs>06 Beliefs] describes how agents knowledge can be specified.
- [Chapter 7, Goals>07 Goals] treats how goals can be used.
- [Chapter 8, Plans>08 Plans] describes how procedural plans are used.
- [Chapter 9, Events>09 Events] handles internal as well as message events.
- [Chapter 10, Expressions>10 Expressions] treats the usage of Jadex expressions and their underlying language.
- [Chapter 11, Conditions>11 Conditions] describes Jadex conditions and their underlying language.
- [Chapter 12, Properties>12 Properties] describes how agent and capability properties can be defined.
- [Chapter 13, Configurations>13 Configurations] introduces configurations for starting agents with different settings.
- [Chapter 14, External Interactions>14 External Interactions] treats the interaction of external processes with BDI agents.
- [Chapter 15, Predefined Capabilities>15 Predefined Capabilities] explains the library of ready-to use capabilities.

<!

- 01 Introduction
- 02 Concepts
- 03 Agent Specification (old)
- 04 Imports (old)
- 05 Capabilities (old)
- 06 Beliefs (old)
- 07 Goals (old)
- 08 Plans (old)
- 09 Events (old)
- 10 Expressions (old)
- 11 Conditions (old)

- 12 Properties (old)
 - 13 Configurations (old)
 - 15 External Interactions (old)
 - 16 Predefined Capabilities (old)
 - A Changes (old)
 - B Platform Adapters (old)
 - C Add-Ons (old)
 - D FAQ & HOWTO (old)
 - E Legal Notice (old)
 - F Bibliography (old)
- >

This chapter shortly sketches the scientific background of Jadex and describes the concepts, and the execution model of Jadex agents.

1.1 The BDI Model of Jadex

Rational agents have an explicit representation of their environment (sometimes called world model) and of the objectives they are trying to achieve. Rationality means that the agent will always perform the most promising actions (based on the knowledge about itself and the world) to achieve its objectives. As it usually does not know all of the effects of an action in advance, it has to deliberate about the available options. For example a game playing agent may choose between a safe action or an action, which is risky, but has a higher reward in case of success.

To realise rational agents, numerous deliberative agent architectures exist (e.g. BDI [Bratman 87]), AOP [Shoham 93] and SOAR [Lehman et al. 96] to mention only the most prominent ones. In these architectures, the internal structure of an agent and therefore its capability of choosing a course of action is based on mental attitudes. The advantage of using mental attitudes in the design and realisation of agents and multi-agent systems is the natural (human-like) modelling and the high abstraction level, which simplifies the understanding of systems [McCarthy 79].

Regarding the theoretical foundation and the number of implemented and successfully applied systems, the most interesting and widespread agent architecture is the Belief-Desire-Intention BDI architecture, introduced by Bratman as a philosophical model for describing rational agents [Bratman 87]. It consists of the concepts of ~belief~, ~desire~, and ~intention~ as mental attitudes, that generate human action. Beliefs capture ~informational~ attitudes, desires ~motivational~ attitudes, and intentions ~deliberative~ attitudes of agents. [Rao and Georgeff 95] have adopted this model and transformed it into a formal theory and an execution model for software agents, based on the notion of beliefs, goals, and plans.

Jadex facilitates using the BDI model in the context mainstream programming,

by introducing beliefs, goals and plans as first class objects, that can be created and manipulated inside the agent. In Jadex, agents have beliefs, which can be any kind of Java object and are stored in a beliefbase. Goals represent the concrete motivations (e.g. states to be achieved) that influence an agent's behavior. To achieve its goals the agent executes plans, which are procedural recipes coded in Java. The abstract architecture of a Jadex agent is depicted in the following Figure 1.

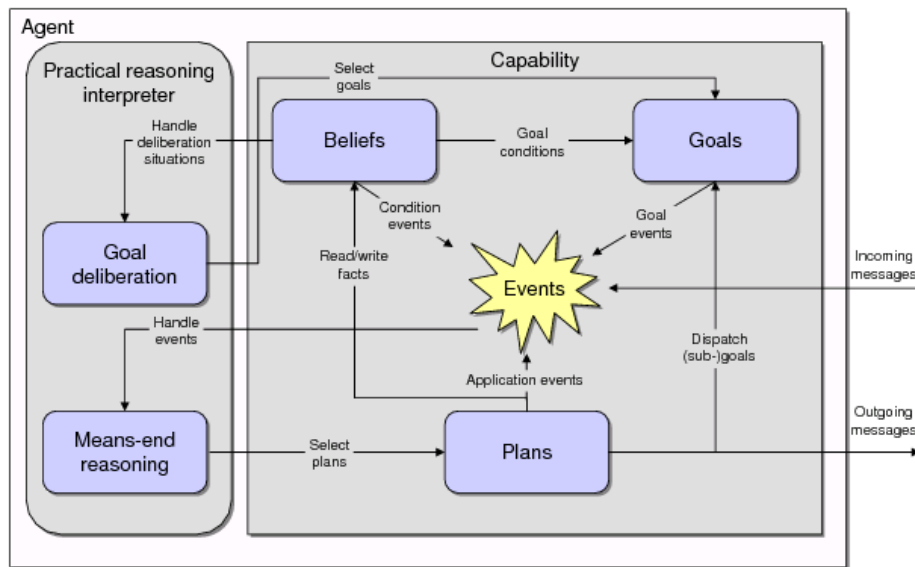


Figure 51:

~Figure 1: Jadex abstract architecture~

Reasoning in Jadex is a process consisting of two interleaved components. On the one hand, the agent reacts to incoming messages, internal events and goals by selecting and executing plans (means-end reasoning). On the other hand, the agent continuously deliberates about its current goals, to decide about a consistent subset, which should be pursued.

The main concepts of Jadex are beliefs, goals and plans. The beliefs, goals and plans of the agent are defined by the programmer and prescribe the behavior of the agent. E.g., the current beliefs influence the deliberation and means-end reasoning processes of the agent, and the plans may change the current beliefs while they are executed. Changed beliefs in turn may cause internal events, which may lead to the adoption of new goals and the execution of further plans. In the following the realisation of each of these main concepts in Jadex will be shortly described.

1.1 The Beliefbase

The beliefbase stores believed facts and is an access point for the data contained in the agent. Therefore, it provides more abstraction compared to e.g. attributes in the object-oriented world, and represents a unified view of the knowledge of an agent. In Jadex, the belief representation is very simple, and currently does not support any (e.g., logic-based) inference mechanism. The beliefbase contains strings that represent an identifier for a specific belief (similar to table names in relational databases). These identifiers are mapped to the beliefs values, called facts, which in turn can be arbitrary Java objects. Currently two classes of beliefs are supported: simple single-fact beliefs, and belief sets. Beliefs and belief sets are strongly typed, and the beliefbase checks at runtime, that only properly typed objects are stored.

On top of this simple belief representation, Jadex adds several advanced features, such as an OQL-like query language (adopted from the object-relational database world), conditions that trigger plans or goals when some beliefs change (resembling a rulebased programming style), and beliefs that are stored as expressions and evaluated dynamically on demand.

1.1 The Goal Structure

Unlike traditional BDI systems, which treat goals merely as a special kind of event, goals are a central concept in Jadex. Jadex follows the general idea that goals are concrete, momentary desires of an agent. For any goal it has, an agent will more or less directly engage into suitable actions, until it considers the goal as being reached, unreachable, or not desired any more. Unlike most other systems, Jadex does not assume that all adopted goals need to be consistent to each other. To distinguish between just adopted (i.e. desired) goals and actively pursued goals, a goal lifecycle is introduced which consists of the goal states `~option~`, `~active~`, and `~suspended~` (see Figure 2).

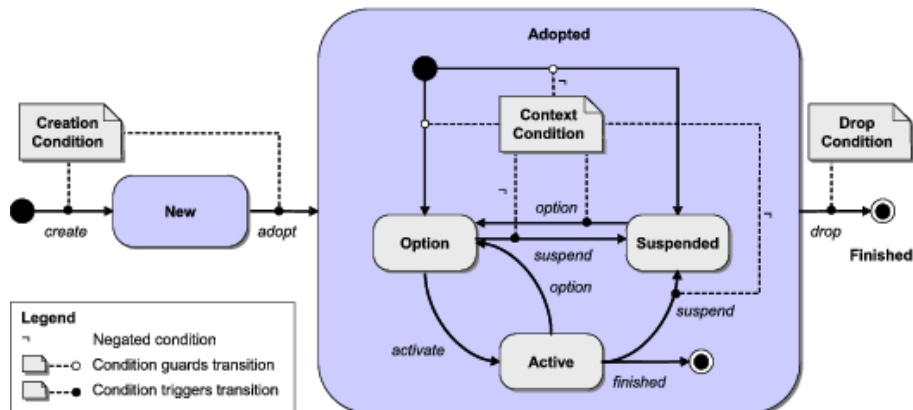


Figure 52:

~Figure 2: Goal life cycle~

When a goal is adopted, it becomes an option that is added to the agent's desire structure. Application specific goal deliberation mechanisms are responsible for managing the state transitions of all adopted goals (i.e. deciding which goals are active and which are just options). In addition, some goals may only be valid in specific contexts determined by the agent's beliefs. When the context of a goal is invalid it will be suspended until the context is valid again.

Four types of goals are supported by the Jadex system: Perform, achieve, query, and maintain goals as introduced by JAM [Huber 99]. A ~perform goal~ states that something should be done but may not necessarily lead to any specific result. For example, a waste-pickup robot may have a generic goal to wander around and look for waste, which is done by a specific plan for this functionality. The ~achieve goal~ describes an abstract target state to be reached, without specifying how to achieve it. Therefore, an agent can try out different alternatives to reach the goal. Consider a player agent that needs certain resources in a strategy game: It could choose to negotiate with other players or try to find the required resources itself. The ~query goal~ represents a need for information. If the information is not readily available, plans are selected and executed to gather the needed information. For example a cleaner robot that has picked up some waste needs to know where the next waste bin is located. If it already knows the location it can directly head towards the waste bin, otherwise it has to find one, e.g by executing a search plan. The ~maintain goal~ specifies a state that should be kept (maintained) once it is achieved. It is the most abstract goal in Jadex. Not only does it abstract from the concrete actions required to achieve the goal, but also it decouples the creation and adoption of the goal from the timepoint when it is executed. For example the goal to keep a reactor temperature below a certain level is a maintain goal that gets triggered whenever the temperature exceeds the normal operating level. As with achieve and query goals, to (re)establish the desired target state of a maintain goal, the agent may try out several plans, until the state is reached.

In Jadex BDI, goals are represented as objects with several attributes. The target state of achieve goals can be explicitly specified by an expression (e.g., referring to beliefs), which is evaluated to check if the goal is achieved. Attributes of the goal, such as the name, facilitate plan selection, e.g. by specifying that a plan can handle all goals of a given name. Additional (user-defined) goal parameters guide the actions of executing plans. For example in a goal to search for services (e.g. using the FIPA directory facilitator service), additional search constraints could be specified (such as the maximum cardinality of the result set). The structure of currently adopted goals is stored in the goalbase of an agent. The agent has a number of top-level goals, which serve as entry points in the goalbase. Goals in turn may have subgoals, forming a hierarchy or tree of goals.

1.1 Plan Specification

The concrete actions an agent may carry out to reach its goals are described in plans. An agent developer has to define the head and the body of a plan. The head contains the conditions under which the plan may be executed and is

specified in the agent definition file. The body of the plan is a procedural recipe describing the actions to take in order to achieve a goal or react to some event. The current version of Jadex supports plan bodies written in Java, providing all the flexibilities of the Java programming language (object-oriented programming, access to third party packages, etc.). At runtime, plans are instantiated to handle events and to achieve goals. Activation triggers in the plan headers are used to specify if a plan should be instantiated when a certain event or goal occurs. In addition, so called initial plans get executed when the agent is born. During the execution of the plan body, running plans may not only execute arbitrary Java code but can also dispatch subgoals and wait for events to occur.

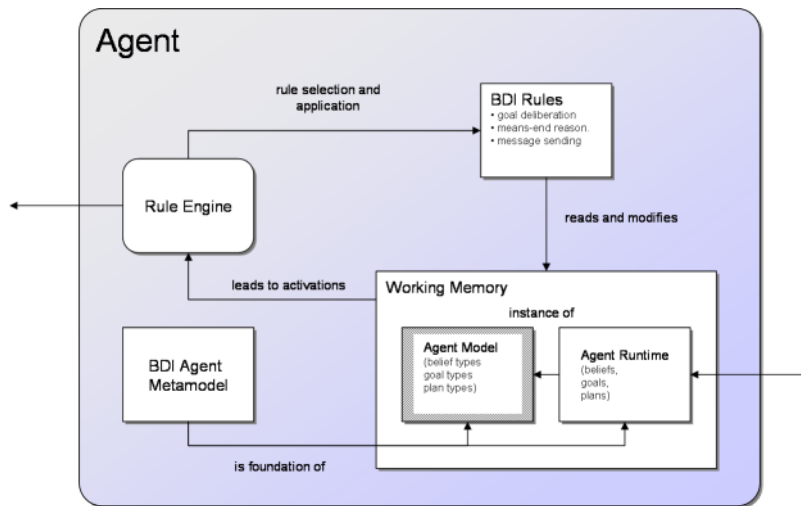
1.1 Agent Definition

The complete definition of an agent is captured in a so called ~agent definition file~ (ADF). The ADF is an XML file, which contains all relevant properties of an agent (e.g. the beliefs, goals and plans). In addition to the XML tags for the agent elements, the developer can use expressions in a Java-like syntax for specifying belief values and goal parameters. The ADF is a kind of a class description for agents: From the ADF agents get instantiated like Objects get instantiated from their class. For example, the different player agents from BlackJack (`src/jadex/bdi/examples/blackjack`) share `Player.agent.xml` as their definition file.

For each element ADF in the all important properties can be defined as attributes or subtags. For example, plans are declared by specifying how to instantiate them from their Java class (body tag), and a trigger (e.g. event) can be stated, that determines under which conditions a plan gets executed. Moreover, in the ADF, the initial state of an agent (how the agent should look like, when it is born) is determined in a so called configuration, which defines the initial beliefs, initial goals, and initial plans.

1.1 Execution Model of a Jadex Agent

This sections shows the operation of the reasoning component, given the Jadex BDI concepts. Since version 0.93 Jadex does not employ the classical BDI-interpreter cycle as described in the literature \[Rao and Georgeff 95\] but uses a new execution scheme (described more extensive in \[Pokahr and Braubach 09\]). In Jadex V2 the interpreter is a rule based system operating on a set of predefined BDI rules. The basic mode of operation is simple: The agent selects an activation (a triggering rule) from the agenda and executes the corresponding action. The BDI interpreter is depicted in the following figure.



~Figure 3: Jadex BDI interpreter architecture~

The working memory of a BDI agent consists of two parts: the agent model and the agent runtime state. The agent model represents the type definition of a BDI agent and determine which beliefs, plans and goals (besides other elements) an agent possesses. In the runtime state concrete instances of these model elements are contained, e.g. belief values, specific goals and plans. The conceptual foundation of how an agent model and instance should look like is defined in the BDI agent metamodel.

Advantages of the new rule based approach are that the new mechanism offers a much higher degree of extensibility and flexibility as new functionalities such as emotions or team behaviour can be added as new set of rules without having to change the interpreter itself. One concrete effect already contained in this version is the support for goal deliberation via the “Easy Deliberation” strategy, which allows users to define which goals inhibit others to ensure that the active set of currently pursued goals by the agent is always consistent.

The BDI programmer’s guide is a reference to the concepts and constructs available, when programming Jadex agents. It is not meant as a step-by-step introduction to the programming of Jadex agents. For a step-by-step introduction consider working through the [BDI Tutorial>BDI Tutorial.01 Introduction]

1.1 Overview

To develop applications with Jadex, the programmer has to create two types of files: XML agent definition files (ADF) and Java classes for the plan implementations. The ADF can be seen as a type specification for a class of instantiated agents. For example Buyer agents (from the booktrading example) are defined by the Buyer.agent.xml file, and use plans implemented, e.g. in the

file PurchaseBookPlan.java. The user guide describes both aspects of agent programming, the XML based ADF declaration and the plan programming Java API, and highlights the interrelations between them. Detailed reference documentation for the XML definition as well as the plan programming API is also separately available in form of the generated XML schema documentation and the generated Javadocs. Figure 1 depicts how XML and Java files together define the functionality of an agent. To start an agent, first the ADF is loaded, and the agent is initialized with beliefs, goals, and plans as specified.

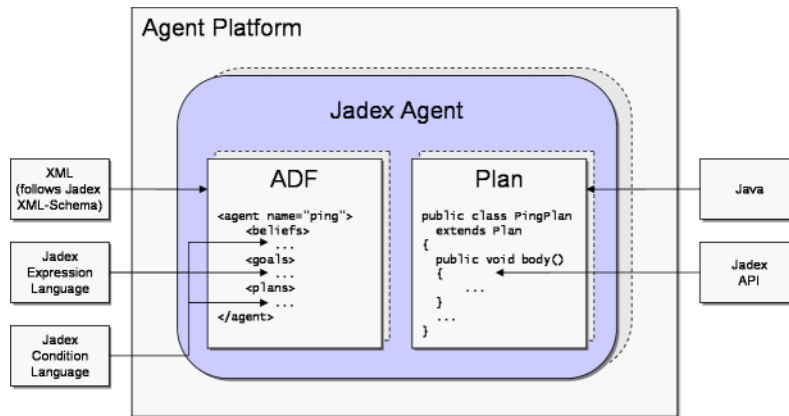


Figure 53:

~Figure 1: Jadex BDI agent components~

Please note that different languages are used for the agent specification in the ADF. For all ~values~ that should be created the ~Jadex expression language~ is used. This language is very similar to Java and extends it with a small set of OQL statements. These basically allow you to use ~select~ statements for fetching specific data in a declarative way. On the other hand there is the ~Jadex condition language~, which is used for most of the conditions (e.g. a target condition of an achieve goal). This language is also very Java like and introduces also some small extensions. Note that it is (in most cases) not used for retrieving a value, but for signalling a specific situation. There are some elements called conditions, which are not real conditions but checked only once. These elements require the usage of the Jadex expression language (precondition of a plan). In most cases the differences between both should not be really apparent due to their Java similarity.

1.1 Structure of Agent Definition Files (ADFs)

```
{code:xml}
<agent xmlns="http://jadex.sourceforge.net/jadex-bdi"(http://jadex.sourceforge.net/jadex-bdi)"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance](http://www.w3.org/2001/XMLSchema-
```

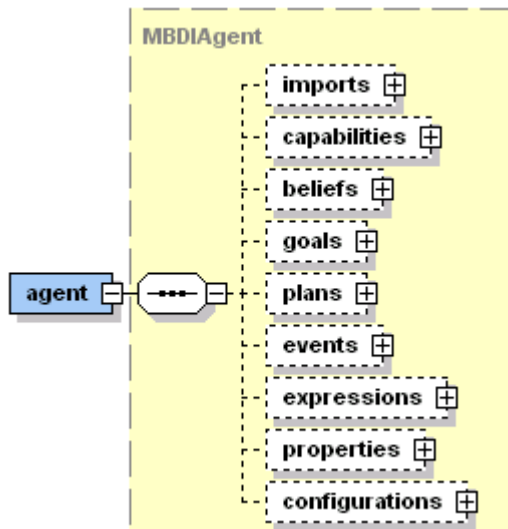
```

instance)”
    xsi:schemaLocation=“http://jadex.sourceforge.net/jadex-bdi |(http://jadex.sourceforge.net/jadex-
bdi)
        http://jadex.sourceforge.net/jadex-bdi-2.0.xsd |(http://jadex.sourceforge.net/jadex-
bdi-2.0.xsd)”
    name=“Buyer” package=“jadex.bdi.examples.booktrading.buyer”>
    ...
</agent>
{code}

```

~Figure 2: Header of an agent definition file~

The head of an ADF looks like shown in Figure 2. First, the agent tag specifies that the XML document follows the jadex-2.0.xsd schema definition which allows to verify that the document is not only well formed XML but also a valid ADF. The name of the agent type is specified in the name attribute of the agent tag, which should match the file name without suffix (<filename>.agent.xml</filename>). It is also used as default name for new agent instances, when the ADF is loaded in the starter panel of the Jadex Control Center. The package declaration specifies where the agent first searches for required classes (e.g., for plans or beliefs) and should correspond to the directory, the XML file is located in. Additionally required packages can be specified using the <imports> tag.



~Figure 3: Jadex top level ADF elements~

Figure 3 above shows which elements can be specified inside an agent definition file (please refer also to the commented schema documentation generated from the schema itself in [BDI Schema Documentation><http://jadex-agents.informatik.uni-hamburg.de/docs/jadex-2.0x/kernel-bdi/schema/jadex-bdi-2.0.html>](<http://jadex-agents.informatik.uni-hamburg.de/docs/jadex-2.0x/kernel-bdi/schema/jadex-bdi-2.0.html>)).

2.0x/kernel-bdi/schema/jadex-bdi-2.0.html)]. The `<imports>` tag is used to specify, which classes and packages can be used by expressions throughout the ADF. To modularize agent functionality, agents can be decomposed into so called capabilities. The capability specifications used by an agent are referenced in the `<capabilities>` tag. The core part of the agent specification regards the definition of the beliefs, goals, and plans of the agent, which are placed in the `<beliefs>`, `<goals>`, and `<plans>` tag, respectively. The events known by the agent are defined in the `<events>` section. The `<expressions>` tag allows to specify expressions and conditions, which can be used as predefined queries from plans. The `<properties>` tag is used for custom settings such as debugging and logging options. Finally, in the `<configurations>` section, predefined configurations containing, e.g., initial beliefs, goals, and plans, as well as end goals and plans are specified.

It should be noted that, unless otherwise stated, the order of occurrence of the elements is prescribed by the underlying XML Schema. Therefore, you should not, e.g., declare plans before beliefs. Throughout this user guide figure like above will always denote the correct order of element appearance (from top to bottom). Of course, it is possible to omit those elements, which are not required for your agent.

When an ADF is loaded, a model is created for the agent containing (e.g., beliefs, goals, plans) defined in the ADF. These model elements are kept hidden from the agent programmer who has access to the runtime elements only. In Jadex V2 the model is assumed to be unchangeable so that accessing model data should not be necessary (in case information is needed from the model the corresponding runtime elements should provide accessor methods to the needed info). When the agent is executed, instances of the model elements are created; so called runtime elements (package `jadex.bdi.runtime`, e.g., `IBelief`, `IGoal`, `IPlan`). This ensures that for modelled elements at runtime several instances (`IGoal` objects) can be created. For example, the buyer agent will instantiate new purchase book goal (`IGoal`) for each book to be bought, based on the goal specification in the ADF. Think of the relation between model elements and runtime elements as corresponding to the relation between `java.lang.Class` and `java.lang.Object`.

The `<imports>` tag is used to specify, which classes and packages can be used by Java expressions throughout an agent or capability definition file. The import section with an ADF resembles very much the Java import section of a class file. A Jadex import statement has the same syntax as in Java allowing single classes as well as whole packages being included.

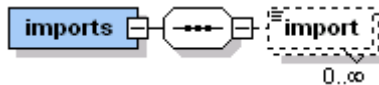


Figure 54:

~Figure 1: The Jadex imports XML schema part~

The imports are used for searching Java classes as well as non-Java agent artifacts such as agent.xml or capability.xml files. It is not necessary to declare an import statement for the actual package of the ADF as this is automatically considered.

1.1 Import Examples

In the following some simple code snippets from an ADF are shown that demonstrate how import statements are declared and subsequently used, e.g., in facts of beliefs, or to include a capability from another package.

```
{code:xml}
<imports>
  <!-- Import only the HashMap class.-->
  <import>java.util.HashMap</import>

  <!-- Import all classes of the awt package.-->
  <import>java.awt.*</import>

  <!-- Import a movement package containing, e.g., a Move capability.-->
  <import>movement.*</import>
  ...
</imports>

<capabilities>
  <!-- Use the imported movement.Move capability.-->
  <capability name="movecap" file="Move"/>
</capabilities>

<beliefs>
  <!-- Use the imported java.util.HashMap.-->
  <belief name="data">
    <fact>new HashMap()</fact>
  </belief>

  <!-- Use the imported java.awt.Frame.-->
  <belief name="gui">
    <fact>new Frame()</fact>
  </belief>
</beliefs>
{code}
```

~Figure 2: Example import declaration and usage~

The term capability is used for different purposes in the agent community. In the context of Jadex, the term is used to denote an encapsulated agent module composed of beliefs, goals, and plans. The concept of an agent module (and the usage of the term capability) was proposed by [Busetta et al. 99] and first implemented in JACK Agents [Winikoff 05]. Capabilities allow for packaging a

subset of beliefs, plans, and goals into an agent module and to reuse this module wherever needed. Capabilities can contain subcapabilities forming arbitrary hierarchies of modules. In Jadex, a revised and extended capability model has been implemented as described in [Braubach et al. 05]. In this model, the connection between a parent (outer) and a child (inner) capability is established by a uniform visibility mechanism for contained elements (see Figure 1).

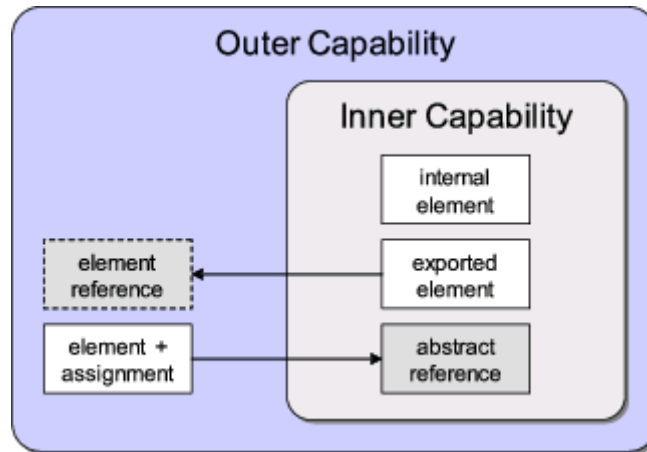


Figure 55:

Figure 1: Capability concept

Capability Definition

A capability is basically the same as an agent, but without its own reasoning process. On the other hand, an agent can be seen as a collection (i.e. subcapability hierarchy) of capabilities plus a separate reasoning process shared by all its capabilities. Each agent has at least one capability (sometimes called root capability) which is given by the beliefs, goals, plans, etc. contained in the agent's XML file. To create additional capabilities for reuse in different agents, the developer has to write capability definition files. A capability definition file is similar to an agent definition file, but with the `<agent>` tag replaced by `<capability>`. The `<capability>` tag has the same substructure as the `<agent>` tag.

Note that the `<capability>` tag has *name* and *package* attributes. As there are so many similarities between agent definition files and capability definition files, we commonly use the term ADF to denote both.

```
<capability xmlns="http://jadex.sourceforge.net/jadex-bdi"
```



```

        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://jadex.sourceforge.net/jadex-bdi
                            http://jadex.sourceforge.net/jadex-bdi-2.0.xsd"
name="MyCapability" package="mypackage">

    <beliefs> ... </beliefs>
    <goals> ... </goals>
    <plans> ... </plans>
    ...
</capability>

```

Figure 2: Capability XML file header

Using Capabilities

Agents and capabilities may be composed of any number of subcapabilities which are referenced in a <capabilities> tag. To reference a capability, a local name and the location of the capability definition has to be supplied in the file attribute as absolute or relative file name or capability type name. Type names are resolved using the package and import declarations, and can therefore be unqualified or fully qualified. Capabilities from the jadex.bdi.planlib package, such as the DF capability, which have platform-specific implementations, must always be referenced using a fully qualified type name.

```

<agent ...>
  <capabilities>
    <!-- Referencing a capability using a filename. -->
    <capability name="mysubcap" file="mypackage/MyCapability.capability.xml"/>

    <!-- Referencing a capability using a fully qualified type name. -->
    <capability name="dfcap" file="jadex.planlib.DF"/>
    ...
  </capabilities>
  ...
</agent>

```

Elements of a Capability

The capability introduces a scoping of the BDI concepts. By default all beliefs,

goals, and plans have local scope (i.e., are not exported), that is they can only be used in the capability where they have been defined. This restriction can be relaxed by declaring elements as exported or abstract for making them accessible from the outer capability (cf. Figure 1). In the outer capability such elements can be used when an explicit reference (with its own possibly different name) to those elements is established. In Figure 2 this reference mechanism, which applies to all elements in the same manner, is exemplarily depicted for beliefs. In the following the possible use cases are described.

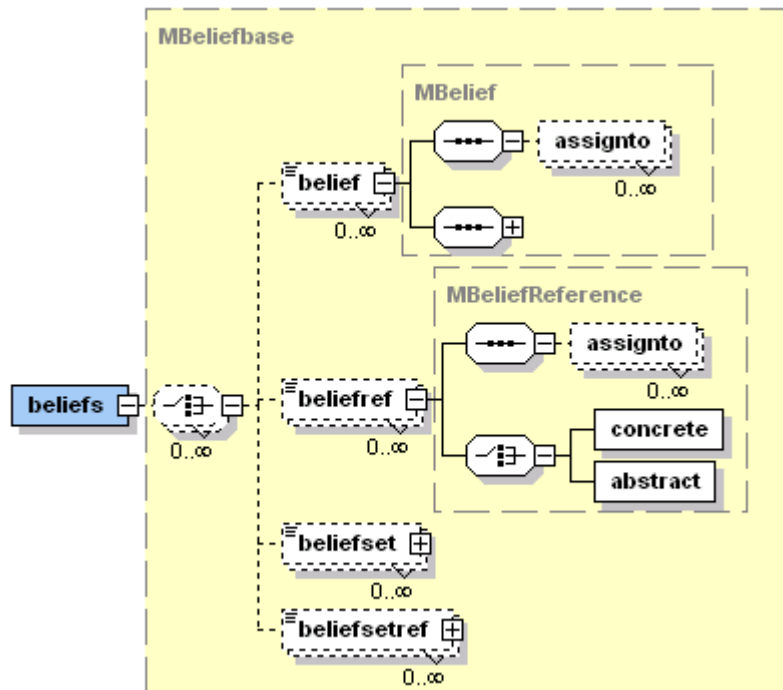


Figure 2: The Jadex references XML schema elements (using beliefs as example)

Making an Element Accessible for the Outer Capability

For this purpose the element must declare itself as exported (using the exported="true" attribute) in the inner capability. In the outer capability, a reference (e.g., <beliefref>) has to be declared, which directly references the original element (using dot notation "capname.belname") within the concrete tag. An example for an exported belief is shown below.

Inner Capability A

```
<belief name="myexportedbelief" exported="true" class="MyFact"/>
```

Outer Capability B includes A under the name mysubcap

```
<beliefref name="mysubbelief">  
  <concrete ref="mysubcap.myexportedbelief"/>  
</beliefref>
```

Defining an Abstract Element

This means the element itself provides no implementation and needs to be assigned from an outer capability. For this purpose an abstract element reference (e.g., <beliefref>) has to be declared. An outer capability can provide an implementation for this abstract element by defining a concrete element (or another reference) and assigning it to the abstract reference (using the <assignto> tag). In addition, the abstract element can be declared as optional (using the optional="true" attribute of the abstract tag) requiring no outer element assignment. At runtime, such unassigned abstract elements are not accessible, and trying to use them will result in runtime exceptions. For some of the elements (e.g., beliefs) it can be tested at runtime with the *isAccessible()* method from within plans, if a reference is connected.

Inner Capability A

```
<beliefref name="myabstractbelief" exported="true">  
  <abstract/>  
</beliefref>
```

Outer Capability B includes A under the name mysubcap

```
<belief name="mybelief" class="MyFact">
```

```

    <assignto ref="mysubcap.myabstractbelief"/>
  </belief>

```

Beliefs represent the agent's knowledge about its environment and itself. In Jadex the beliefs can be any Java objects. They are stored in a belief base and can be referenced in expressions, as well as accessed and modified from plans using the beliefbase interface.

Defining Beliefs in the ADF

The beliefbase is the container for the facts known by the agent. Beliefs are usually defined in the ADF and accessed and modified from plans. To define a single valued belief or a multi-valued belief set in the ADF the developer has to use the corresponding `<belief>` or `<beliefset>` tags and has to provide a name and a class. The name is used to refer to the fact(s) contained in the belief. The class specifies the (super) class of the fact objects that can be stored in the belief. The default fact(s) of a belief may be supplied in enclosed `<fact>` tags. Alternatively, for belief sets a collection of initial facts can be directly specified using a `<facts>` tag. This is useful, when you do not know the number of initial facts in advance, e.g., when invoking a static method or retrieving values from a database (see Figure 1). References to beliefs and belief sets from inner capabilities can be defined using the `<beliefref>` and `<beliefsetref>` tags (cf. Figure 2 in the Capabilities chapter).

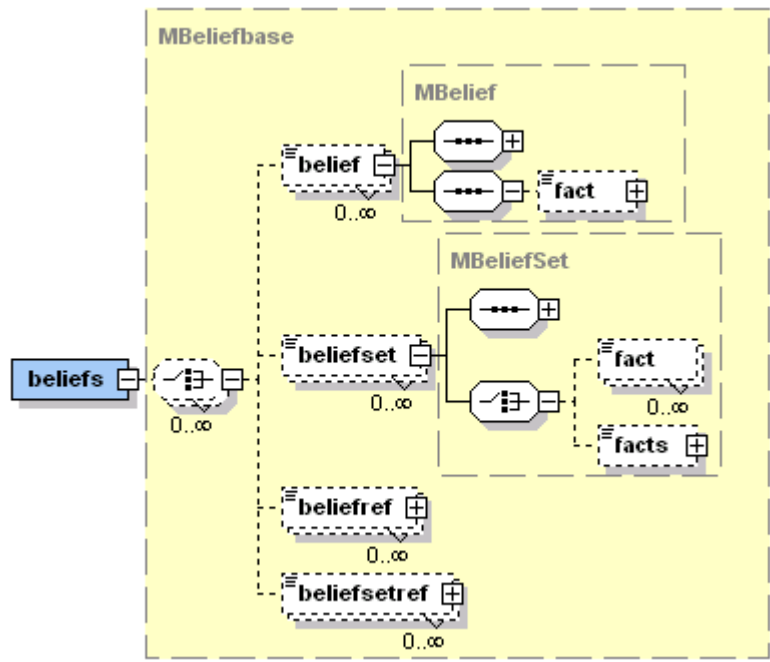


Figure 1: The Jadex beliefs XML schema part

```

...
<beliefs>
  <belief name="my_location" class="Location">
    <fact>new Location("Hamburg")</fact>
  </belief>
  <beliefset name="my_friends" class="String">
    <fact>"Alex"</fact>
    <fact>"Blandi"</fact>
    <fact>"Charlie"</fact>
  </beliefset>
  <beliefset name="my_opponents" class="String">
    <facts>Database.getOpponents()</facts>
  </beliefset>
  ...
</beliefs>
...
</agent>

```

Example belief definition

Accessing Beliefs from within Plans

From within a plan, the programmer has access to the beliefbase (interface `IBeliefbase`) using the `getBeliefbase()` method. The beliefbase provides `getBelief()` / `getBeliefSet()` methods to get the current beliefs and belief sets by name, as well as methods to create new beliefs and belief sets or remove old ones. The content of a belief (interface `IBelief`) can be accessed by the `getFact()` method. A belief set (interface `IBeliefSet`) is accessed through the `getFacts()` method and will return an appropriately typed array of facts. To check if a fact is contained in a belief set the `containsFact()` method can be used.

The contents of a single fact belief are modified using the `setFact()` method. Setting a fact on a belief will result in overwriting the previous value, if any. For deleting the fact of a single fact belief, you can set the belief value to null. Belief sets are manipulated using the `addFact(fact)` / `removeFact(fact)` methods. When removing facts that do not exist from the belief set, the belief set remains unchanged and a warning message will be produced. For the remove operation, the beliefbase relies on the implementation of the `equals()` method of the fact objects. Additionally, `<methodname>updateFact(fact)` can be used to replace an existing fact value.

```

public void body
{

```

```

...
IBelief hungry = getBeliefbase().getBelief("hungry");
hungry.setFact(new Boolean(true));
...
Food[] food = (Food[])getBeliefbase().getBeliefSet("food").getFacts();
...
}

```

A simple example of using a boolean belief

Dynamically Evaluated Beliefs

In the ADF the initial facts of beliefs are specified using expressions. Normally, the fact expressions are evaluated only once: When the agent is born. The evaluation behavior of the fact expression can be adjusted using the `evaluationmode` attribute. Possible values are 'static' (default), 'pull' and 'push'. Additionally, an `updaterate` may be specified as attribute of the belief that will cause the fact to be continuously evaluated and updated in the given time interval (in milliseconds).

In the example, the first belief "time" is evaluated on access (pull), and will therefore always contain the exact current time as returned by the Java function `System.currentTimeMillis()`. The second belief "timer" is not only evaluated on access (i.e., when accessed), but also every 10 seconds (10000 milliseconds). The advantage of using an `updaterate` for continuously evaluating a belief is that the fact value changes even when it is not accessed, and therefore may trigger conditions referring to that belief. For example, using the "timer" belief you could define a condition to invoke a plan that has to be executed in continuous intervals. Both options also provide an easy and effective way for making an agent aware of external input (e.g., sensory data available through a Java API).

```

<beliefs>
  <!-- A belief holding the current time (re-evaluated on every access). -->
  <belief name="time" class="long" evaluationmode="pull">
    <fact>System.currentTimeMillis()</fact>
  </belief>

  <!-- A belief continuously updated every 10 seconds. -->
  <belief name="timer" class="long" updaterate="10000">
    <fact>System.currentTimeMillis()</fact>
  </belief>
</beliefs>

```

Examples of dynamically evaluated beliefs

When setting the evaluation mode to 'push' the fact expression will be monitored for changes and a belief change event will be automatically generated as described in the next section.

Propagation of Belief Changes

To monitor conditions, an agent observes the beliefs and automatically reacts to changes of these beliefs, as necessary. Jadex is aware of manipulation operations that are executed directly on beliefs, e.g., by setting the fact of a belief, and of changes due to belief dependencies (i.e., a dynamically evaluated fact expression referencing another belief).

On the other hand, when you retrieve a complex fact object from a belief or belief set and perform operations on it subsequently, the system cannot detect the changes made. To enable the system detecting these changes the standard Java beans event notification mechanism can be used. This means that the bean has to implement the `add/removePropertyChangeListener()` methods and has to fire property change events, whenever an important change has occurred. The belief will automatically add and remove itself as a property change listener on its facts. An example how to implement this functionality inside a Java bean is shown below.

```
import java.beans.PropertyChangeSupport;
import java.beans.PropertyChangeListener;

public class Location
{
    private int x, y;
    private PropertyChangeSupport pcs;

    public Location(int x, int y)
    {
        this.x = x;
        this.y = y;
        this.pcs = new PropertyChangeSupport(this);
    }

    public int getX()
    {
        return this.x;
    }

    public void setX(int x)
```

```

{
    int old = this.x;
    this.x = x;
    this.pcs.firePropertyChange("X", old, this.x);
}

public int getY()
{
    return this.y;
}

public void setY(int y)
{
    int old = this.y;
    this.y = y;
    this.pcs.firePropertyChange("Y", old, this.y);
}

public void addPropertyChangeListener(PropertyChangeListener listener)
{
    pcs.addPropertyChangeListener(listener);
}

public void removePropertyChangeListener(PropertyChangeListener listener)
{
    pcs.removePropertyChangeListener(listener);
}
}

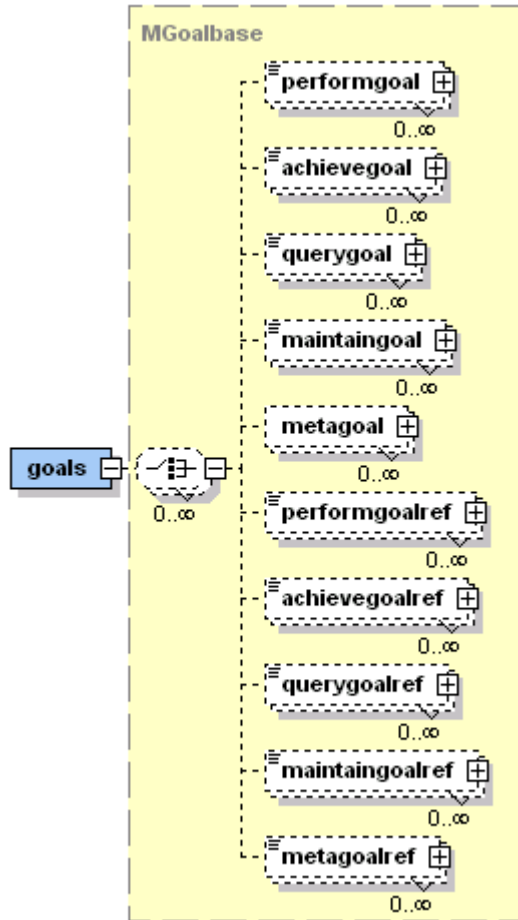
```

Example bean class with property change support

Goals make up the agent's motivational stance and are the driving forces for its actions. Therefore, the representation and handling of goals is one of the main features of Jadex. The concepts that make up the basis for the representation of goals in Jadex are described in the following sections and in more detail in [Braubach et al. 04] and [Braubach et al. 09]. Currently Jadex supports four different goal kinds and a meta-level goal kind. A perform goal specifies some activities to be done. Therefore the outcome of the goal depends only on the fact, if activities were performed. In contrast, an achieve goal can be seen as a goal in the classical sense by representing a target state that needs to be achieved. Similar to the behavior of the achieve goal is the query goal, which is used to enquire information about a specified issue. The maintain goal has the purpose to observe some desired world state and actively reestablishes this state when it gets violated. Meta-level goals can be used in the plan selection process for reasoning about events and suitable plans.

Figure 1 shows that a specific tag for each goal kind exists. Additionally, the

\<...goalref\> tags allow for the definition of references to goals from other capabilities (cf. Figure 1 in the Capabilities section).



~Figure 1: The Jadex goals XML schema part~

At runtime an agent may have any number of top-level goals, as well as subgoals (i.e. belonging to some plan). Top-level goals may be created when the agent is born (contained in an initial state in the ADF) or will be later adopted at runtime, while subgoals can only be dispatched by a running plan. Regardless of how a goal was created, the agent will automatically try to select appropriate plans to achieve all of its goals. The properties of a goal, specified in the ADF, influence when and how the agent handles these goals. In the following, the features common to all goal kinds will be described, thereafter the special features of the specific goal kinds will be explained.

1.1 Common Goal Features

In Figure 2 the base type of all four goal kinds is depicted to illustrate the shared

goal features. In Jadex, goals are strongly typed in the sense that all goal types can be identified per name and all parameters of a goal have to be declared in the XML. The declaration of parameters resembles very much the specification of beliefs. Therefore it is distinguished between single-valued parameters and multi-valued parameter sets. As with beliefs, arbitrary expressions can be supplied for the parameter values. The system distinguishes `~in~`, `~out~`, and `~inout~`, parameters, specified using the `~direction~` attribute. `~in~` parameters are set before the goal is dispatched, while `~out~` parameters are set by the plan processing the goal, and can be read when the goal returns. Additionally, it can be specified that a parameter is not mandatory by using the `~optional~` attribute.

Whenever a goal instance of the declared type is created and dispatched to the system it will be checked with respect to its parameters, and when no value has been supplied for a mandatory `~in~` parameter or parameter set a runtime exception will be thrown. The creation of new goals can be further influenced by using binding options for parameters via the a normal `<bindingoptions>` tag that expects a set of values. All possible combinations of assignments of binding parameters will be calculated when the creation condition is affected from a change. For those bindings that fulfill the creation condition new goals are instantiated. If you expect the creation condition to trigger for each element that should be processed it may be sufficient to use a normal `<value>` tag and refer in it to a variable that is bound in the creation condition.

~Figure 2: The Jadex common goal features~

The `<unique>` settings influence if a goal is adopted or ignored. When the unique tag is present, the agent does not adopt two equal instances of the goal at once. By default two goal instances of the same type are equal, when all parameters and parameter sets have the same values. Using the `<exclude>` tag this default behavior can be overridden by specifying which parameter(set)s should not be considered in the comparison. When a plan tries to adopt a goal that already exists and is declared as unique, a goal failure exception is thrown.

To describe the situations in which a new goal of the declared user type will be automatically instantiated, the `<creationcondition>` may be used. For adopted goals, it can be specified under which conditions such a goal has to be suspended or dropped using the `<contextcondition>` and `<dropcondition>` respectively. The suspension of a goal means that all currently executing plans for that goal and all subgoals are terminated at once. If the suspension is cancelled, new means for achieving the goal will be initiated. On the other hand, when a goal is dropped it is removed from the agent, and cannot be reactivated. Finally, the `<recurcondition>` can be used to specify in what cases the reactivation of a goal should be considered. In case the recur mode of a goal is enabled it is considered as a potential long-term goal and will not be dropped when no available plan can immediately achieve the goal. Instead of failing the goal is paused and will be reconsidered when the recur condition triggers. The `<deliberation>` settings, which influence which of the possible (i.e., not suspended) goals get pursued,

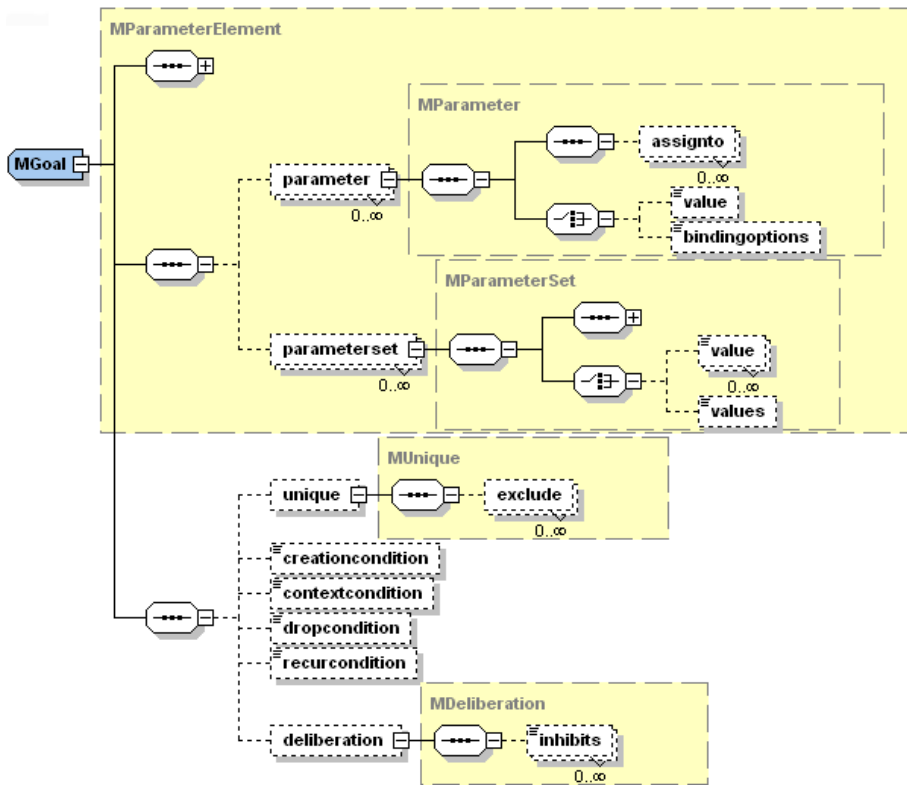


Figure 56:

will be explained in the deliberation section.

1.1 Example Goal

The following Figure shows an example goal using most of the features described above. It is a simplified example taken from the Hunter-Prey scenario (package `jadex.bdi.examples.hunterprey`) from the `BasicBehaviour` capability, common to all prey creatures. The goal is named “`eat_food`” and has one parameter `$food`, which is assigned from binding options taken from the food belief set. It is created whenever there is food (in the `$food` parameter) and the creature is allowed to eat (see creation condition). The goal is `<unique/>` meaning that the creature will not pursue two goals to eat the same food at the same time. Moreover, the `<deliberation/>` settings specify that the `eat_food` goal is more important than the `wander_around` goal.

```
{code:xml}
<achievegoal name="eat_food">
  <parameter name="food" class="ISpaceObject">
    <value>$food</value>
  </parameter>
  <unique/>
  <creationcondition language="jcl">
    $beliefbase.eating_allowed
  </creationcondition>
  <dropcondition language="jcl">
    !Arrays.asList($beliefbase.seen_food).contains($goal.food)
  </dropcondition>
  <deliberation>
    <inhibits ref="wander_around"/>
  </deliberation>
</achievegoal>
{code}
```

~Example goal (similar in Hunter-Prey scenario)~

1.1 BDI Flags

The handling and the exposed behavior of goals can be adapted to the requirements of your application using the so called BDI flags as depicted in the table below. The flags can be specified as attributes of the different goal tags in the ADF. The `retry` flag indicates that the goal should be retried or redone, until it is reached, or no more plans are available, which can handle the goal. An optional waiting time (in milliseconds) can be specified using the `retrydelay`. The `exclude` flag is used in conjunction with `retry` and indicates that, when retrying a goal, only plans should be called that where not already executed for that goal.

```
{table}
Name| Default| Possible Values
retry| true| true false
```

```

retrydelay| 0| positive long value
exclude| “when_tried”| “when_tried”, “when_succeeded”, “when_failed”,
“never”
posttoall| false| true false
randomselection| false |true false
metalevelreasoning| true |true false
recur| false| true false
recurdelay| 0 |positive long value
{table}
~Common goal attributes (BDI flags)~

```

The `posttoall` flag enables parallel processing of a goal by dispatching the goal to all applicable plans at once. The first plan to reach the goal “wins” and all other plans are terminated. When all plans terminate without achieving the goal, it is regarded as failed. The `randomselection` flag can be used to choose among applicable plans for a given goal randomly. Using this flag, the order of plan declarations within the ADF becomes unimportant, i.e., random selection is only applied to plans of the same priority and rank.

The `metalevelreasoning` flag activates the meta-level reasoning for processing of that goal. Meta-level reasoning means, that the selection among the applicable plans for a given goal (or event) is shifted to a meta-level. This is done by the system by creating a meta-level goal which subsequently needs to be handled by a meta-level plan, which actually has to make the decision and return the result. As the description indicates this process could be made recursive to further meta-meta levels if more than one meta-plan is applicable for the meta-goal, but in our experience this is only a theoretical issue without practical relevance. In Jadex the meta-goals and plans need to be explicitly defined within an ADF. From this circumstance the Meta Goal type is derived which will be explained in more detail in the following section about meta goals.

Furthermore the goal introduce the `recur` flag and the `recurdelay` (in milliseconds) option as further BDI settings. Consider a goal to have failed after trying all available plans. Setting `recur` to true, this goal will not be dropped but try to execute again, when the specified delay has elapsed. For recurring goals, all plans are considered again, i.e. `recur` allows to completely restart the reasoning for a goal after some delay.

1.1 Perform Goal

Perform goals are conceived to be used when certain activities have to be done. Below, an example declaration from the cleaner world example is shown. You can see that the perform goal “patrol” refines some BDI flags to obtain the desired behavior. By allowing the goal to redo activities (`retry=“true”`), it is assured that the agent does not conclude to knock off after having performed one patrol round, but instead patrols as long as it is night. Even when the agent only knows one patrol plan, it will reuse this plan and perform the same patrol rounds, because it is not allowed to exclude a plan (`exclude=“never”`).

```
{code:xml}
<performgoal name="performlookforwaste" retry="true" exclude="never">
  <contextcondition language="jcl">
    $beliefbase.daytime
  </contextcondition>
</performgoal>
{code}
~Example perform goal~
```

1.1 Achieve Goal

Achieve goals are used to reach some desired world state. Therefore, they extend the presented common goal features by adding a `<targetcondition>`. With the target condition it can be specified in what cases a goal can be considered as achieved. The opposite can be specified using the drop condition that can be found in all goal types. If no target condition is specified, the results of the plan executions are used to decide if the goal is achieved. In contrast to a perform goal, an achieve goal without target condition is completed when the first plan completes without error, while the perform goal would continue to execute as long as more applicable plans are available. Below another goal specification from the cleaner world example is shown. The "moveto" goal tries to bring the agent to a target position as specified in the location parameter. The goal has been reached, when the agent's position is near the target position as described in the target condition.

```
{code:xml}
<achievegoal name="moveto">
  <parameter name="location" class="Location"/>
  <targetcondition>
    $beliefbase.my_location.isNear($goal.location)
  </targetcondition>
</achievegoal>
{code}
~Example achieve goal~
```

1.1 Query Goal

Query goals can be used to retrieve specified information. From the specification and runtime behavior's point of view they are very similar to achieve goals with one exception. Query goals exhibit an implicit target condition by requesting all out parameters to have a value other than null and out parameter sets to contain at least one value. Therefore, a query goal automatically succeeds, when all out parameter(set)s contain a value. The agent will engage into actions by performing plans only, when the required information is not available. Below, the "query_wastebin" example realizes a query goal to find the nearest not full waste bin. It defines an out parameter, which contains a query expression. If

one or more not full waste bins are already known by the agent and therefore contained in the wastebins belief set, the result will be set to the nearest waste bin calculated from the agent’s current position (as described in the order by clause). Otherwise the agent does not know any not full waste bin and will try to reach the goal by using matching plans. Note that in the example the “\&” entity is used to escape the AND character (“\&”) in XML.

```
{code:xml}
<querygoal name="query_wastebin" exclude="never">
  <parameter name="result" class="Wastebin" evaluationmode="push"
direction="out">
    <value variable="$wastebin">
      Wastebin $wastebin &amp;&amp; !$wastebin.isFull()
      &amp;&amp; !(Wastebin $wastebin2 &amp;&amp; !$wastebin2.isFull()
      &amp;&amp; $beliefbase.my_location.getDistance($wastebin.getLocation())
      > $beliefbase.my_location.getDistance($wastebin2.getLocation()))
    </value>
  </parameter>
</querygoal>
{code}
~Example query goal~
```

1.1 Maintain Goal

Maintain goals allow a specific state to be monitored and whenever this state gets violated, the goal has the purpose to reestablish its original maintain state. Hence it adds a mandatory \<maintaincondition\> tag for the specification of the state to observe. Sometimes it is desirable to be able to refine the maintain state for being able to define more accurately what state should be achieved on a violation of the maintained state. Therefore the optional \<targetcondition\> can be declared.

Note that maintain goals differ from the other kinds of goals in that they do not necessary lead to actions at once, but start processing automatically on demand. In addition, maintain goals are never finished according to actions or state, so the only possibility to get rid of a maintain goal, is to drop it either by specifying a drop condition or by dropping it from a plan.

The maintain goal “battery_loaded” shown below, makes sure that the cleaner agent recharges its battery whenever the charge state drops under 20%. To avoid the agent moving to the charging station and loading only until 21% (which satisfies the maintain condition), the extra target condition is used. It ensures that the agent stays loading until the battery is fully recharged. <!~~Note that in the example the “\>” entity is used to escape the greater than character (“>”) in XML.~~>

```
{code:xml}
<maintaingoal name="maintainbatteryloaded">
```

```

<deliberation>
  <inhibits ref="performlookforwaste" inhibit="when_in_process"/>
  <inhibits ref="achievecleanup" inhibit="when_in_process"/>
  <inhibits ref="performpatrol" inhibit="when_in_process"/>
</deliberation>
<maintaincondition language="jcl">
  $beliefbase.my_chargestate > 0.2
</maintaincondition>
<targetcondition language="jcl">
  $beliefbase.my_chargestate >= 1.0
</targetcondition>
</maintaingoal>
{code}
~Example maintain goal~

```

1.1 Creating and Dispatching New Goals

Jadex distinguishes between top-level goals and subgoals. Subgoals are created in the context of a plan, while top-level goals exist independently from any plans. When a plan terminates or is aborted, all its not yet finished subgoals are dropped automatically. There are four ways to create and dispatch new goals: Goals can be contained in the configuration of an agent or capability, and are directly created and dispatched as top-level goals when an agent is born or terminated. In addition, goals are automatically created and dispatched as top-level goals, when the goal's creation condition triggers. Subgoals may be created inside plans only, while top-level goals may be created manually from plans, as well as from external interactions [cf. Chapter External Interactions>14 External Interactions].

When a plan wants to dispatch a subgoal or make the agent adopt a new top-level goal it also has to create an instance of some goal model. For convenience a method `~createGoal()~` is provided in `~jadex.runtime.AbstractPlan~` that automatically performs the necessary goal lookup for the model element of the new goal instance. The name therefore specifies the goal model to use as basis for the new `~IGoal~`.

A subgoal is dispatched as child of the root goal of the plan, and remains in the goal hierarchy until it is finished or aborted. To start processing of a subgoal, the plan has to dispatch the goal using the `~dispatchSubgoal()~` method. When the subgoal is finished (e.g., failed or succeeded), an appropriate notification will be generated, which can be handled by the plan that created the subgoal. A `~dispatchSubgoalAndWait()~` method is provided in the `Plan` class, which dispatches the goal and waits until the goal is completed. Alternatively to subgoals, the plan can make the agent adopt a new top-level goal by using the `~dispatchTopLevelGoal()~` method. Further on, a plan may at any time decide to abort one of its subgoals or a top-level goal by using the `~drop()~` method of the goal. Note, that a goal cannot be dropped when it is already finished.


```

{code:java}
public void body()
{
    Create new top-level goal.
    IGoal goal1 = createGoal("mygoal");
    dispatchTopLevelGoal(goal1);
    ...
    Create subgoal and wait for result.
    IGoal goal2 = createGoal("mygoal");
    dispatchSubgoalAndWait(goal2);
    Object val = goal2.getParameter("someoutparam").getValue();
    ...
    Drop top-level goal.
    goal1.drop();
}
{code}
~Dispatching goals from plan bodies~

```

When dispatching and waiting for a subgoal from a standard plan, a goal failure will be indicated by a `GoalFailureException` being thrown. Normally, this exception need not be caught, because most plans depend on all of their subgoals to succeed. If the plan may provide alternatives to failed subgoals, you can use `try/catch` statements to recover from goal failures:

```

{code:java}
public void body()
{
    ...
    Goal failure will cause plan to fail.
    dispatchSubgoalAndWait(goal1);

    Goal failure will not cause plan to fail.
    try
    {
        dispatchSubgoalAndWait(goal2);
    }
    catch(GoalFailureException e)
    {
        Recover from goal failure.
    }
    ...
}
{code}
~Handling of failed subgoals~

```

1.1 Goal Deliberation with “Easy Deliberation”

One aspect of rational behavior is that agents can pursue multiple goals in

parallel. Unlike other BDI systems, Jadex provides an architectural framework for deciding how goals interact and how an agent can autonomously decide which goals to pursue. This process is called goal deliberation, and is facilitated by the goal lifecycle (introduced in [Chapter 2, BDI Concepts>02 Concepts]) and refined here to facilitate the understanding of the strategy. The life cycle introduces the *~active~*, *~option~*, and *~suspended~* states. The context condition of a goal specifies which goals can possibly be pursued, and which goals have to be suspended. A goal deliberation strategy then has the task to choose among the possible (i.e., not suspended) goals by activating some of them, while leaving the others as options (for later processing).

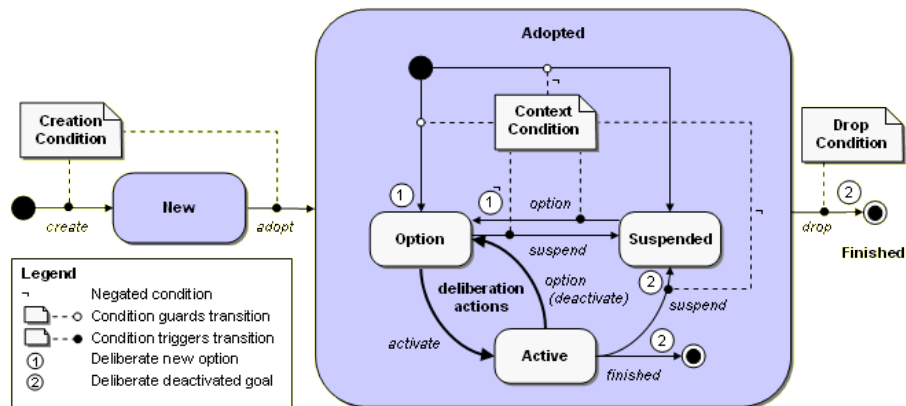


Figure 57:

~Figure 3: Easy deliberation strategy~

The current release of Jadex includes a goal deliberation strategy called *~Easy Deliberation~*, which is designed to allow agent developers to specify the relationships between goals in an easy and intuitive manner. It is based on goal cardinalities, which restrict the number of goals of a given type that may be active at once, and goal inhibitions, which prohibit certain others goal to be pursued in parallel. The figure above shows which goal transitions the strategy uses. On the one hand it may transfer an inhibited goal from the active to the option state and on the other hand it can reactivate an option by activating it. More details and scientific background about the Easy Deliberation strategy and goal deliberation in general can be found in [Pokahr et al. 05].

The goal deliberation settings are included in the goal specification in the ADF Using the `<deliberation>` tag. The cardinality is specified as an integer value in the cardinality attribute of the `<deliberation>` tag. The default is to allow an unlimited number of goals of a type to be processed at once. Inhibition arcs between goal types are specified using the `~ref~` attribute of the `<inhibits>` tag, which specifies the name of the goal to inhibit. Per default, any instance

of the inhibiting goal type inhibits any instance of the referenced goal type. An expression can be included as content of the `inhibits` tag, in which case the inhibition only takes effect when the expression evaluates to true. Using the expression variables `$goal` and `$ref`, fine-grained instance-level inhibition relationships may be specified. Some goals, such as idle maintain goals, might not always be in conflict with other goals, therefore it is sometimes required to restrict the inhibition to only take effect when the goal is in process. This can be specified with the `inhibit` attribute of the `<inhibits>` tag, using “when_active” (default) or “when_in_process” as appropriate. For a better understanding of the goal deliberation mechanism in the following the deliberation settings of the cleanerworld example will be explained.

`<xref linkend=“goals.deliberation.example.fig”/>`

shows the dependencies between the goals of a cleaner agent (cf. package `jadex.bdi.examples.cleanerworld`). The basic idea is that the cleaner agent (being an autonomous robot) has at daytime the task to look for waste in some environment and clean up the located pieces by bringing them to a near waste-bin. At night it should stop cleaning and instead patrol around to guard its environment. Additionally, it always has to monitor its battery state and reload it at a charging station when the energy level drops below some threshold.

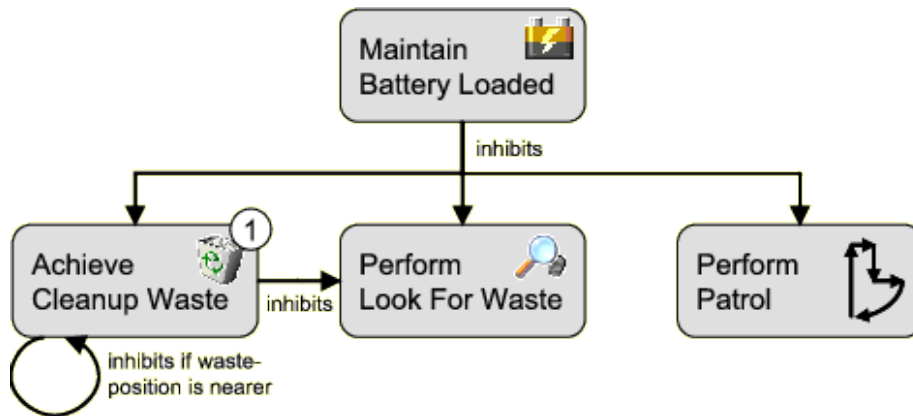


Figure 58:

~Figure 4: Example goal dependencies (taken from Cleanerworld scenario)~

The dependencies can be naturally mapped to the goal specifications in the ADF

(see `<xref linkend=“goals.deliberation.example.xml”/>`).

The `<inhibits>` tags are used to specify that the “maintainbatteryloaded” goal is more important than the other goals. As the “maintainbatteryloaded” is a

maintain goal, it only needs to precede the other goals when it is in process, i.e., the cleaner is currently recharging its battery. The cardinality of the “achievecleanup” goal specifies, that the agent should only pursue one cleanup goal at the same time. The goal inhibits the “performlookforwaste” goal and additionally introduces a runtime inhibition relationship to other goals of its type. The expression contained in the inhibits declaration means that one “achievecleanup” goal should inhibit other instances of the “achievecleanup” goal, when its waste location is nearer to the agent.

```
{code:xml}
<maintaingoal name="maintainbatteryloaded">
  <deliberation>
    <inhibits ref="performlookforwaste" inhibit="when_in_process"/>
    <inhibits ref="achievecleanup" inhibit="when_in_process"/>
    <inhibits ref="performpatrol" inhibit="when_in_process"/>
  </deliberation>
</maintaingoal>

<achievegoal name="achievecleanup" retry="true" exclude="when_failed">
  <parameter name="waste" class="Waste" />
  <!--Omitted conditions for brevity.-->
  <deliberation cardinality="1">
    <inhibits ref="performlookforwaste"/>
    <inhibits ref="achievecleanup">
      $beliefbase.my_location.getDistance($goal.waste.getLocation())
      < $beliefbase.my_location.getDistance($ref.waste.getLocation())
    </inhibits>
  </deliberation>
</achievegoal>
{code}
```

~Example goals (taken from Cleanerworld scenario)~

1.1 Meta Goal

Meta Goals are used for meta-level reasoning. This means, whenever an event or goal is executed and it is determined that meta-level reasoning needs to be done (i.e., because there are multiple matching plans) the corresponding meta-level goal of the goal or event is created and dispatched. Corresponding meta-level plans are then executed to achieve the meta goal (i.e., find a plan to execute). When the meta goal is finished the result contains the selected plans, which are afterwards scheduled for execution.

With the trigger tag, it is specified for which kind of event or goal the meta goal should be activated. Possible meta goal triggers are shown in Figure 5. As can be seen, meta goals can be used to select among applicable plans for an internal event, message event, a goal finished event, and a new goal to process. Any number of these triggers can appear inside a meta goal specification, i.e., a meta goal can be used to control meta-level reasoning of more than one event

type. Each triggering element can be further described using a match expression, which can be used, e.g., to match only elements with given parameter values. For backwards compatibility it is also possible to specify the triggering events in form of a single filter expression. This should only be needed in very special cases and should otherwise be avoided, because support for filter expressions might be dropped in future releases of Jadex.

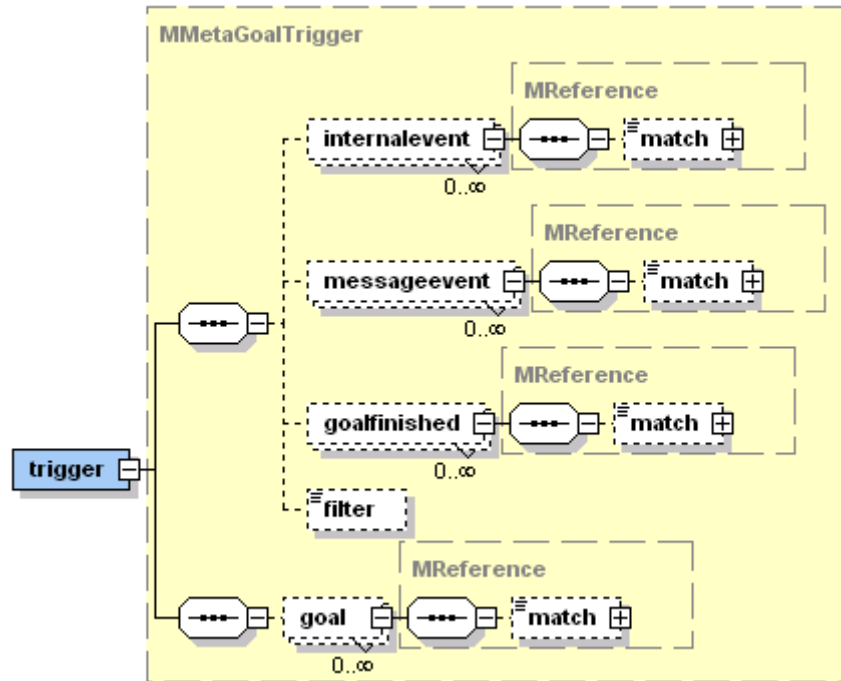


Figure 59:

~Figure 5: Meta goal trigger tag~

Besides the declaration of a triggering goal or event, the specification of a meta goal requires including the in parameter set “applicables” and the out parameter set “result” (both of type `jadex.bdi.runtime.ICandidateInfo`). The applicables are filled in by the system, while the result is set by the meta-level plan executed to achieve the meta goal. Furthermore, a failure condition can be specified (similar to query goals) as meta goals are also used for information retrieval (to find a plan to execute for a goal resp. event). Meta-goals are only created internally by the system when the demand for meta-level reasoning arises. Therefore, in contrast to the query goal and the other goal types presented here, meta-goals exhibit several restrictions, as for these kinds of goals creation condition, unique settings and binding parameters are not allowed. On the other hand, meta-plans do not differ from other plans (there is no a separate tag for meta plans). A plan is a meta plan, when its plan trigger contains a meta goal.

In the example below, adapted from the `jadex.bdi.examples.puzzle` example, for every “makemove” goal a large number of plan instances might be applicable, as the “move_plan” has a binding option which always contains all possible moves. Therefore, the “choosemove” meta goal is used to decide which of the applicable “move_plan” instances should be executed. In turn, handling the “choosemove” meta goal another plan is executed (“choose_move_plan”). As you can see in the plan body code snippet below, the “choose_move_plan” has access to the parameters of applicable plans and may use this information to decide which plan(s) to execute. The selected plans are placed in the “result” parameter of the “choose_move_plan” goal.

```
{code:xml}
<goals>
  <achievegoal name="makemove">
    ...
  </achievegoal>

  <metagoal name="choosemove" recalculate="false">
    <parameterset name="applicables" class="ICandidateInfo"/>
    <parameterset name="result" class="ICandidateInfo" direction="out"/>
    <trigger>
      <goal ref="makemove"/>
    </trigger>
  </metagoal>
</goals>

<plans>
  <plan name="move_plan">
    <parameter name="move" class="Move">
      <bindingoptions>$beliefbase.board.getPossibleMoves()</bindingoptions>
    </parameter>
    ...
    <trigger>
      <goal ref="makemove"/>
    </trigger>
  </plan>

  <plan name="choose_move_plan">
    <parameterset name="applicables" class="ICandidateInfo">
      <goalmapping ref="choosemove.applicables"/>
    </parameterset>
    <parameterset name="result" class="ICandidateInfo" direction="out">
      <goalmapping ref="choosemove.result"/>
    </parameterset>
    <body class="ChooseMovePlan"/>
    <trigger>
```

```

    <goal ref="choosemove"/>
  </trigger>
</plan>
</plans>
{code}
~Example meta goal and corresponding plan~

{code:java}
public void body()
{
  ICandidateInfo[] apps = (ICandidateInfo[])getParameterSet("applicables").getValues();
  ICandidateInfo sel = null;

  for(int i=0; i<apps.length; i++)
  {
    Decide which plan to select, e.g. using the move parameter of the move_plan.
    Move move = (Move)apps[i].getPlan().getParameter("move").getValue();
    ...
  }
  getParameterSet("result").addValue(sel);
}
{code}
~Body of the ChooseMovePlan~

```

Chapter 8. Plans

Plans represent the agent's means to act in its environment. Therefore, the plans predefined by the developer compose the library of (more or less complex) actions the agent can perform. Depending on the current situation, plans are selected in response to occurring events or goals. The selection of plans is done automatically by the system and represents one main aspect of a BDI infrastructure. In Jadex, plans consist of two parts: A plan head and a corresponding plan body. The plan head is declared in the ADF whereas the plan body is realized in a concrete Java class. Therefore the plan head defines the circumstances under which the plan body is instantiated and executed.

Defining Plan Heads in the ADF

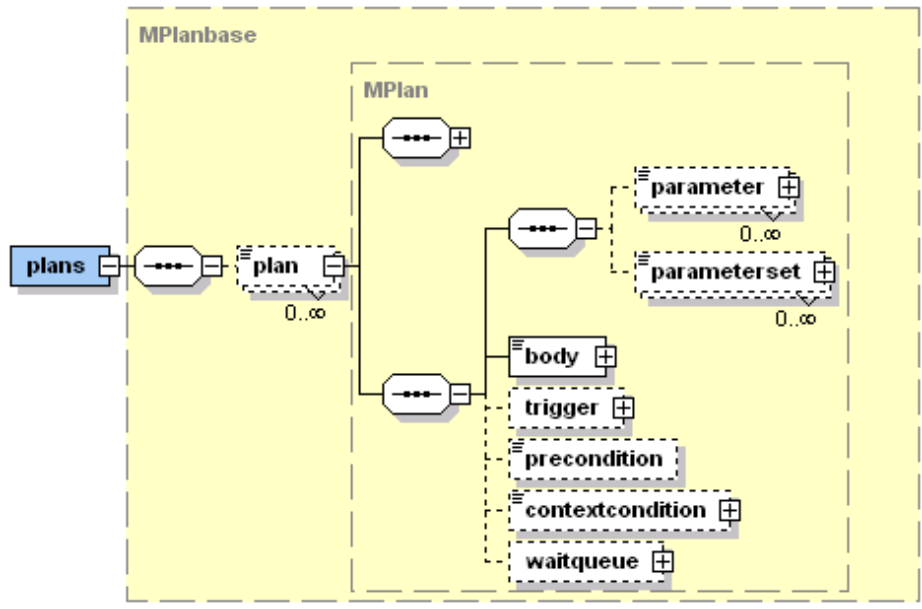


Figure 1: The Jadex plans XML schema part

In Figure 1 the XML schema part for the plans section is shown. Inside the `<plans>` tag an arbitrary number of plan heads denoted by the `<plan>` tag can be declared. For each plan head several attributes (as shown in the following Table) and contained elements can be defined. For each plan a name has to be provided. The priority of a plan describes its preference in comparison to other plans. Therefore it is used to determine which candidate plan will be chosen for a certain event occurrence, favouring higher priority plans (random selection, if activated, applies only to plans of equal priority). Per default all applicable plans have a default priority of 0 and are selected in order of appearance (or randomly when the corresponding BDI flag is set).

Tag	Attribute	Required	Default	Possible Values
plan	name	yes		
plan	priority	no	0	integer
body	impl	no		implementation class name
body	class	no		implementation file name
body	type	no	standard	standard, bpmn, ...

Important attributes of the plan and the body tag

For each plan the corresponding plan body has to be declared using the `<body>`

element. The “impl” attribute (for backwards compatibility also the “class” attribute) is used for defining the implementation of the plan. In case of standard Java plans the classname is used to identify the implementation. The type attribute determines which kind of plan body is used. Currently, the available options are *standard* and *bpmn* plan bodies as further described in the following sections. To clarify things, a simple example ADF is given below that shows the declaration of a plan reacting on a ping message.

```
<agent ...>
  ...
  <plans>
    <plan name="ping">
      <body impl="PingPlan"/>
      <trigger>
        <messageevent ref="query_ping"/>
      </trigger>
    </plan>
  </plans>
  ...
  <events>
    <messageevent name="query_ping" type="fipa">
      ...
    </messageevent>
  </events>
  ...
</agent>
```

A plan reacting on a ping message

Plan Triggers

To indicate in what cases a plan is applicable and a new plan instance shall be created the <trigger> tag can be used (see Figure 2). Its subtags specify the goals, internal- or message events for which the plan is applicable.

For goals, it is distinguished between plans triggered for handling a goal (<goal>tag) and plans reacting on the completion of a goal (<goalfinished>tag). The <goal> tag indicates that a goal has been newly activated, while the <goalfinished> tag corresponds to a goal that has been dropped.

These events or goals can be further restricted, by specifying a match expression. When a match expression is included in the trigger element, the plan will only be selected for those goal or event instances, for which the expression evaluates to true. For backwards compatibility to older Jadex versions, additionally, a filter instance can be used, although its use is discouraged, because of the lack of declarativeness and readability.

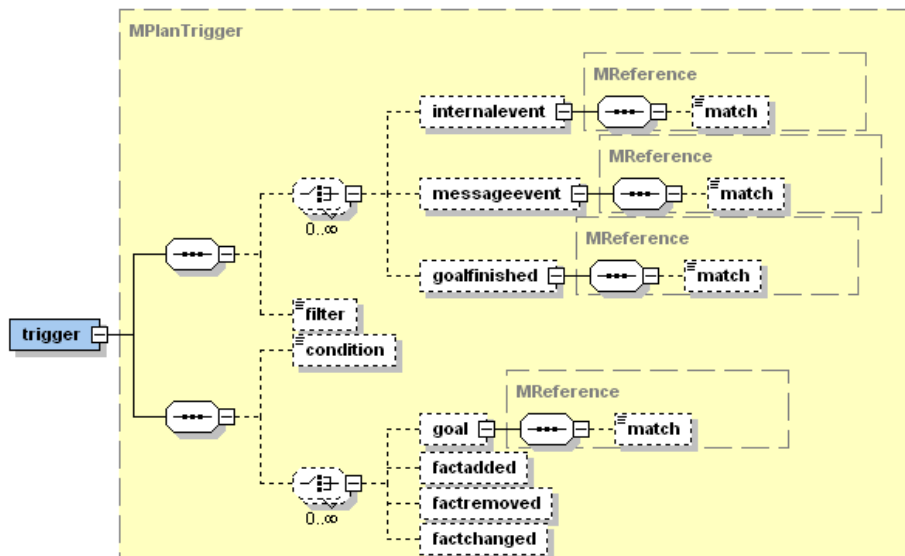


Figure 60:

Figure 2: The Jadex plan trigger XML schema part

In addition to the reaction on certain event or goal types, it is also possible to define data-driven plan execution by using the `<condition>` tag. A trigger condition can consist of arbitrary boolean Jadex expressions, which may refer to certain beliefs when their states needs to be supervised. If only some specific belief needs to be monitored the `<factchanged>` tag can be used. In this respect a belief change is reported whenever the belief's new fact value is different from the value held before. Similarly, belief sets can be monitored for addition or removal of facts by using the tags `<factadded>` and `<factremoved>` respectively.

Defining Plan Applicability with Pre- and Context Conditions

To find out if the plan is applicable not only with respect to the current event or belief change but also considering the current situation, the pre- and context conditions can be used. The precondition is evaluated before a plan is instantiated and when it is not fulfilled this plan is excluded from the list of applicable plans. In contrast, the context condition is evaluated during the execution of a plan and whenever it is violated the plan execution is aborted and the plan has failed. Both conditions can be specified in the corresponding tags supplying some boolean Jadex term (please note that the precondition must be specified in the expression and the context condition in the condition language). The

following example shows how to execute a “repair” plan whenever the belief “out_of_order” becomes *true*, and as long as the agent believes to be repairable.

```
<plans>
  <plan name="repair">
    <body impl="RepairPlan"/>
    <trigger>
      <condition>$beliefbase.out_of_order</condition>
    </trigger>
    <contextcondition>$beliefbase.repairable</contextcondition>
  </plan>
</plans>
```

Example of a plan with context condition

Waitqueue

When an event occurs, and triggers an execution step of a plan, it may take a while, before the plan step is actually executed, due to many plans being executed concurrently inside an agent. Therefore, it is sometimes possible, that a subsequent event, which might be relevant for a plan, is not dispatched to that plan, because it still has to execute previous plan step, and does not yet wait for the event. To avoid this, each plan has a waitqueue to collect such events. The waitqueue for a plan is set up using the `<waitqueue>` tag or the `getWaitqueue()` method in plan bodies. The waitqueue of a plan is always matched against events, although the plan may not currently wait for that specific event. The `<waitqueue>` tag provides support for waiting for finished goals and the occurrence of message and internal events. Events that match against the waitqueue of a plan are added to the plan's internal waitqueue. They will be dispatched to the plan later, when it calls `waitFor()` or `getWaitqueue().getElements()`. You may have a look at the `jadex.bdi.runtime.IWaitqueue` interface for more details.

Parameters, Binding, and Parameter Mapping

Similar to goals, plans may have parameters and parameter sets, which can store local values, required or produced during the execution of the plan. Plan parameters can be accessed from plan bodies for read and write access depending on the parameter direction attribute: *in* parameters allow only read access, *out* parameters can only be written, while *inout* parameters allow both kinds of access. Default values for any of these parameters and parameter sets can be provided in the ADF. Just like facts for belief sets, initial values for parameter

sets can be either specified as a sequence of <value> tags, or as a single <values> tag. The parameter(set)s of a plan can also be accessed from the body tag or the context condition, by referencing the plan via the reserved variable \$plan concatenated with the parameter(set) name, e.g. \$plan.result. The precondition and the trigger condition are evaluated before the plan is instantiated, therefore from these conditions no parameters and parameter sets can be accessed with exception of the binding parameters. As binding parameters are evaluated before plan instantiation the value of a binding parameter can be accessed directly via its name (without prepending \$plan).

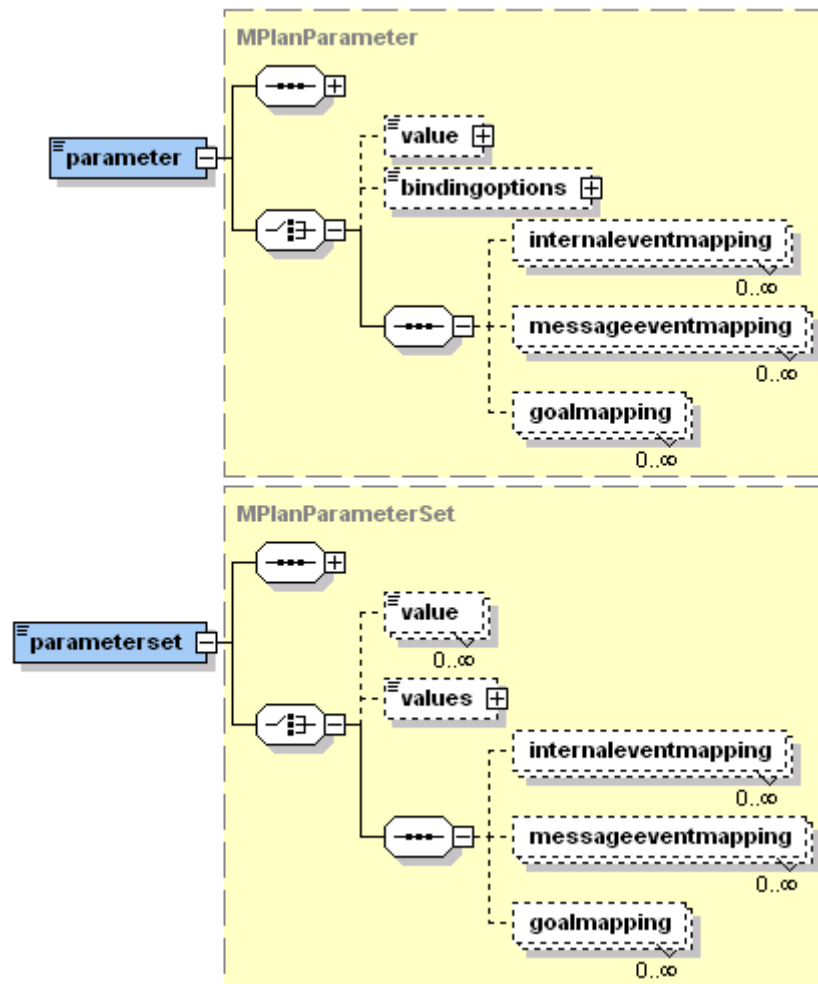


Figure 61:

Figure 3: The Jadex plan parameters XML schema part

For (single valued) parameters it is possible to use binding options instead of an initial value. A binding option is an expression, that will be evaluated to a collection of values (supported are arrays or an object implementing *Iterator*, *Enumeration*, *Collection*, or *Map*). The binding options of a parameter therefore represent a set of possible initial values for that parameter. The cartesian product of all binding parameters (if there is more than one parameter with binding options) determines the number of candidate plans that is considered in the event dispatching process. (In mathematics, the Cartesian product (or direct product) of two sets X and Y, denoted X x Y, is the set of all possible ordered pairs whose first component is a member of X and whose second component is a member of Y. Example: The cartesian product of {1,2}x{3,4} is {(1,3),(1,4),(2,3),(2,4)}, cf. Cartesian Product). Please note that the calculation of the cartesian product can easily lead to large numbers of applicable plans so that binding options should always be used with care. For example, the code example below shows a plan from the “puzzle” example, where for each possible move a plan instance is created. In addition to accessing the binding values like other parameters by writing *\$plan.paramname*, it is also possible to access the binding value directly via its name via *paramname*. This allows binding values also to be considered for evaluating the pre- and trigger condition, before the plan instance is created.

```
<plan name="move_plan">
  <parameter name="move" class="Move">
    <bindingoptions>$beliefbase.board.getPossibleMoves()</bindingoptions>
  </parameter>
  ...
</plan>
```

Example binding parameter (from the puzzle example)

A common use case for plan parameter(set)s is to capture parameter(set)s from a goal or event that triggered the plan. To make this relationship between event and plan parameters explicit, the `<internaleventmapping>`, `<messageeventmapping>`, and `<goalmapping>` tags can be used. A mapping definition contains a *ref* attribute denoting the event or goal parameter to be mapped. The reference is given in the form *type.param*, where *type* is the name of the goal or event, and *param* is the name of the goal or event parameter. When a plan parameter is mapped, the parameter properties like class and direction are ignored, as the values from the mapped parameter are used. Depending on the direction of the parameter, the default values of the plan parameter are automatically assigned from the event or goal (direction *in*, *inout*), and can also automatically be written back to a goal (direction *out*, *inout*), when the plan has finished. Note that when a plan reacts to more than one goal or event, you cannot just provide a mapping for one of these events. If you want to use a mapping for a parameter, you have to provide mappings for all events or goals handled by the

plan.

Implementing a Plan Body in Java

A plan body represents a part of the agent's functionality and encapsulates a recipe of actions. In Jadex, plan bodies are written in pure Java (or alternatively in bpmn) and therefore it is easily possible to write plans that access any available Java libraries, and to develop plans in your favourite Java Integrated Development Environment (IDE). The connection between a plan body and a plan head is established in the plan head, thus plan bodies can be reused within different plan declarations. For developing reusable plans, plan parameters in combination with parameter mappings from some triggering event or goal to/from the plan should be used.

As mentioned earlier, currently two types of plan bodies are supported in Jadex, which are both implemented as conventional Java classes. The standard plans inherit from *jadex.bdi.runtime.Plan*. The code of standard plans is placed in the *body()* method.

Plans that are ready to run are executed by the main interpreter (cf. Section 2). The system takes care that only one plan step is running at a time. The length of a plan step depends on the plan itself. The *body()* method of standard plans is called only once for the first step, and runs until the plan explicitly ends its step by calling one of the *waitFor()* methods, or the execution of the plan triggers a condition (e.g., by changing belief values). For subsequent steps the *body()* method is continued, where the plan was interrupted.

Below is shown a cutout of an example standard Java plan. It is a snippet of a protocol plan, which waits for messages and acts accordingly. Most interestingly are the *waitForXYZ()* calls in the plans, because these are the interruption points in the plan, i.e. the plan is interrupted when these calls are executed. Given that a *waitForMessageEvent()* method is invoked the plan is continued when a fitting message arrives (or a timeout occurs, if specified).

```
public void body()
{
    // Send request.
    ...

    // Wait for agree/refuse.
    IMessageEvent e1 = waitForMessageEvent(...);
    boolean agreed = ...;
    ...
}
```

```

// Wait for inform/failure.
if(agreed)
{
    IMessageEvent e2 = waitForReply(...);
    boolean informed = ...;

    ...
    if(informed)
    {
        ...
    }
    else
    {
        ...
    }
}
else
{
    ...
}
}

```

Example plan body

Plan Success or Failure and BDI Exceptions

If a plan completes without producing an exception it is considered as succeeded. Completion means for standard plans that the *body()* method returns. To perform cleanup after the plan has finished, you can override the *passed()*, *failed()*, and *aborted()* methods, which are called when the plan succeeds (runs through without exception), fails (e.g., due to an exception), or was aborted during execution (e.g., because the root goal was dropped or has been achieved before the plan reached its end). In [plans.plan_skeleton](#) a plan skeleton of a standard Jadex plan is depicted including all predefined methods. In the *failed()* method, a plan may call the *getException()* method to see which problem occurred. To find out whether the plan was aborted, because its root goal was achieved, you can call the *isAbortedOnSuccess()* method inside the *aborted()* method.

```

public class MyPlan extends Plan
{
    public void body()
    {

```

```

        // Application code goes here.
        ...
    }

    public void passed()
    {
        // Clean-up code for plan success.
        ...
    }

    public void failed()
    {
        // Clean-up code for plan failure.
        ...
        getException().printStackTrace();
    }

    public void aborted()
    {
        // Clean-up code for an aborted plan.
        ...
        System.out.println("Goal achieved? "+isAbortedOnSuccess());
    }
}

```

Standard plan skeleton

Regardless if standard or mobile plans are used, a plan is considered as failed if it produces an exception. To aid debugging, occurring exceptions are (by default) printed on the console (*logging.level.WARNING*), although the agent continues to execute. Subclasses of *jadex.bdi.runtime.BDIFailureException* are not printed, because they are produced by the system and indicate “normal” plan failure (*logging.level.INFO*). If you want your plan explicitly to fail without printing an exception, you can throw a *PlanFailureException* or, as a shortcut, call the *fail()* method. Other subclasses of the *BDIFailureException* are generated automatically by the system, to denote certain failures during plan execution. All of these exceptions can be explicitly handled if desired, or just ignored (causing the plan to fail). The *GoalFailureException*, is thrown, when a subgoal of a plan could not be reached or if the subgoal could not be adopted due to its uniqueness settings (i.e. there exists already a goal that is considered equal to the new one). The *MessageFailureException* indicates that a message could not be sent, e.g., because the receiver is unknown. A *TimeoutException* occurs when calling *waitFor()* with a timeout, and the awaited event does not happen. Finally, the *ComponentTerminatedException* is thrown when an operation could not be performed, because the agent has died. This usually does not occur inside plans, but only when accessing an agent from external processes.

Atomic Blocks

Standard plans might be interrupted whenever the agent regards it as necessary, e.g., when a belief has been changed leading to the adoption of a new goal. Sometimes it is desirable that a sequence of actions is considered as a single atomic action. For example when you change multiple beliefs at once, which might trigger some conditions, you may want to perform all changes before the conditions are evaluated. In standard plans, this can be achieved by using a pair of *startAtomic()* / *endAtomic()* calls around the code you want to execute as a whole. Note that you are not allowed to end the plan step inside an atomic block (e.g., by calling *waitFor()*).

```
public void body()
{
    ...
    startAtomic();
    // Atomic code goes here.
    ...
    endAtomic();
    ...
}
```

How to establish an atomic block

Goal Processing

The mechanism to make a plan reacting to a specific goal is for all kinds of goals the same. A trigger matching goal events has to be defined within the plan declaration.

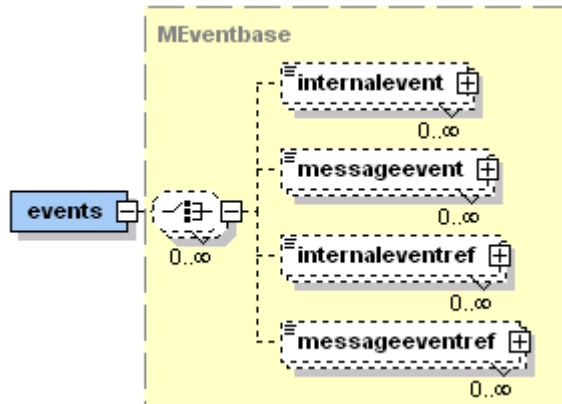
```
<plan name="wanderplan">
  <constructor>...</constructor>
  <trigger>
    <goal ref="patrol" />
  </trigger>
</plan>
```

Plan triggered by a goal

1 Chapter 9. Events

An important property of agents is the ability to react timely to different kinds of events. Jadex supports two kinds of application-level events, which can be defined by the developer in the ADF. Internal events can be used to denote an

occurrence inside an agent, while message events represent a communication between two or more agents. Events are usually handled by plans. For example the ping plan gets triggered when a ping request message arrives. When an event occurs in Jadex and no plan is found to handle this event a warning message is generated, which can be printed to the console depending on the logging settings (see [Properties>12 Properties]).



~Figure 1: The Jadex events XML schema part~

Two kinds of events are supported in Jadex: message events and internal events, as well as references to these types, as shown in Figure 1. Generally, all event types are parameter elements meaning that any number of parameter(set)s can be specified for a user-defined kind of event. A parameter itself can be used for passing values from the source to the consumer of the event. Unlike goals, events are single points in time and therefore only support “in” parameters, which denote the “source to consumer”-direction of value passing. In addition all kinds of events share the common attributes: “posttoall” and “randomselection”. The “posttoall” flag determines if the event should be dispatched to a single receiver or to all applicable plans. The “randomselection” flag can be used to turn off the importance of the declaration order of plans for the plan selection process. Nevertheless, the priority of a plan is still respected.

Flags	Default Value
posttoall	internal event=true, otherwise false
randomselection	false

~Event Flags~

At runtime, e.g., when accessed from plans, instances of the elements are represented by the `~jadex.bdi.runtime.IMessageEvent~` and `~jadex.bdi.runtime.IInternalEvent~` interfaces. Both interfaces inherit from the common super interface `~jadex.bdi.runtime.IEvent~`. The following sections will describe these different

types of events in more detail.

1.1 Internal Events

Internal events are the typical way in Jadex for explicit one-way communication of an interesting occurrence inside the agent. The usage of internal events is characterized by the fact that an information should be passed from a source to some consumers (similar to the object oriented observer pattern). Hence, if an internal event occurs within the system, e.g., because some plan dispatches one, it is distributed to all plans that have declared their interest in this kind of event by using a corresponding trigger or by waiting for this kind of internal event. The internal event can transport arbitrary information to the consumers if custom parameter(set)s are defined in the type for that purpose. A typical use case for resorting to internal events is, e.g., updating a GUI.

```
{code:xml}
...
<events>
  <internalevent name="gui_update">
    <parameter name="content" class="String"/>
  </internalevent>
</events>
```

```
...
{code}
~Example of the declaration of an internal event~
```

```
{code:java}
...
public void body()
{
  String update_info;
  ...
  "gui_update" internal event type must be defined in the ADF
  InternalEvent event = createInternalEvent("gui_update");
  Setting the content parameter to the update info
  event.getParameter("content").setValue(update_info);
  dispatchInternalEvent(event);
  ...
}
```

```
{code}
~Plan snippet showing the creation and dispatching of the internal event~
```

1.1 Message Events

All message types an agent wants to send or receive need to be specified within the ADF. The message event (class `~jadex.bdi.runtime.IMessageEvent~` denotes the arrival or sending of a message. Note, that only incoming messages are handled by the event dispatching mechanism, while outgoing messages are just sent. The native underlying message object (which is platform depen-

dent) can be retrieved using the `getMessage()` method. In addition, the message type, which may be FIPA or some other message format, can be retrieved using the `getMessageType()` method. The message type (subclass of `jadex.bridge.MessageType`) is meant to represent the way a message is composed, i.e. it defines which parameter and parametersets exists and (partially) what their meaning is. Hence, the message type can e.g. be used to determine which parameter(set) contains the receivers of the message. The message type is comparable to meta-information of a specific kind of message such as FIPA. It represents an extension point is Jadex and allows for introducing new message types without having to modify underlying message passing infrastructure. An agent developer normally can ignore the message type and use the default FIPA for all messages.

The message passing mechanism is based on using `jadex.bridge.IComponentIdentifier`s for the unambiguous identification of agents. A component identifier hence contains an agents globally unique name which consists of a local part followed by the “@” character and the platforms name (schema: `\<agentname\>@\<platformname\>`, example: `ams@lars`). In addition to the name a component identifier can contain additional information. On the one hand arbitrary many transport addresses might be present. These addresses can be used to contact the agent from a remote platform and normally represent the address of platform wide transport mechanisms (schema: `\<transportname\>:\<address\>`, example: `http://www.fipa.org/specs/fipa00023/SC00023J.html` [`http://www.fipa.org/specs/fipa00023/SC00023J.html`]).
A component identifier of another agent can be obtained in two ways. If all details about the agent are known an agent identifier can directly be created using a local or global name by using the `IComponentManagementService`. Please note that it is not allowed to create a component identifier via a simple constructor call, because the concrete implementation may vary in different platform infrastructures. The needed creation code is illustrated in the code snippet below. If the details of an agent are not known in advance, an agent may search for other agents using either the CMS listing all agents on a platform or the DF, which allows to search for agents providing a given service. Searching CMS and DF is explained in detail in [Predefined Capabilities>14 Predefined Capabilities].

```
{code:java}
IComponentManagementService ces= (IComponentManagementService)container.getService(IComponentManagementService.class);
IComponentIdentifier cid = ces.createComponentIdentifier("Heinz", true, null);
{code}
```

~Creation of a component identifier~

Templates for message events are defined in the ADF in the `\<events\>` section. The direction attribute can be used to declare whether the agent wants to receive, send or do both (default) for a given event. Possible values for that attribute are “send”, “receive” and “send_receive” respectively. The type of the message

constrains the available parameters of a message. Currently, the only available type is “fipa” which automatically creates parameter(set)s according to the FIPA message specification (e.g., parameters for the receivers, content, sender, etc. are introduced). Through this message typing Jadex does not require that only FIPA messages are being sent, as other options may be added in future. In the following table, all available parameter(set)s are itemized. For details about the meaning of the FIPA parameters, see the FIPA specifications available at [FIPA ACL Message Structure Specification><http://www.fipa.org/specs/fipa00061/SC00061G.html>](<http://www.fipa.org/specs/fipa00061/SC00061G.html>)]. The meanings of all of these parameters are explained in the following subsections.

{table}

Name	Class	Meaning
performative	String	Speech act of the message that expresses the senders intention towards the message content. You can use the constants from <code>jadex.base.fipa.SFipa.{ACCEPT_PROPOSAL, AGREE, ...}</code>
sender	IComponentIdentifier	The senders agent identifier, which contains besides other things its globally unique name.
reply_to	IComponentIdentifier	The agent identifier of the agent to which should be replied.
receivers	<code>\[set\]</code> IComponentIdentifier	Arbitrary many (at least one) agent identifier of the intended receiver agents.
content	Object	The content (string or object) of the message. If the content is an object it has to be specified how the content can be marshalled for transmission. For this purpose codecs are used. Jadex has built in support for marshalling arbitrary Java beans via setting the language of the message to <code>~jadex.base.fipa.SFipa.JADEX_XML~</code> .
language	String	The language in which the content of the message should be encoded.
encoding	String	The encoding of the message.
ontology	String	The ontology that can be used for understanding the message content. Can also be used for deciding how to marshal the content.
protocol	String	The interaction protocol of the the message if it belongs to a conversation. There are constants available for the predefined FIPA interaction protocols in <code>~jadex.base.fipa.SFipa.PROTOCOL_{REQUEST, QUERY, ...}~</code>
reply_with	String	Reply_with is used for assigning a reply to a original message. The receiver of the message should respond to this message by putting the reply_with value in the in_reply_to field of the answer. Unique ids can e.g. be generated via the method <code>SFipa.createUniqueId()</code> .
in_reply_to	String	Used in reply messages and should contain the reply_with content of the answered message.
conversation_id	String	The conversation_id is used in interactions for identifying messages that belong to a specific conversation. All messages of one interaction should share the same conversation_id. Unique ids can e.g. be generated via the method <code>SFipa.createUniqueId()</code> .
reply_by	Date	The reply_by field can contain the latest time for a response

message.
{table}
~Reserved FIPA message event parameters~

1.1 Receiving Messages

Typically in the ADF of an agent a number of message event types for sending and receiving message events are declared for the application domain. Examples for such user-defined message event types might be “inform_time”, “request_vision”, etc. As those message types are defined for each agent separately there are consequently no global message types. So how does an agent know the message type of a newly received message? For this purpose a simple matching process is used. This means that all locally known message types of an agent and its subcapabilities (with direction “receive” or “send_receive”) are matched against the newly received message and the best fitting is selected. For the matching process the parameter values of a message type are checked against the values in the received message. For this purpose only parameters with direction=“fixed” are considered important, as they represent fixed type information. In addition to fixed parameter values, message matching can be fine-tuned by using a match expression that can be specified for each message event. As shown in the second example below, in the match expression the parameters of a message can be accessed by prepending a “\$” before the parameter name. ~~Additionally, it is not allowed having variable names in Java that contain a “-” character as this is interpreted as minus. Therefore, in all parameter(set)s names the “-” characters have been replaced by a “_” character. This means you need to write e.g. “\$reply_with” instead of “\$reply-with”.~~ A message event type matches an incoming message if all fixed parameter values are the same in the received message and the match expression evaluates to true.

```
{code:xml}
<imports>
  <import>jadex.base.fipa.SFipa</import>
</imports>
...
<events>
  <!-- A query-ref message with content “ping” -->
  <messageevent name=“query_ping” type=“fipa” direction=“receive”>
    <parameter name=“performative” class=“String” direction=“fixed”>
      <value>SFipa.QUERY_REF</value>
    </parameter>
    <parameter name=“content” class=“String” direction=“fixed”>
      <value>“ping”</value>
    </parameter>
  </messageevent>

  <!-- An inform message where content contains the word “hello” -->
  <messageevent name=“inform_hello” type=“fipa” direction=“receive”>
```

```

    <parameter name="performative" class="String" direction="fixed">
      <value>SFipa.INFORM</value>
    </parameter>
    <match>((String)$content).indexOf("hello")!=-1</match>
  </messageevent>
</events>
{code}

```

~Examples for receiving messages~

There are several reasons why an agent may fail to correctly process an incoming message. These are indicated by different logging outputs at different logging levels (see [<xref linkend="events.message_matching"/>](#)). In the first case, if more than one message event type has a match with the incoming event the most specific match will be used. The number of fixed parameters and the presence of a match expression are used as indicator for the specificity. As this is a common case, it is only logged at level ~INFO~. When a message is received, which does not match any of the declared message events of the agent, a ~WARNING~ is generated, indicating that this message is ignored by the agent. Finally, when there are two or more message events, which all match an incoming message to the same degree (e.g., all have the same number of fixed parameters) the system cannot decide which message event to use, and has to choose one arbitrarily. As this probably indicates a programming error in the ADF, a ~SEVERE~ output is produced.

```

{table}
Level| Output
INFO| \<agentname\> multiple events matching message, using message event
with highest specialization degree
WARNING| \<agentname\> cannot process message, no message event matches
SEVERE| \<agentname\> cannot decide which event matches message, using
first

```

```
{table}
```

~Possible problems when matching messages~

1.1 Sending Messages

Messages to be sent also have to be declared in the ADF. The actual sending is usually done inside a plan, which instantiates the declared message event, fills in desired parameter values, and dispatches the message using one of the `<methodname>sendMessage...()` methods. The super class of both plan types (`~jadex.bdi.runtime.AbstractPlan~`) provides several convenience methods to create message events. To send a message, a message event has to be created using the `~createMessageEvent(String type)~` method supplying the declared message event type name as parameter. The receivers of fipa messages are specified by agent identifiers (interface `~jadex.bdige.IComponentIdentifier~`). The message content can be supplied as `String` or as `Object` with `~getParameter(SFipa.CONTENT).setValue(Object content)~`. If the content is provided

as Object it must be ensured that the agent can encode it into a transmissible representation as described in the following section about content languages.

To actually send the message event it is sufficient to call the `~sendMessage(IMessageEvent me)~` method with the prepared message event as parameter. It is also possible to send a message and directly wait for a reply with an optional timeout by using the `~sendMessageAndWait(IMessageEvent me [, timeout])~` method. This is described in

```
{code:xml}
<imports>
  <import>jadex.base.fipa.SFipa</import>
</imports>
...
<events>
  <!A query-ref message with content "ping">
  <messageevent name="query_ping" type="fipa" direction="send">
    <parameter name="performative" class="String">
      <value>SFipa.QUERY_REF</value>
    </parameter>
    <parameter name="content" class="String">
      <value>"ping"</value>
    </parameter>
  </messageevent>
</events>
{code}
~Example of declaration for a message~

{code:java}
public void body()
{
  IMessageEvent me = createMessageEvent("query_ref");
  me.getParameterSet(SFipa.RECEIVERS).addValue(cid);
  me.getParameter(SFipa.CONTENT).setValue("ping 2"); Set/change content if
  necessary
  sendMessage(me);
}
{code}
~Plan snippet showing the creation and sending of the message~
```

1.1 Using Ontologies and Content Languages

Message based communication allows that agents can communicate even when they are distributed across the network. One important property in the context of message based communication is the separation of address spaces, i.e., that

agents do not have direct access to the data inside other agents. Therefore data needs to be encoded into a message before sending and decoded from a message after receipt. In the context of multi-agent systems, so called content languages and ontologies are responsible for describing how data should be encoded into messages. A content language defines the syntactical mechanism used to represent data and an ontology specifies the meaning of the concepts used in the message. Together, content language and ontology assure a shared common understanding among agents.

The data inside a Jadex agent is usually represented as a collection of Java objects referencing each other. The Jadex framework provides some useful features that allow to encode/decode object structures, such that they can be used directly for the communication between agents. For this purpose, the agent knows about so called ~content codecs~, some of which are available by default, but can also be extended with custom codecs by the agent programmer. These codecs are selected automatically, when sending and receiving messages and are used to encode or decode the content of a message. From the viewpoint of an agent programmer, the agent is just sending or receiving messages containing Java objects. All the encoding and decoding works behind the scenes.

Two simple examples for sending and receiving a Java object inside a message are shown below (taken from the marsworld classic example). These examples use a ~Target~ object from package ~jadex.bdi.examples.marsworld_classic~. On the sender side, the message defines to use the language ~SFipa.JADEX_XML~, which is per default available in each agent.

The corresponding codec can handle arbitrary Java Beans (i.e. Java objects, which provide public getter and setter methods for their properties). For detailed information about JavaBean you should have a look at the [JavaBeans Specification><http://java.sun.com/products/javabeans/docs/spec.html>](<http://java.sun.com/products/javabeans/docs/spec.html>)]

```
{code:xml}
<!--Message declaration in the ADF-->
<messageevent name="inform_target" type="fipa" direction="send">
  <parameter name="performative" class="String" direction="fixed">
    <value>SFipa.INFORM</value>
  </parameter>
  <parameter name="language" class="String" direction="fixed">
    <value>SFipa.JADEX_XML</value>
  </parameter>
  <parameter name="ontology" class="String" direction="fixed">
    <value>MarsOntology.ONTOLOGY_NAME</value>
  </parameter>
</messageevent>
{code}
```

~Message template sending declaration~

```
{code:java}
public void body()
{
    Message sending in the plan.
    IComponentIdentifier receiver = ...
    Target target = ...
    IMessageEvent me = createMessageEvent("inform_target");
    me.getParameterSet(SFipa.RECEIVERS).addValue(receiver);
    me.getParameter(SFipa.CONTENT).setValue(target); The Java object is
    directly used as content.
    sendMessage(me);
}
{code}
```

~Example of sending an object inside a message~

As the decoded object is already available for matching an incoming message, on the receiver side, the match expression can be used to only match messages containing a ~Target~ object.

```
{code:xml}
<!--Message declaration in the ADF-->
<messageevent name="target_inform" type="fipa" direction="receive">
  <parameter name="performative" class="String" direction="fixed">
    <value>SFipa.INFORM</value>
  </parameter>
  <parameter name="ontology" class="String" direction="fixed">
    <value>MarsOntology.ONTOLOGY_NAME</value>
  </parameter>
  <match>$content instanceof Target</value>
</messageevent>
{code}
```

~Message template receiving declaration~

```
{code:java}
public void body()
{
    Message receiving in the plan.
    IMessageEvent msg = (IMessageEvent)getReason();
    Target target = (Target)msg.getParameter(SFipa.CONTENT).getValue();
    ...
}
{code}
```

~Example of receiving an object inside a message~

Three content languages are predefined in Jadex itself and therefore

are available on all platforms. These languages are defined in the constants `~SFipa.JAVA_XML~` and `~SFipa.NUGGETS_XML~` and `~SFipa.JADEX_XML~`. All three are Java bean converters. The Java XML language uses the bean encoder available in the JDK, to convert Java objects adhering to the JavaBeans specification to standardized XML files. The nuggets XML language is a proprietary language in Jadex, that works similar to the Java XML language but the encoding and decoding is much faster. Finally, the third alternative is also a Jadex variant, which is part of the Jadex XML databinding framework and is meant to replace nuggets in the long term. All languages allow marshalling content objects independently from the underlying ontology as they rely completely on the Java Bean specification. Using these languages requires that Java bean information about the content object can be found or inferred by the Java bean introspector. ~~<!Please have a look at the Beanyizer tool (available from the <mlink url="http://vsis-www.informatik.uni-hamburg.de/projects/jadex/|http://vsis-www.informatik.uni-hamburg.de/projects/jadex/)">Jadex homepage</mlink> if you are interested in converting an ontology to Java beans including the necessary bean infos.> Other content languages are available depending on the underlying platform (e.g. the JADE platform supports the FIPA SL language). <!The usage of these platform-specific languages is described in <xref linkend="adapters"/>.>~~

If you want to use your own mechanism for encoding and decoding of message contents, you can implement the interface `~IContentCodec~` from package `~jadex.bridge~`. The interface requires you to implement three methods. The `~match()~` method is used by Jadex, to determine if your codec applies to a given message. For this decision, the important message properties (e.g. language and ontology) are supplied. The other two methods are called to `~encode()~` an object to a string for sending and to `~decode()~` a string back to an object, when receiving a message. To register a custom content codec in an agent, it is sufficient to add a property starting with `~contentcodec.~` in the properties section of an agent:

```
{code:xml}
<properties>
  <property name="contentcodec.my_codec">new MyContentCodec()</property>
</properties>
{code}
~Include a custom content codec~
```

1.1 Using Conversations for Managing Sequences of Messages

Normally messages are not sent in isolation, but occur inside a conversation of many messages that are sent and received. Because of this, you often want to identify a certain message as belonging to a specific conversation or being a direct reply to some other message sent before. In the FIPA message structure, three parameters are responsible for this kind of conversation management. A unique `~conversation_id~` can be used to group together several messages

belonging to a single conversation. In addition, the `~in_repy_to~` parameter allows to identify a message as being an answer to a previous message with a corresponding `~reply_with~` parameter value.

In Jadex, the relation between messages is used to achieve two things: First, it allows to wait for a specific message while ignoring other messages that do not belong to an ongoing conversation or are a reply to another message. Thanks to this, e.g., when two plans simultaneously wait for the same type of message, a received message will automatically be posted to the correct plan, from which the previous message of the conversation was sent.

Second, it allows to restrict message receipt to a certain capability, namely the capability from which an earlier message was sent. This means, e.g., that if an agent defines two similar message events in two different capabilities (as is commonly the case, when the same capability is included twice in an agent), the message will automatically be routed to the correct capability where the corresponding conversation originated.

In both cases, the mechanism is based on the usage of the `~conversation_id~` and/or `~in_repy_to~` and `~reply_with~` parameters. The developer has to make sure that, when sending an initial message, a useful value has been set to one or more of these parameters. When replying to an initial message (by using `~msg.createReply(...)`), the parameter values are set automatically, based on the values of the initial message (i.e. the `conversation_id` is retained while the `reply_with` is copied to the `in_repy_to` parameter). The setting of initial parameter values can directly be done in the message declaration as shown in the following example. In the example, the method `~createUniqueId()` is used to generate a unique id for the conversation, whenever an instance of the message is created. The plan can send the message using `~dispatchMessageAndWait()` and directly receive the corresponding reply message. When using a timeout in `~dispatchMessageAndWait()` and the message is not received before the timeout has elapsed, a `~jadex.runtime.TimeoutException~` is thrown (see also [Plans>08 Plans]). For a reply message (e.g. the inform below) no special settings have to be defined in the ADF.

```
{code:xml}
<events>
  <messageevent name="request" type="fipa" direction="send">
    <parameter name="performative" class="String">
      <value>SFipa.REQUEST</value>
    </parameter>
    <parameter name="conversation_id" class="String">
      <value>SFipa.createUniqueId($scope.getAgentName())</value>
    </parameter>
  </messageevent>
```

```

    <messageevent name="inform" type="fipa" direction="receive">
      <parameter name="performative" class="String" direction="fixed">
        <value>SFipa.INFORM</value>
      </parameter>
    </messageevent>
  </events>
}code}
~Message declarations~

{code:java}
public void body()
{
  IMessageEvent me = createMessageEvent("request");
  ... Set other parameters as desired
  IMessageEvent reply = sendMessageAndWait(me);
  ... Handle reply message
}
}code}
~Example of an initial conversation message~

```

On the other hand, if you have received a message event and want to reply to the sender you don't have to create a new message event from scratch but can directly create a reply. This ensures that all important information such as the `conversation_id` or `in_reply_to` also appears in the answer. Moreover, message properties, which should not change during a conversation (e.g. protocol, language and ontology) are also automatically copied into the reply. A reply can be created by calling `~createReply(String type [, Object content])~` method directly on the received message event. This method takes the message event type for the reply as parameter. Note that the message type with which you are replying also has to be present in your ADF as shown in the following example.

Example for Replying to a Message

```

{code:xml}
<events>
  <messageevent name="request" type="fipa" direction="receive">
    <parameter name="performative" class="String" direction="fixed">
      <value>SFipa.REQUEST</value>
    </parameter>
  </messageevent>
  <messageevent name="inform" type="fipa" direction="send">
    <parameter name="performative" class="String">
      <value>SFipa.INFORM</value>
    </parameter>
  </messageevent>
</events>

```

```

{code}
~Message declarations~

{code:java}
public void body()
{
    Message receiving in the plan.
    IMessageEvent msg = (IMessageEvent)getInitialEvent();
    Object content = ... Prepare content for reply
    IMessageEvent reply = msg.createReply("inform", content);
    sendMessage(reply); Take care to send 'reply' and not 'msg'!
}
{code}
~Example code for creating and sending a reply message~

```

The way of handling conversations described in this section is pretty different to programming agents based on abstract goals, as the programmer has to directly deal with all alternatives of the interaction flow. This process can be tedious and error-prone. Therefore, in Jadex a predefined capability is available, that already implements common use cases of interactions as specified in standardized FIPA interaction protocols (e.g. request, contract-net, auctions). The protocols capability allows to focus on the goals of the agents participating in a conversation. The protocols capability is described in detail in [Predefined Capabilities>15 Predefined Capabilities]. Even if you want to implement your own custom interaction protocol, you should have a look at the protocols capability, because it introduces helpful patterns that can be applied to other interactions as well.

Chapter 10. Expressions

For many elements (parameter values, default and initial facts of beliefs, etc.) the developer has to specify expressions in the ADF. The most important part of an expression is the expression string. In addition, some meta information can be attached to expressions, e.g., to specify the class the resulting value should have.

Expression Syntax

The expression language follows a Java-like syntax. In general, all of the *operators* of the Java language are supported (with the exception of assignment operators), while no other constructs can be used. Operators are, for example, math operators (+-*/%), logical operators (&&, ||, !), and method, or constructor invocations. Other unsupported constructs are loops, class declarations, variable declarations, if-then-else blocks, etc. As a rule you can use every Java code

that can be contained in the right hand side of a variable assignment (e.g., *var* = <expression>). There are just two exceptions to this rule: Declarations of anonymous inner classes are not supported. Assignment operators (=, +=, *=...) as well as de- and increment operators (++ , -) are not allowed, because they would violate declarativeness.

In addition to the Java-like syntax, the language has some extensions: Parameters give access to specific elements depending on the context of the expression. OQL-like select statements allow to create complex queries, e.g., for querying the beliefbase. To simplify the Java statements in the expressions, imports can be declared in the ADF (see Imports) that allow to use unqualified class names. The imports are defined once, and can be used for all expressions throughout the ADF.

Expression Properties

Expressions have properties which can be specified as attributes of the enclosing XML tag. The optional class attribute can be specified for any expression, and is used for cross checking the expression string against the expected return type. This allows to detect errors in the ADF already at load time, which could otherwise only be reported at runtime.

The evaluation mode influences when and how often the expression is evaluated at runtime. A “static” expression caches the value once the expression created, while the value of a “dynamic” expression is reevaluated for every access. The default values of these properties depend on the context in which the expression is used. E.g. initial facts of beliefs are usually static, while conditions are dynamic.

Reserved Variables

Within expressions, several variables can be accessed depending on the context the expression is used in. Generally, the following variable names are reserved for agent components and can be accessed directly by their name. In the following table the reserved variables, their type and accessibility settings are summarized. Values of beliefs and belief sets (from the \$beliefbase) and parameter(set)s (from \$plan, \$event, \$goal, and \$ref) can be accessed using a shortcut notation (allowing to write statements like “\$beliefbase.mybelief”, which is the same as “\$beliefbase.getBelief(“mybelief“).getFact()).

Name	Class	Accessibility
\$scope	ICapability	In any expression

Name	Class	Accessibiliy
\$beliefbase	IBeliefbase	In any expression
\$planbase	IPlanbase	In any expression
\$goalbase	IGoalbase	In any expression
\$eventbase	IEventbase	In any expression
\$expressionbase	IExpressionbase	In any expression
\$propertybase	IPropertybase	In any expression
\$goal	IGoal	In any goal expression (except creation condition and binding options)
\$plan	IPlan	In any plan expression (except trigger and pre condition and binding options)
\$event	IEvent	In any event expression (except binding otpions)
\$ref	IGoal	In any inhibition arc expression.
\$messagemap	Map	In match expressions of message events.
		<p/> <i>Reserved expression variables</i>

Expressions Examples

In the following, two example expressions are shown. Here the expressions are used to specify the facts of some beliefs. In fact there are many places besides beliefs in the ADF where expressions can be used. In the first case, the “starttime” fact expression is evaluated only once when the agent is born. The second belief represents the agent’s lifetime and is recalculated on every access.

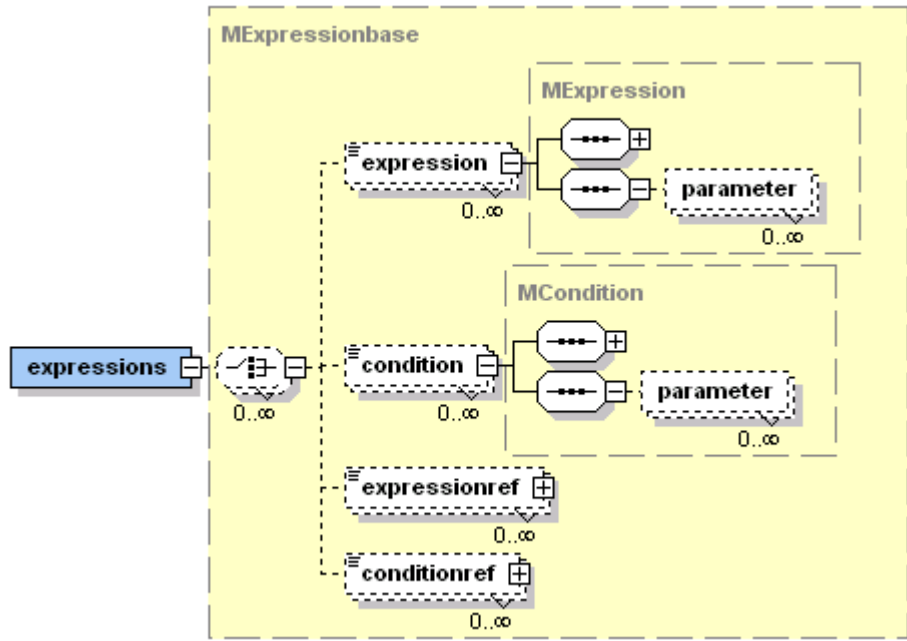
```
<belief name="starttime" class="long">
  <fact>
    System.currentTimeMillis()
  </fact>
</belief>

<belief name="lifetime" class="long" evaluationmode="pull">
  <fact>
    System.currentTimeMillis() - $beliefbase.starttime
  </fact>
</belief>
```

Example expressions

ADF Expressions

The expression language cannot only be used to specify values for beliefs, plans, etc. in the ADF but also for dynamic evaluation, e.g., to perform queries on the state of the agent, most notably the current beliefs. Expressions (*jadex.bdi.runtime.IExpression*) can be created at runtime by providing an expression string. A better way is to predefine expressions in the ADF in the expression base (see Figure below). Because predefined expressions only have to be parsed and precompiled once and can be reused by different plans, they are more efficient. The following example shows a predefined expression for searching the beliefbase for a certain person contained in the belief persons, using the OQL-like language extension described in more detail below. Moreover, this example uses a custom parameter \$surname to specify which person to retrieve from the belief set.



The Jadex expressions XML schema part

Primary usage of predefined expression is to perform queries, when executing plans. The *getExpression(String name)* method creates an expression object based on the predefined expression with the given name. In addition, the *createExpression(String exp [, String[] paramnames, Class[] paramtypes])* method is used to create an expression directly by parsing the expression string (without referring to a predefined expression). Custom parameters can be optionally be defined for such queries by additionally providing the parameter names and classes. Values for these parameters have to be supplied when executing the query. The expression object provides several *execute()* methods to evaluate a

query specifying either no parameters, a single parameter as name/value pair, or a set of parameters that are defined as a *String* and an *Object array* containing parameter names and values separately. <!-- You can also pre-set parameters before executing the query using the *setParameter()* method. For example, one can execute the person query with a given surname. ->

```

<agent ...>
  ...
  <expressions>
    <expression name="find_person" class="Person">
      select one Person $person
      from $person in $beliefbase.persons
      where $person.getSurname().equals($surname)
    </expression>
  ...
</expressions>
  ...
</agent>

public void body
{
  IExpression query = getExpression("find_person");
  ...
  Person person = (Person)query.execute("$surname", "Miller");
  ...
}

```

Evaluating an expression from a plan

OQL-like Select Statements

Jadex provides an OQL-like query syntax, which can be used in conjunction with any other expression statements. OQL (Object-Query-Language) is an extension of SQL (Structured-Query-Language) for object-oriented databases. The generic query syntax as supported by Jadex is very similar to OQL (note that until now only select statements are supported). The syntax is shown in next code snippet.

```

[select (one)? <class>? <result-expression>
from (<class>? <element> in)? <collection-expression>
  (, <class>? <element> in <collection-expression>)*
(where <where-expression>)?

```

(order by <ordering-expression> (asc | desc)?)?

Syntax of OQL-like select statements

<p/>

The <collection-expression> has to evaluate to an object that can be iterated (an array or an object implementing *Iterator*, *Enumeration*, *Collection*, or *Map*). In the other expressions (result, where, ordering) the query variables can be accessed using <element>. When using “<element>” as result expression, the second “<element> in” part can be omitted for readability. While you are free to use any expression for the result and the ordering, the where clause, of course, has to evaluate to a boolean value.

Some simple example queries (assuming that the beliefbase contains a belief set “persons”, where each person has attributes “forename”, “surname”, “age”, and “address”) are shown in the code snippets below. The first query returns a *java.util.List* of all persons in the order they are contained in the belief set. The second query only returns persons that are older than 21. In this case a cast is used to invoke the *getAge()* method. The third example orders the returned list by the addresses of the persons, using a type declaration at the beginning of the query, and therefore does not need a cast for accessing the *getAddress()* method. The order-by implementation relies on the *java.lang.Comparable* interface. In the example, the addresses have to be comparable for the query to work. The next query shows that it is possible to use complex expressions to create the result elements. Note, that in this case, the “\$person in” part cannot be omitted. The last example shows how to do a join. The expression returns a list of strings of any two (distinct) persons, which have the same address.

```
select $person from $beliefbase.persons
```

```
select $person from $beliefbase.persons where ((Person)$person).getAge()>21
```

```
select Person $person from $beliefbase.persons order by $person.getAddress()
```

```
select $person.getSurname()+" , "+$person.getForename()  
from Person $person in $beliefbase.persons
```

```
select $p1+" , "+$p2 from Person $p1 in $beliefbase.persons,  
                             Person $p2 in $beliefbase.persons
```

```
where $p1!=$p2 & $p1.getAddress().equals($p2.getAddress())</programlisting>
```

Examples of OQL-like select statements

An extension to OQL is the support of the “one” keyword. The default (without “one”) is standard OQL semantics to return all objects matching the query. The

“one” keyword is used to select a single element. For queries without ordering, this returns the first found element that matches the query. When using ordering, the query is evaluated for all input elements and returns the first element after having applied the ordering. In both cases null is returned, when no element matches the query. Without the “one” keyword, an empty collection is returned, when no element matches the query.

Chapter 11. Conditions

In essence, a condition is a monitored boolean expression, what means that whenever some of the referenced entities (e.g., beliefs) change the expression of the condition is evaluated. Associated with a condition is an action, that gets executed whenever the condition is triggered. Context-specific conditions as defined in the ADF have special associated actions (e.g., for activating goals).

The trigger of a predefined condition depends on the context, for example the maintain condition of a maintain goal is triggered when the expression value changes to false, because the goal should be processed whenever the maintain condition is violated.

When programming plans, it is also possible to explicitly wait for certain conditions using the *waitForCondition(ICCondition cond)* method. Conditions are obtained in a similar fashion to expressions, either by instantiating a predefined condition from the ADF, or by creating a new condition from an expression string. When waiting for a condition, the plan will be blocked until the condition triggers, which by default means that its value changes to true. The condition is monitored automatically by the agent, by considering all internal state changes that may affect the condition value, e.g., when some other plan changes a belief.

The following example uses the “timer” belief from Section 6.3 to execute some action when the alarmtime has reached (belief not shown here).

```
<agent ...>
  ...
  <expressions>
    <condition name="alarmtime#95;reached">
      $beliefbase.timer >= $beliefbase.alarmtime
    </condition>
    ...
  </expressions>
  ...
</agent>

public void body
```

```

{
    ICondition condition = getCondition("alarmtime&#95;reached");
    ...
    IEvent event = waitForCondition(condition);
    ...
}

```

Chapter 12. Properties

This chapter contains an overview about the usage of agent and capability properties, that allow to change the behavior of the agent. In general, properties represent static expressions, i.e. they are interpreted but only once when an agent instance is loaded. Properties can be defined using the properties section of the agent (and capability) XML file and add an arbitrary number of properties.

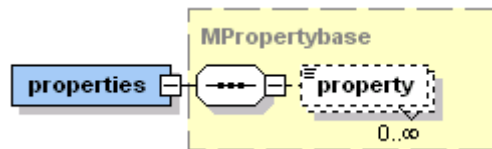


Figure 62:

Propertybase xml specification

The table below gives an overview of all available properties. The scope denotes, if the property can only be specified for the agent as a whole, or can be adjusted to different values for individual capabilities.

Scope	Property	Default	Possible Values
agent	max_planstep_time	unlimited	Positive long or 0 for unlimited
agent	storedmessages.size	unlimited	Positive int or 0 for unlimited
agent	debugging	false	{true, false}
capability	logging.level	SEVERE	java.util.logging.Level instances
capability	logging.useParentHandlers	true	{true, false}
capability	logging.addConsoleHandler	null	java.util.logging.Level instances
capability	logging.level.exceptions	SEVERE	java.util.logging.Level instances

The Jadex system has to take care that only one plan step is executed at a time, therefore it waits until a plan step returns. With the help of the “max_planstep_time” property it is possible to set the maximum execution time for a single planstep in milliseconds. Per default the execution time is not limited

and a plan might execute as long plan steps as it want to (note that long plan steps are not recommended, because they hinder the agent in responding to urgent events). A plan running longer than the maximum plan step time will be forcefully aborted by the system. This feature is only available for standard plans.

The “storedmessages.size” property can be used to restrict the number of monitored conversations. Generally, an agent has to keep track of its sent messages for being able to associate an incoming message to already sent messages. This means an agent has to know what it sent to determine if it received some reply of a previous message. When restricting the number of conversations, and a message arrives belonging to an ongoing conversation that was removed from the cache, the agent might not be able to route the message to the correct capability.

The “debugging” property influences the execution mode of the agent. When setting debugging to *true* the agent is halted after startup and set to single-step mode. You can then use the debugger tab of the introspector tool execute the agent step-by-step and observe its behavior.

The logging properties can be used to adjust the logging behavior according to the Java Logging API . The level influences the amount of logging information produced by the agent (logging information below the level will be completely ignored). Setting “useParentHandlers” to “true” will forward logging information to the parent handler, which by Java default causes logging output up to the INFO level to be displayed on the console. If you want to direct more detailed logging output to the console use the “addConsoleHandler” property, which creates a custom logging handler for console output with the specified logging level. <!-- More about logging settings can be found in <xref linkend=“JadexToolGuide”/>.->

The “logging.level.exceptions” property can be used to specify the logging level for uncaught exceptions occurring in plan bodies. Using the default settings for logging (non-BDI specific) exceptions are printed out as SEVERE log messages to the console. You can adjust the level settings to suppress exception log messages from plans that you expect to throw exceptions.

The following concrete subclasses of the abstract *jadex.bdi.runtime.BDIFailureException* may occur:

- **jadex.bdi.runtime.GoalFailureException** A goal failure exception indicates that a goal could not successfully pursued. It is thrown by the reasoning engine when e.g. `dispatchSubgoalAndWait()` is called and the goal does not succeed.
- **jadex.runtime.PlanFailureException** Can be thrown from user code in plans for indicating that a normal plan failure has occurred. Also calling the `fail()` method will lead to throwing a plan failure exception.
- **jadex.bdi.runtime.TimeoutException** Occurs, when any `waitFor...()` method is called with exception of the basic `waitFor(time)` method, which will only block until the given time interval elapsed.

The following figure shows an example property section setting logging and plan step options.

```
<properties>
  <property name="logging.level">Level.WARNING</property>
  <property name="scheduler.max_planstep_time">5000</property>
</properties>
```

Example properties section

1 Chapter 13. Configurations

Configurations represent both the initial and/or end states of an agent type. Initial instance elements can be declared that are created when the agent (resp. the capability) is started. This means that initial elements such as goals or plans are created immediately when an agent is born. On the contrary, end elements can be used to declare instance elements such as goals or plans that will be created when an agent is going to be terminated. After an agent has been urged to terminate (e.g. by calling `~killAgent()` from within a plan or by an CMS `~cms_destroy_component~` goal), all normal goals and plans will be aborted (except plans that perform their cleanup code, i.e. execute one of the `~passed()`, `~failed()` or `~aborted()` methods) and the declared end elements will be created and executed.

Instance and end elements always have to refer to some original element via the “ref” attribute. ~~<!-Additionally, an optional instance name can be provided via the “name” attribute. This can be useful if the element should be accessible later on via this name.->~~ Besides the reference also bindings can be used in combination with initial/end elements. If (at least one) binding parameter is declared instance elements will be created for all possible bindings.

It is possible to declare any number of configurations for a single agent or capability. When starting an agent or including a capability you can choose among the available configurations. In the XML portion for specifying configurations is depicted. Each configuration must have a name for identification purposes. The default configuration can be set up by using the `~default~` attribute of the `\<configurations\>` base tag. If no explicit default configuration is specified, the first one declared in the ADF is used.

~The Jadex configurations XML schema part~

A configuration allows to specify various properties. Generally, the configuration allows two different kinds of adaptations. The first one is the creation of instance elements for declared types, e.g., initial resp. end goals or plans. The second one is the configuration of instance elements such as beliefs or capabilities at start time. In the following, the possible settings will be discussed.

1.1 Capabilities

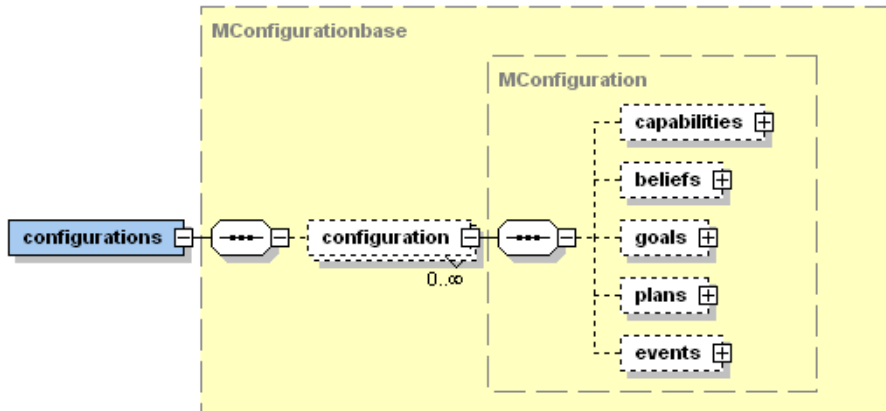


Figure 63:

The `<capabilities>` tag allows to configure included capabilities. For this purpose a reference to an included `<initialcapability>` must be declared. The reference to the capability is established by setting the `~ref~` attribute to the symbolic name of the capability specified within the `<capabilities>` section of the agent/capability (i.e., not the type name but the instance name). The configuration to be used by the included capability can be set by using the `~configuration~` attribute of the initial capability tag.

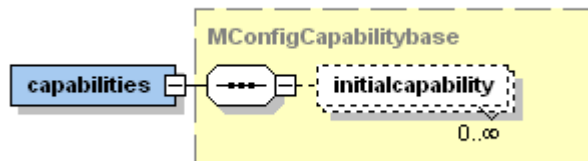


Figure 64:

~The Jadex initial capabilities XML schema part~

In the figure below an example is shown how the initial state can be used to declare two different initial states. In state “one” the included capability “mycap” is configured to use its initial state “a”, while in state “two” “b” is used. Per default the agent would start using initial state “two” as it is declared as default.

{code:xml}

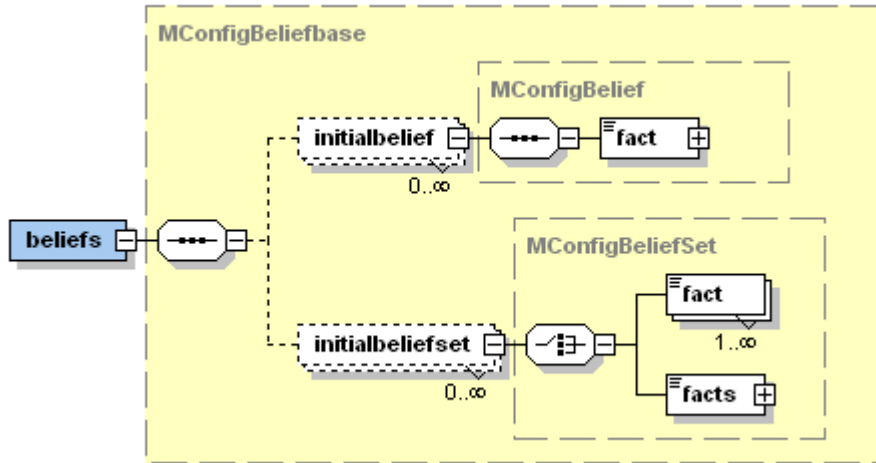

```

<agent ...>
...
<capabilities>
  <capability name="mycap" file="SomeCapability"/>
</capabilities>
...
<configurations default="two">
  <configuration name="one">
    <capabilities>
      <initialcapability ref="mycap" configuration="a"/>
    </capabilities>
  </configuration>
  <configuration name="two">
    <capabilities>
      <initialcapability ref="mycap" configuration="b"/>
    </capabilities>
  </configuration>
</configurations>
</agent>
{code}
~Initial capability configuration~

```

1.1 Beliefs

In the `<beliefs>` section the initial facts of beliefs and belief sets can be altered or newly introduced. In order to set the initial fact(s) of a belief or belief set an `<initialbelief>` resp. an `<initialbeliefset>` tag should be used. The connection to the “real” belief is again established via the `~ref~` attribute and the facts can be declared in the same way as default values of beliefs and belief sets. The initial state does not distinguish between original beliefs and references to beliefs from other capabilities, therefore the same tags can also be used to change initial facts of belief references and belief set references as well.



~The Jadex initial beliefs XML schema part~

The example below shows how a configuration can be used to change belief facts. Belief “name” has a default value of “Jim” which is overridden by the initial belief fact “John”. The belief set “names” has no default values. In the initial state it is filled with some data from a database. This means that for all results that the method `~DB.queryNames()~` produces, a new fact is added to the belief set.

```
{code:xml}
<agent ... >
  ...
  <beliefs>
    <belief name="name" class="String">
      <fact>"Jim"</fact>
    </belief>
    <beliefset name="names" class="String"/>
  </beliefs>
  ...
  <configurations>
    <configuration name="one">
      <beliefs>
        <initialbelief ref="name">
          <fact>"John"</fact>
        </initialbelief>
        <initialbelief set ref="names">
          <facts>DB.queryNames()</facts>
        </initialbelief set>
      </beliefs>
    </configuration>
  </configurations>
</agent >
```

```

    </configuration>
  </configurations>
</agent>
{code}
~Initial belief configuration~

```

1.1 Goals

In the `<goals>` section initial and end goals can be specified. Initial goals will be instantiated when an agent is born whereas end goals are created when an agent is beginning the termination phase. This means that a new goal instance is created for each declared initial resp. end goal at the mentioned points in time. The specification of an `<initialgoal>` and an `<endgoal>` requires the connection to the underlying goal template which is used for instantiation. For this purpose the `~ref~` attribute is used. Optionally, further parameter(set) values can be declared by using the corresponding `<parameter>` and `<parameterset>` tags.

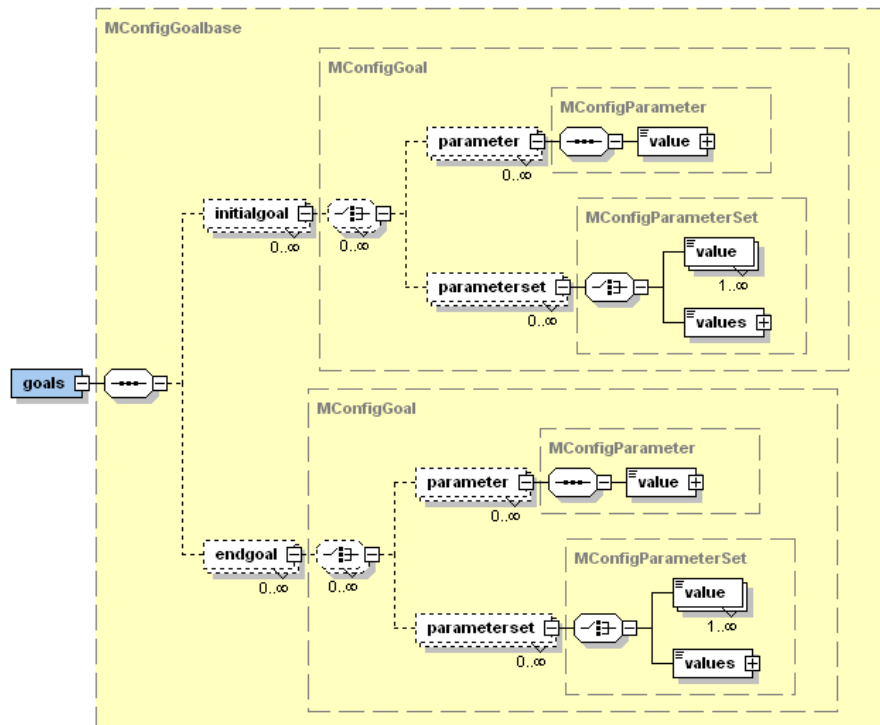


Figure 65:

~The Jadex initial and end goals XML schema part~

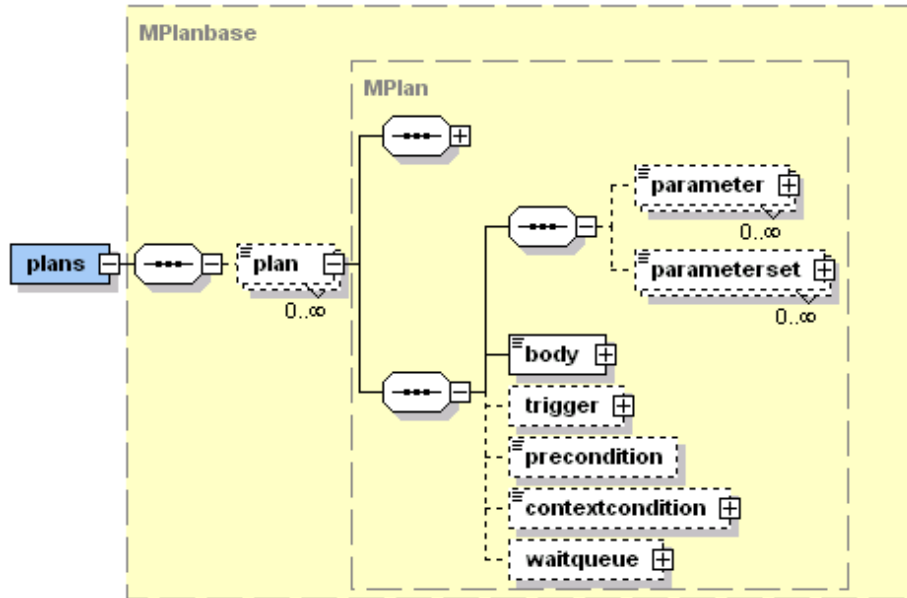
In the example below is depicted how an initial and end goal can be created. Both, the initial and end goal refer to the declared “play_song” perform goal of the agent and provides a new parameter value for the song parameter. When the agent is started in this initial state it creates the initial goal and pursues it. So, given that the agent has some plan to play an mp3 file, it will play a welcome song in this example. On the other hand the agent will also play a good bye jingle when it is terminated by creating the corresponding end goal.

```
{code:xml}
<agent ...>
...
<goals>
  <performgoal name="play_song">
    <parameter name="song" class="URL"/>
  </performgoal>
</goals>
...
<configurations>
  <configuration name="one">
    <goals>
      <initialgoal name="welcome" ref="play_song">
        <parameter ref="song">
          <value>new URL("http://someserver/welcome.mp3](http://someserver/welcome.mp3)")</value>
        </parameter>
      </initialgoal>
      <endgoal name="goodbye" ref="play_song">
        <parameter ref="song">
          <value>new URL("http://someserver/goodbye.mp3](http://someserver/goodbye.mp3)")</value>
        </parameter>
      </endgoal>
    </goals>
  </configuration>
</configurations>
</agent>
{code}
~Initial and end goals~
```

1.1 Plans

In the `<plans>` section initial and end plans can be specified. This means that a new plan instance is created for each declared initial and end plan. The specification of an `<initialplan>` and `<endplan>` requires the connection to the underlying plan template which is used for instantiation. For this purpose the `~ref~` attribute is used. Optionally, further parameter(set) values can be declared by using the corresponding `<parameter>` and `<parameterset>`

tags.



~The Jadex initial and end plans XML schema part~

In the example is depicted how an initial and end plan can be used. In this case an initial “print_hello” plan is declared which refers to the “print_hello” plan template of the agent. As result the agent will print “Hello World!” to the console on start-up. On the contrary it will also print “Goodbye World” when the agent gets terminated by creating the corresponding end plan.

```
{code:xml}
<agent ... >
...
<plans>
  <plan name="print_plan">
    <parameter name="text" class="String"/>
    <body class="PrintOnConsolePlan" />
  </plan>
</plans>
...
<configurations>
  <configuration name="one">
    <plans>
      <initialplan ref="print_hello">
        <parameter name="text">"Hello World!"</parameter>
      </initialplan>
      <endplan ref="print_goodbye">
        <paramter name="text">"Goodbye World!"</parameter>
      </endplan>
    </plans>
  </configuration>
</configurations>
```

```

        </endplan>
    </plans>
</configuration>
</configurations>
</agent>
{code}
~Initial and end plans~

```

1.1 Events

Finally, in the `<events>` section initial and end events can be specified. This means that a new event instance is created for each declared initial event after startup of the agent. Additionally, new event instances are created for all declared end events whenever the agent is shutdown. It is possible to define initial/end internal and initial/end message events (goal events are not necessary as initial goals can be declared). The specification of an `<initialinternalevent>` resp. an `<endinternalevent>` or an `<initialmessageevent>` resp. an `<endmessageevent>` requires the connection to the underlying event template which is used for instantiation. For this purpose the `~ref~` attribute is used. Optionally, further `parameter(set)` values can be declared by using the `<parameter>` and `<parameterset>` tags.

~The Jadex initial and end events XML schema part~

In the example below it is shown how an initial and end message event can be created. The initial/end message events refer to the underlying message event template `inform_state` and set the parameter values for the content as well as for the receiver accordingly. When an agent named `Harry` is started, it sends an initial message event with the content `Harry is born` to an agent named `Uncle` on the same platform. Likewise it sends the message `Harry is terminating` to `Uncle` when the agent shuts down.

```

{code:xml}
<events>
  <messageevent name="inform_state" type="fipa" direction="send">
    <parameter name="performative" class="String" direction="fixed">
      <value>SFipa.INFORM</value>
    </parameter>
  </messageevent>
</events>
...
<configurations>
  <configuration name="one">
    <events>
      <initialmessageevent ref="inform_state">
        <parameter ref="content">

```

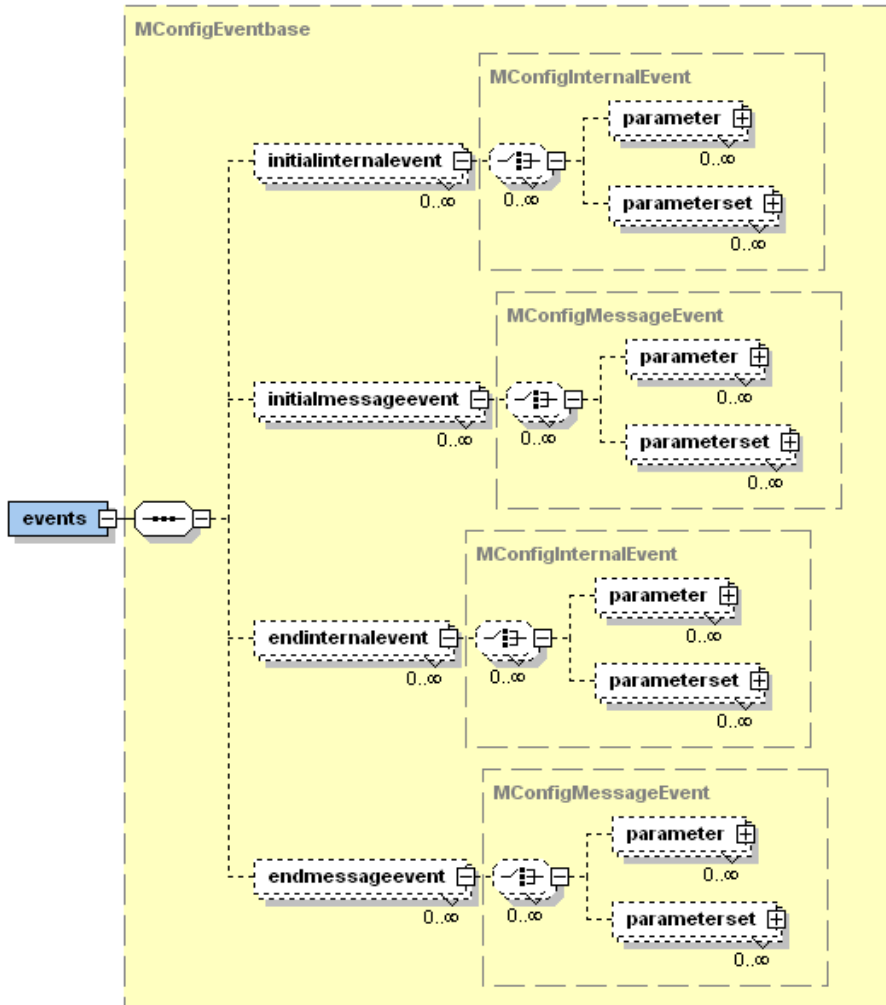


Figure 66:

```

        <value>${scope.getAgentName()}+" is born."</value>
    </parameter>
    <parameterset ref="receivers">
        <value>${scope.getEventbase().createComponentIdentifier("Uncle")}</value>
    </parameterset>
</initialmessageevent>
<endmessageevent ref="inform_state">
    <parameter ref="content">
        <value>${scope.getAgentName()}+" is terminating."</value>
    </parameter>
    <parameterset ref="receivers">
        <value>${scope.getEventbase().createComponentIdentifier("Uncle")}</value>
    </parameterset>
</endmessageevent>
</events>
</configuration>
</configurations>
{code}
~Initial events~

```

1 Chapter 14. External Interactions

In this chapter it is explained how the interaction of Jadex agents with other system components that are not necessarily agents can be done. For this purpose it is shown how agent internals can be accessed from other (non-agent) threads and additionally how agent listeners can be employed to get notified whenever changes within the agent happen (cf. the following two sections respectively).

1.1 External Processes

A Jadex agent is synchronized in the sense, that only one plan step at a time is executed (or none, if the agent is busy performing internal reasoning processes). Sometimes one may want to access agent internals from external threads. A good example is when your agent provides a graphical user interface (GUI) to accept user input. When the user clicks a button your Java AWT/Swing event handler method is called, which is executed on the Java AWT-Thread (there is one AWT Thread for each Java virtual machine instance). To force that such external threads are properly synchronized with the internal agent execution, every invocation on a BDI API component (the flyweights implementing e.g. IBelief, IGoal, etc.) is checked. If the agent calls these methods from its own thread the code is directly executed. Otherwise, if an external thread is the caller, the call is redirected to the agent thread and the external thread has to wait until the agent call returns. Please note that this call scheme can lead to deadlocks when agents try to invoke methods on other agents forming a cycle in the callgraph. To gain access to agents and call methods on them from an external thread the external access interface should be used. It can be fetched from

the component management service (currently only for local agents) via the `~getExternalAccess(IComponentIdentifier cid, IResultListener listener)~` method. An alternative from within a plan is to use the method `~getExternalAccess()~` provided by the `~jadex.bdi.runtime.AbstractPlan~` class.

The external access object for BDI agents implements the `~IBDIExternalAccess~` and `~ICapability~` interfaces. Together, they provide access to all important features of a capability (beliefbase, goalbase, etc.) as well as plan methods for directly creating and dispatching goals, message and internal events. The following code presents an example where a belief is changed when the user presses a button.

```
{code:java}
public void body()
{
    ...
    JButton button = new JButton("Click Me");
    button.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            This code is executed on the AWT thread (not on the plan thread!)
            IBeliefbase bb = getExternalAccess().getBeliefbase();
            bb.getBelief("button_pressed").setFact(new Boolean(true));
        }
    });
    ...
}
{code}
~External process example~
```

1.1 Agent Listeners

Agent listeners can be used to get informed whenever agent state changes happen. Normally, listeners will be employed in agent external components such as a GUI for getting information about declared elements. A GUI e.g. could use a listener to update its view with respect to belief changes in the agent. Generally, for all important agent attitudes such as belief, plans and goals as well as the agent itself different listener types exist (cf. the listener table below).

Depending on the listener type different callback methods are provided that are automatically invoked when relevant changes happen. Whenever a callback method is invoked a so called `~AgentEvent~` is passed and contains relevant information about the change that happened. It basically offers the two methods

`~getSource()~` and `~getValue()~`. The source here is the originating element of the change event. For belief and beliefset changes, the agent event additionally contains the changed fact object, accessible by `~getValue()~`.

The invocation of listener methods can happen either on the agent thread or on a separate thread. The notification is performed on the agent thread so that it is not directly possible to use blocking calls such as `~dispatchTopLevelGoalAndWait()~` in the listener implementation code. If you want to use these methods you have to redirect the call to an agent external thread.

The addition and removal of listeners can be done either on the instance elements themselves (e.g. a goal) or on the bases (e.g. the goalbase). In case the listener shall be added on an instance element it is only necessary to pass the listener object itself as parameter of the call (e.g. `~addBeliefListener(IBeliefListener listener)~`). In case a type-based listener shall be used e.g. for getting informed about new goal instances in addition to the parameters aforementioned also the type needs to be declared (e.g. `~addGoalListener(String type, IGoalListener listener)~`).

In the listener example below it is shown how a belief listener can be directly added to a “name” belief via the external access interface. It is used to update the value of a textfield whenever the belief value changes.

```
{code:java}
IExternalAccess agent = ...
agent.getBeliefbase().getBelief("name").addBeliefListener(new IBeliefListener()
{
    public void beliefChanged(AgentEvent ae)
    {
        textfield.setText("Name: ["+ae.getValue()+"]");
    }
});
{code}
```

~Agent listener example~

```
{table}
Listener| Element| Listener Methods
IAgentListener| ICapability| agentTerminating() agentTerminated()
IBeliefListener| IBelief| beliefChanged()
IBeliefSetListener| IBeliefSet| factAdded() factRemoved() factChanged()
IConditionListener| ICondition| conditionTriggered()
IGoalListener| IGoalbase IGoal| goalAdded(), goalFinished()
IInternalEventListener| IEventbase| internalEventOccurred()
IMessageEventListener| IMessageEvent IEventbase| messageEventReceived()
messageEventSent()
IPlanListener| IPlan IPlanbase| planAdded() planFinished()
```

{table}
~Available listeners~

1 Chapter 16. Using Predefined Capabilities

The documentation of the predefined capabilities is not yet finished. Please also take a look at the [BDI Tutorial>BDI Tutorial.07 Using Events] (Exercise F4) and at the [legacy documentation of Jadex 0.96>http://jadex.informatik.uni-hamburg.de/docs/jadex-0.96x/userguide/predef_cap.html] (http://jadex.informatik.uni-hamburg.de/docs/jadex-0.96x/userguide/predef_cap.html)].

Jadex uses capabilities for the modularization of agents (see [Chapter 5. Capabilities>05 Capabilities]), whereby capabilities contain ready to use functionalities. The Jadex distribution contains several ready-to-use predefined capabilities for different purposes. Besides the basic management capabilities for using the CMS (component management service, see [CMSCapability>#HTheComponentManagementService28CMS29Capability]) and the DF (see [DFCapability>#HTheDirectoryFacilitator28DF29Capability]) also a [Protocols Capability>#HTheProtocolsCapability] is available for the efficient usage of some predefined FIPA interaction protocols. The interface of a capability mainly consists of a set of exported goals which is similar to an object-oriented method-based interface description. This chapter aims at depicting their usage by offering the application programmer an overview and explanation of their functionalities and additionally a selection of short code snippets that can directly be used in your applications.

The test capability for writing agent-based unit test is explained in the ~Jadex Tool Guide~, which also illustrates the usage of the corresponding Test Center user interface.

<! -CMS->

1.1 The Component Management Service (CMS) Capability

The Component Management Service (CMS) capability provides goals, that allow the application programmer to use functionalities of the local or some remote CMS. Basically the CMS is responsible for managing the component lifecycle and for interacting with the platform. Concretely this means the CMS capability can be used for:

- [Creating Components>#HCreatingComponents]

- [Destroying Components>#HDestroying Components]
- [Suspending Components>#HSuspendingComponents]
- [Resuming Components>#HResumingComponents]
- [Searching Components>#HSearchingComponents]
- [Shutting Down the Platform>#HShuttingDownthePlatform]

<! -CMS:CREATE_COMPONENTS ->

1.1.1 Creating Components

The goal `~cms_create_component~` creates a new component via the CMS on the platform.

This goal has the following parameters:

Name	Type	Description
<code>type</code>	String	The component type (name/path of component model).
<code>name*</code>	String	The name of the instance to create. If no name is specified, a name will be generated automatically.
<code>configuration*</code>	String	The initial component configuration to use. If no configuration is specified, the default configuration will be used.
<code>arguments*</code>	Map	The arguments as name-value pairs for the new component. Depending on the platform, Java objects (for Jadex Standalone or local JADE requests) or string expressions (for remote JADE requests) have to be supplied for the argument values.
<code>cms*</code>	IComponentIdentifier	The component identifier of the CMS (only required for remote requests)
<code>start*</code>	boolean	True, when the component should be directly started after creation (default). Note that some platforms will not support decoupling of component creation and starting (e.g. for remote requests in JADE).
<code>componentidentifier \[out\]</code>	IComponentIdentifier	Output parameter containing the component identifier of the created component.

~Parameters for `cms_create_component` goal (* denotes optional parameters)~

To use the “`cms_create_component`”-goal, you must first of all include the CMS-capability in your ADF (if not yet done in order to use other goals of the CMS-capability) and set a reference to the goal as described below. The name of the goal reference can be arbitrarily chosen, but it will be assumed here for convenience that the same as the original name will be used.

```
{code:xml}
...
<capabilities>
  <capability name="cmscap" file="jadex.bdi.planlib.cms.CMS" />
```

```

...
</capabilities>
...
<goals>
  <achievegoalref name="cms_create_component">
    <concrete ref="cmscap.cms_create_component" />
  </achievegoalref>
...
</goals>
...
{code}
~Including the CMS capability and the cms_create_component-goal~

```

Now you can use this goal to create a component in your plan:

```

{code:java}
public void body()
{
  ...
  IGoal cc = createGoal("cms_create_component");
  cc.getParameter("type").setValue("mypackage.MyComponent");
  dispatchSubgoalAndWait(cc);
  IComponentIdentifier createdcomponent =
    (IComponentIdentifier)cc.getParameter("componentidentifier").getValue();
  ...
}
{code}
~Creating a component on the local platform~

```

In the above listing - in order to create a component - you instantiate a new goal using the `createGoal()` method with the parameter "cms_create_component". Then you set its parameters to the desired values, dispatch the subgoal and wait. After the goal has succeeded, you can fetch the `IComponentIdentifier` of the created component by calling the `getValue()` method on the parameter "componentidentifier".

The same goal is used for remote creation of a component:

```

{code:java}
public void body()
{
  IComponentManagementService cms = IComponentManagementService.getScope()
    .getServiceContainer().getService(IComponentManagementService.class);
  IComponentIdentifier remote_cms_id = cms.createComponentIdentifier("cms@remoteplatform",
    false, new String[] {"nio-mtp://134.100.11.232:5678 "(nio-mtp://134.100.11/232:5678)"});

  IGoal cc = createGoal("cms_create_component");
  cc.getParameter("type").setValue("mypackage.MyComponent");

```

```

cc.getParameter("cms").setValue(remote_cms_id);
dispatchSubgoalAndWait(cc);
IComponentIdentifier createdcomponent =
    (IComponentIdentifier)cc.getParameter("componentidentifier").getValue();
...
}
{code}
~Creating a component on a remote platform~

```

In the above listing you can see how to create a component on a remote platform using its remote CMS. In order to do so, it's of course crucial that you know at least one address of the remote CMS.

Moreover, the corresponding transport must be available on the local platform. The transport used by the other platform can be recognized by the prefix of the address (ending with the :). *In this case the prefix is `~nio-mtp://~`](`nio-mtp://~`) , which represents the transport `~jadex.adapter.standalone.transport.niotcpmtp.NIOTCPTransport~`.*

If you know the address of the remote CMS and you're sure that the local platform supports its transport, you must create an `~IComponentIdentifier~` using the local component management service and set its name and address to that of the CMS that should create the new component.

Thereafter you can instantiate a new goal using the `~createGoal()~` method with the parameter `"cms_create_component"`. Then you set its parameters to the desired values, dispatch the subgoal and wait. After the goal has succeeded, you can fetch the `~IComponentIdentifier~` of the created component by calling the `~getValue()~` method on the parameter `"componentidentifier"`.

<! -CMS: DESTROY COMPONENTS->

1.1.1 Destroying Components

The CMS capability offers the goal `~cms_destroy_component~` to give the application programmer the possibility to destroy components, both on a local as well as on remote platforms.

The goal has the following parameters:

Name	Type	Description
componentidentifier	IComponentIdentifier	Identifier of the component to be destroyed.
cms*	IComponentIdentifier	The component identifier of the CMS (only required for remote requests)

{table}

~Parameters for cms_destroy_component (* denotes optional parameters)~

To use the ~cms_destroy_component~-goal, you must first of all include the CMS-capability in your ADF (if not yet done in order to use other goals of the CMS-capability) and set a reference to the goal as described below:

{code:xml}

```
...
<capabilities>
  <capability name="cmscap" file="jadex.bdi.planlib.cms.CMS" />
  ...
</capabilities>
...
<goals>
  <achievegoalref name="cms_destroy_component">
    <concrete ref="cmscap.cms_destroy_component" />
  </achievegoalref>
  ...
</goals>
```

{code}

~Including the CMS capability and the cms_destroy_component-goal~

Thus you can destroy a component in your plan:

{code:java}

```
public void body()
{
  IGoal dc = createGoal("cms_destroy_component");
  dc.getParameter("componentidentifier").setValue(createdcomponent);
  dc.getParameter("cms").setValue(cms); Set cms in case of remote platform
  dispatchSubgoalAndWait(dc);
  ...
}
```

{code}

~Destroying a component on a local/remote platform~

In the listing above - in order to destroy a component - you instantiate a new goal using the ~createGoal()~-method with the parameter "cms_destroy_component". Then you set its componentidentifier-parameter to the desired value, dispatch the subgoal and wait for success. The same goal is used to destroy a remote component. In this case you only have to additionally supply the remote CMS component identifier.

<! -CMS: SUSPENDING COMPONENTS *->

1.1.1 Suspending Components

The CMS offers the goals “`cms_suspend_component`” and “`cms_resume_component`” in order to suspend the execution of a component and later resume it. When a component gets suspended the platform will not process any actions of this component. Nevertheless, the component is able to receive messages from other components and will process them when its execution is resumed.

The “`cms_suspend_component`”-goal has the following parameters:

Name	Type	Description
<code>componentidentifier</code>	<code>IComponentIdentifier</code>	Identifier of the component to be suspended.
<code>cms*</code>	<code>IComponentIdentifier</code>	The component identifier of the CMS (only required for remote requests)
<code>componentdescription \[out\]</code>	<code>ICMSComponentDescription</code>	This output parameter contains the possibly changed <code>CMSComponentDescription</code> of the suspended component.

{table}

~Parameters for `cms_suspend_component` (* denotes optional parameters)~

To use the “`cms_suspend_component`”-goal, you must first of all include the CMS-capability in your ADF (if not yet done in order to use other goals of the CMS-capability) and set a reference to the goal as described below:

```
{code:xml}
...
<capabilities>
  <capability name="cmscap" file="jadex.bdi.planlib.cms.CMS" />
  ...
</capabilities>
...
<goals>
  <achievegoalref name="cms_suspend_component">
    <concrete ref="cmscap.cms_suspend_component" />
  </achievegoalref>
  ...
</goals>
...
{code}
~Including the CMS capability and the cms_suspend_component-goal~
```


Thus you can suspend a component in your plan:

```
{code:java}
public void body()
{
    IComponentIdentifier component;  The component to suspend
    ...
    IGoal sc = createGoal("cms_suspend_component");
    sc.getParameter("componentidentifier").setValue(component);
    sc.getParameter("cms").setValue(cms);  Set cms in case of remote platform
    dispatch.SubgoalAndWait(sc);
    ...
}
{code}
~Suspending a component on a local/remote platform~
```

In the listing above - in order to suspend a component - you instantiate a new goal using the ~createGoal()~method with the paramter "cms_suspend_component". Then you set its componentidentifier-parameter to the desired value, dispatch the subgoal and wait for success. As result the goal returns a possibly modified CMS component description of the suspended component. The same goal is used to suspend a remote component. In this case you only have to additionally supply the remote CMS component identifier.

<! -CMS: RESUMING COMPONENTS -*>

1.1.1 Resuming Components

If you want to resume a suspended component you can use the goal "cms_resume_component". It offers the following parameters:

Name	Type	Description
componentidentifier	IComponentIdentifier	Identifier of the component to be resumed.
cms*	IComponentIdentifier	The component identifier of the CMS (only required for remote requests)
componentdescription \[out\]	ICMSComponentDescription	This output parameter contains the possibly changed CMSComponentDescription of the resumed component.

{table}
~Parameters for cms_resume_component (* denotes optional parameters)~

To use the "cms_resume_component"-goal, you must first of all include the

CMS-capability in your ADF (if not yet done in order to use other goals of the CMS-capability) and set a reference to the goal as described below:

```
{code:xml}
...
<capabilities>
  <capability name="cmscap" file="jadex.bdi.planlib.cms.CMS" />
  ...
</capabilities>
...
<goals>
  <achievegoalref name="cms_resume_component">
    <concrete ref="cmscap.cms_resume_component" />
  </achievegoalref>
  ...
</goals>
...
{code}
~Including the CMS capability and the cms_resume_component-goal~
```

Thus you can resume a component in your plan:

```
{code:java}
public void body()
{
  ComponentIdentifier component; The component to resume
  ...
  IGoal rc = createGoal("cms_resume_component");
  rc.getParameter("componentidentifier").setValue(component);
  rc.getParameter("cms").setValue(cms); Set cms in case of remote platform
  dispatch.SubgoalAndWait(rc);
  ...
}
{code}
~Resuming a component on a local/remote platform~
```

In the above listing - in order to resume a component - you instantiate a new goal using the `createGoal()`-method with the parameter "cms_resume_component". Then you set its componentidentifier-parameter to the desired value, dispatch the subgoal and wait for success. As result the goal returns a possibly modified CMS component description of the resumed component. The same goal is used to resume a remote component. In this case you only have to additionally supply the remote CMS component identifier.

<! -CMS: SEARCHING COMPONENTS ->

1.1.1 Searching for Components

The goal “cms_search_components” allows you to search for components, both on the local platform and on remote platforms, thereby determining if the component is available at all and learning about its state (e.g. active or suspended).

The goal has the following parameters:

Name	Type	Description
description	ICMSComponentDescription	The template description to search for matching components.
cms*	IComponentIdentifier	The component identifier of the CMS (only required for remote requests)
constraints*	ISearchConstraints	Representation of a set of constraints to limit the search process. As a default, only one matching result is returned. You can set the max-results setting of the search constraints to -1 for unlimited number of search results. See [FIPA Agent Management Specification> http://www.fipa.org/specs/fipa00023/XC00023H.html#_Toc526742642] (http://www.fipa.org/specs/fipa00023/XC00023H.html#_Toc526742642)].
result \[set\]\[out\]	ICMSComponentDescription	This output parameter set contains the component descriptions that have been found.

{table}

~Parameters for cms_search_components (* denotes optional parameters)~

To use the “cms_search_components”-goal, you must first of all include the CMS-capability in your ADF (if not yet done in order to use other goals of the CMS-capability) and set a reference to the goal as described below.

{code:xml}

```
...
<capabilities>
  <capability name="cmscap" file="jadex.bdi.planlib.cms.CMS" />
  ...
</capabilities>
...
<goals>
  <achievegoalref name="cms_search_components">
    <concrete ref="cmscap.cms_search_components" />
  </achievegoalref>
  ...
</goals>
...
{code}
```

~Including the CMS capability and the cms_search_components-goal~

To search for components in your plan use the goal in the following manner:

```
{code:java}
public void body()
{
    CMSComponentDescription desc = new CMSComponentDescription(new ComponentIdentifier("a1", true));
    IGoal search = createGoal("cms_search_components");
    search.getParameter("description").setValue(desc);
    search.getParameter("cms").setValue(cms); Set cms in case of remote platform
    dispatchSubgoalAndWait(search);
    CMSComponentDescription[] result = (CMSComponentDescription[])
    search.getParameterSet("result").getValues();
    ...
}
{code}
~Searching a component on a local/remote platform~
```

In the listing above - in order to search for a component - you instantiate a new goal using the `createGoal()`-method with the parameter "cms_search_components". The search is constrained by an CMS component description that need to be provided. You could e.g. create an `CMSComponentDescription` with a new `ComponentIdentifier` and the boolean `true` for a local component as parameter, that is defined only by its name.

Then you set its description-parameter to that just created `CMSComponentDescription`, dispatch the subgoal and wait for success. Supplying an empty component description with component identifier of null allows to perform an unconstrained search, i.e. returning all components on the platform.

In case of a remote request you have to set the component identifier of the remote CMS well.

The documentation of the predefined capabilities is not yet finished.

Please also take a look at the [BDI Tutorial>BDI Tutorial.07 Using Events] (Exercise F4)

and at the [legacy documentation of Jadex 0.96>http://jadex.informatik.uni-hamburg.de/docs/jadex-0.96x/userguide/predef_cap.html] (http://jadex.informatik.uni-hamburg.de/docs/jadex-0.96x/userguide/predef_cap.html).

<!

The same goal is used to search for remote components:

```
{code:java}
public void body()
```

```

{
  ...
  IComponentIdentifier cms = ...
  CMSComponentDescription desc = new CMSComponentDescription(new Com-
ponentIdentifier("my_component@myplatform"));
  IGoal search = createGoal("cms_search_components");
  search.getParameter("description").setValue(desc);
  search.getParameter("cms").setValue(cms);
  dispatchSubgoalAndWait(search);
  CMSComponentDescription[] result = (CMSComponentDescription[])
  search.getParameterSet("result").getValues();
  ...
}
{code}
~Searching for a component on a remote platform~

```

In the above listing a component with the name “my_component” is sought-after.

Assuming that the remote CMS was created as per description in section [Creating Components>#HCreatingComponents], you have to create the ~CMSComponentDescription~ with a new ~ComponentIdentifier~ as parameter, that is defined only by its name. Afterwards, you must instantiate the “cms_destroy_component”-goal by using the ~createGoal()~ method with the parameter “cms_search_components”.

After dispatching the goal and waiting for success, you can fetch the result by calling ~getParameterSet("result").getValues()~ on the goal and casting to an array of CMS-component-descriptions. If no matching components were found, the resulting array will be empty.

>

Appendix D. FAQ & HOWTO

The Jadex frequently asked questions (FAQ) of Jadex are managed in the Jadex Wiki at activecomponents.org. At this wiki you can find always the current questions and answers. If you want to add a question/answer that you find missing feel free to add it here (the wiki is public meaning that everyone has the chance to add her/his stuff there).

What does “retrydelay” flag mean?

Without retrydelay goal processing works as follows:

goal -> plan 1 -> plan 2 -> plan 3 -> ...

until the goal is failed or succeeded. The retrydelay just specifies a delay in milliseconds before trying the next plan, when the previous plan has finished,

i.e.:

goal -> plan 1 -> wait -> plan 2 -> wait -> plan 3 -> ...
until goal fails or succeeds.

This is e.g. useful, when already tried plans are not excluded from the applicable plan set, leading to the same plan being tried over and over again.

How can the environment of a Jadex MAS be programmed?

If distribution is needed we used the approach of a separate environment. The environment holds the global state permits tasks, actions and processes being executed. See the environment user guide for details.

I have change the .java file, e.g. a plan. Why are my changes not reflected in the running Jadex system?

Jadex relies on the Java class loading mechanism. This means that normally Java classes are loaded only once into the VM. You need to restart the Platform for taking the changes effect.

In my agents there is always one plan for each goal. Why do I need goals anyway?

You don't need to use goals for every problem. But, in our opinion using goals in many cases simplifies the development and allows for easier extensions of an application. The difference between plans and goals is fundamental. Goals represent the "what" is desired while plans are characterized by the "how" could things be accomplished. So if you e.g. use a goal "achieve happy programmers" you did not specify how you want to pursue this goals. One option might be the increase of salary, another might be to buy new TFT monitors. Genereally, the usefulness of goals depends on the concrete problem and its complexity at hand.

If you find that you don't need goals for your application, consider using the more light-weight Jadex micro agents.

How can the agent become aware of or react to its own death?

Jadex supports not only an initialization phase but also a termination phase. Whenever an agent is terminated its execution will not be immediately stopped. Instead the agent changes its state to "terminating", aborts all running goals and plans and activates elements declared in the end state. For details please have a look at Chapter 13, Configurations.

If you want to be notified when an agent dies you can use an agent listener.

How can I parametrize an agent and set parameter values before starting?

All Jadex components support the use of arguments. For BDI agents , beliefs can be marked as arguments. The JCC gui automatically creates input fields for these arguments. Programmatically, the arguments can e.g. be directly referenced via their associated beliefs.

Is there some preferred persistence mechanism for beliefs?

In the current version Jadex does not provide a ready-to-use persistence mech-

anism for the beliefs of an agent. We have successfully used normal object-relational mapping frameworks such as Hibernate in combination with Jadex. Nonetheless, the task of persisting data cannot be fully automated and needs to be done in plans. This topic should be an issue of further research and improvement.

Can capabilities be used for group communication, i.e. are they some kind of tuple space, where one agent puts in data and other can read it?

No, this seems to be a common misunderstanding of the concept. A capability is comparable to a module. Each agent that includes a capability get a separate instance of that module. For details have a look at Chapter 5, Capabilities.

[**Bauer et al. 2001**] B.Bauer, J.Müller, and J.Odell. Agent UML: A Formalism for Specifying Multiagent Interaction. P.Ciancarini and M.Wooldridge. Proceedings of the First International Workshop on Agent-Oriented Software Engineering (AOSE 2000). Springer. Berlin, New York. 2001. pp.91-103.

[**Bellifemine et al. 2007**] F.Bellifemine, G.Caire, and D.Greenwood. Developing Multi-Agent Systems with JADE. John Wiley & Sons. New York, USA. 2007.

[**Bratman 1987**] M.Bratman. Intention, Plans, and Practical Reason. Harvard University Press. Cambridge, MA, USA. 1987.

[**Braubach et al. 2004**] L.Braubach, A.Pokahr, D.Moldt, and W.Lamersdorf. Goal Representation for BDI Agent Systems. R.Bordini, M.Dastani, J.Dix, and A.El Fallah Seghrouchni. Proceedings of the Second Workshop on Programming Multiagent Systems: Languages, frameworks, techniques, and tools (ProMAS04). Springer. Berlin, New York. 2004. pp.9-20.

[**Braubach et al. 2005a**] L.Braubach, A.Pokahr, and W.Lamersdorf. Jadex: A BDI Agent System Combining Middleware and Reasoning. R.Unland, M.Klusch, and M.Calisti. Software Agent-Based Applications, Platforms and Development Kits. Birkhäuser. 2005. pp.143-168.

[**Braubach et al. 2005b**] L.Braubach, A.Pokahr, and W.Lamersdorf. Extending the Capability Concept for Flexible BDI Agent Modularization. R.Bordini, M.Dastani, J.Dix, and A.El Fallah Seghrouchni. Proceedings of the Third International Workshop on Programming Multi-Agent Systems (ProMAS'05). . 2005. pp.99-114.

[**Busetta et al. 2000**] P.Busetta, N.Howden, R.Rönnquist, and A.Hodgson. Structuring BDI Agents in Functional Clusters. N.Jennings and Y.Lespérance. Intelligent Agents VI, Proceedings of the 6th International Workshop, Agent Theories, Architectures, and Languages (ATAL) '99. Springer. Berlin, New York. 2000. pp.277-289.

[**Hindriks et al. 1999**] K.Hindriks, F.de Boer, W.van der Hoek, and J.-J.Meyer. Agent Programming in 3APL. N.Jennings, K.Sycara, and M.Georgeff.

Autonomous Agents and Multi-Agent Systems. Kluwer Academic publishers. 1999. pp. 357-401.

[**Huber 1999**] M.Huber. JAM: A BDI-Theoretic Mobile Agent Architecture. O.Etzioni, J.Müller, and J.Bradshaw. Proceedings of the Third Annual Conference on Autonomous Agents (AGENTS-99). ACM Press. New York. 1999. pp. 236-243.

[**Lehman et al. 1996**] J. F.Lehman, J. E.Laird, and P. S.Rosenbloom. A gentle introduction to Soar, an architecture for human cognition. Invitation to Cognitive Science Vol. 4. MIT press. 1996.

[**McCarthy et al. 1979**] J.McCarthy. Ascribing mental qualities to machine. M.Ringle. Philosophical Perspectives in Artificial Intelligence. Humanities Press. Atlantic Highlands, NJ. 1979. pp. 161-195.

[**Pokahr et al. 2005a**] A.Pokahr, L.Braubach, and W.Lamersdorf. A Goal Deliberation Strategy for BDI Agent Systems. T.Eymann, F.Klügl, W.Lamersdorf, M.Klusck, and M.Huhns. In Proceedings of the third German conference on Multi-Agent System TEchnologieS (MATES-2005). Springer-Verlag. Berlin Heidelberg New York. 2005.

[**Pokahr et al. 2005b**] A.Pokahr, L.Braubach, and W.Lamersdorf. A Flexible BDI Architecture Supporting Extensibility. A.Skowron, J.P.Barthes, L.Jain, R.Sun, P.Morizet-Mahoudeaux, J.Liu, and N.Zhong. Proceedings of The 2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT-2005). IEEE Computer Society. 2005. pp. 379-385.

[**Pokahr et al. 2005c**] A.Pokahr, L.Braubach, and W.Lamersdorf. Jadex: A BDI Reasoning Engine. R.Bordini, M.Dastani, J.Dix, and A.El Fallah Seghrouchni. Programing Multi-Agent Systems. Kluwer Academic Publishers. 2005. pp.149-174.

[**Rao and Georgeff 1995**] A.Rao and M.Georgeff. BDI Agents: from theory to practice. V.Lesser. Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95). The MIT Press. Cambridge, MA, USA. 1995. pp.312-319.

[**Shoham 1993**] Y.Shoham. Agent-oriented programming. D. G.Bobrow. Artificial Intelligence Volume 60. Elsevier. Amsterdam. 1993. pp.51-92.

[**Winikoff 2005**] M.Winikoff. JACK Intelligent Agents: An Industrial Strength Platform. R.Bordini, M.Dastani, J.Dix, and A.El Fallah Seghrouchni. Programing Multi-Agent Systems. Kluwer Academic Publishers. 2005. pp.175-193.

1 Introduction

Agent applications consist not only of agents but also of an environment the agents are situated in. Hence, the construction of an agent application requires efforts in both areas, whereby the environmental aspects should not be underestimated. One problem is that many different kinds of environments exist and depending

on the type of application quite different environment requirements may exist. The environment support, called `~EnvSupport~` described in this guide is meant to support the rapid development of virtual 2d and 3d environments. This allows quickly developing e.g. simulation applications, in which agents purely act in this virtual environments. But it is also feasible to use a virtual environments as an enhancement for a real one, e.g. a pheromone-based coordination for robots. `EnvSupport` covers many aspects for a complete and seamless integration of agents with a virtual environments. The idea for building `EnvSupport` emerged from our experiences with building example applications. We noticed that a lot of similarities between our example applications exist and that it could be beneficial to have a generic infrastructure that supports the construction of virtual worlds. If you have used agent simulation toolkits such as `NetLogo` or `SeSAM`, you will know that these toolkits have built-in support for (different kinds of) 2d and 3d environments. In `Jadex` with `EnvSupport` something similar is available but with one important difference. `EnvSupport` is optional and application may or may not make use of it. It has been realized as a specific kind of `~environment space~`, which can be added to the application description. This does also mean that an application may define multiple different environments if this is advantageous. Currently `EnvSupport` has the following main features and limitations.

Features:

- Declarative specification of the environment as `~application space~`
- Model definition consisting mainly of `~space objects~`, `~tasks~` and `~environment processes~` in a 2d or 3d grid or continuous world
- Agent-environment interaction via customizable `~percepts~` and `~actions~`
- 2d and 3d visualization of the environment, its objects and agents, including possibilities for animation etc.
- Customizable space execution with built-in support for continuous and round-based execution semantics
- Highly extensible with a lot of ready-to-use components for frequent use cases

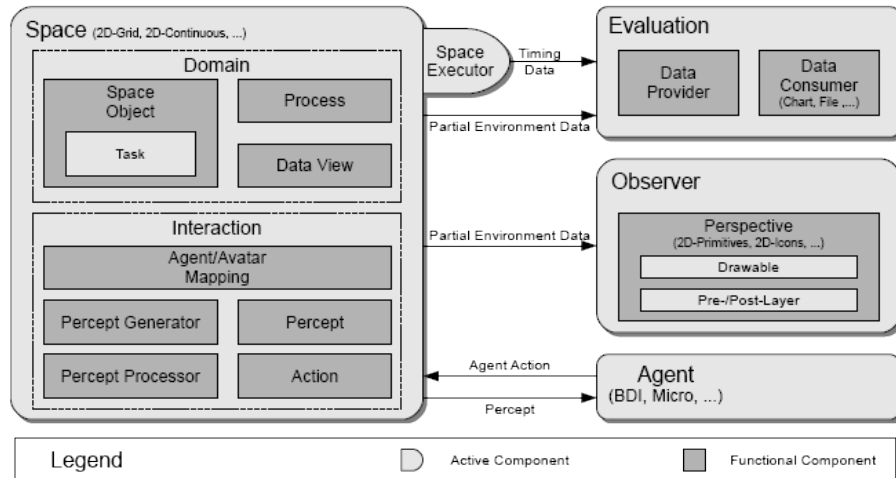
Current Limitations:

- No direct manipulation of the environment from the user interface
- No integrated collision detection
- Applications with `EnvSupport` cannot be distributed over several platform nodes

This guide will continue describing the details of `EnvSupport` following coarsely its internal structuring by `~domain model~`, `~agent environment interaction~` and `~visualization~`. These aspects will first be explained on a relatively high level with respect to their conceptual meanings and then again be picked up for a more detailed discussion on the programming level.

`$xwiki.ssx.use("XWiki.Lightbox")`

`$xwiki.jsx.use("XWiki.Lightbox")`



~Figure 1: Main conceptual building blocks of EnvSupport~

In the figure above the conceptual parts of EnvSupport and their interplay is shown. It consists of the parts ~spaces~, ~agents~, ~observers~ and ~evaluation~. The environment space mainly contains the ~domain model~ and the ~interaction~ specification.

1.1 Space

In the domain model the constituents of the space are declared. In EnvSupport these constituents are so called ~space objects~, which may represent arbitrary artifacts of the described world. This may include passive as well as active objects, whereby the activity can be exerted on the objects internally as well as externally. Internally space objects can be modified by ~tasks~ and ~processes~. A task is directly associated to a space object and encapsulates the object manipulation logic. In contrast, a process is not directly connected to a specific space object, but has instead a global scope and access to all objects of the space. A typical example for a task is a move function, which continuously changes the object's location based on the elapsed time and the current speed and direction. Considering processes, an example is heat dissipation, which distributes the heat on the landscape according to some physical formula. Besides the space objects, tasks and processes also ~data views~ can be specified. A data view is a definable cutout of the model world and hence similar to a database view in relation to the database. Currently, it is used to transfer a specific view of the world to dedicated observers, which can use these data for presentation purposes. Examples for data views are the built-in complete view representation the whole world and local avatar view, which contains only object in the vision range of the object.

The interaction part serves for the specification of how the interrelationship between agents and space objects is. In EnvSupport space objects which are representatives for agents in the virtual world are called ~avatars~. The connection between such avatars and agents are specified in so called ~avatar mappings~. It defines what happens when an agent or an avatar is created or destroyed. Both sides of an agent avatar association can be tied together, so that the creation of an agent automatically leads to the creation of its avatar. The same applies for the other direction, which means that the creation of an avatar also can initiate the creation of an associated agent. Such kind of connections can also be set-up for the destruction of agents or avatars. The communication between agents and the environment is organized via ~percepts~ and ~actions~. A percept is a meaningful event created in the environment and directed towards specific kinds of agents. In this way information is passed on a semantic level to the agents. The creation of percepts is performed by ~percept generators~, which can react on basic environment events, like the movement of a space object, and transform them to percepts. These percepts are then fed into the fitting ~percept processors~, which have the task to interact with the agent and inform it about the new percept. On the other hand agents can influence the environment by executing ~actions~ on it. These actions are predefined in the environment space and may modify arbitrary environmental properties and objects. The execution can directly be performed by an agent on the space.

As described until now, a space represents a passive entity. In order to allow also active environments containing tasks and processes, a ~space executor~ is used. Such a space executor is typically coupled to the clock of the platform and performs some kind of “execution cycle” at each time step. One aspect of this cycle is the processing of percepts and actions and executing the tasks and processes. Hence, a space executor encapsulates the environment execution semantics and can e.g. implement a round-based or a continuous execution scheme.

1.1 Observers

So far the internals of an environment have been considered. Observers represent user interfaces for watching an environment. It typically allows for viewing the current state of the environment and its objects. The definition of observers include two main concepts: ~data views~ and ~perspectives~. A ~data view~ is a cutout of the environment, which represents the view of a specific entity. One example is the local view of an object, which is e.g. defined by the radius of its sight. Another kind of view is the global data view encompassing the complete environment data model. Besides the selection of the data that should be presented also the way of its presentation is of importance.

The presentation of the data can be defined using ~perspective~. A perspective is composed of ~drawables~, which describe the way a space object will be rendered. Typical types of drawables include geometrical shapes like triangle, rectangle, circle and also image icons. The drawables are organized in a simplified version of a ~scenegrph~. This means that in a drawable multiple shapes can

be combined in arbitrary flavors. In addition to the relative placement and sizes of these drawable parts `~drawconditions~` can be used to determine if a part is currently visible. This allows to visualize special parts of a drawable depending on its current environmental conditions (e.g. using different icons for different movement directions). A perspective is organized in layers to which drawables can be assigned. The layers determine the order in which the elements are painted, so that earlier painted drawables may be partially or completely hidden by elements of a higher layer. In addition to drawables so called `~pre- and post-layers~` can be defined. These can be used to e.g. show background or foreground elements that are not necessarily backed by space objects.

1.1 Evaluation

In addition to the space itself and its visualization, an optional evaluation can be set up. An evaluation allows for collecting specific data of the space and processing it further. The collection of data is specified via `~data providers~`, which make use of a table based data format similar to relational databases. For this purpose data sources and data elements can be specified. The sources are typically space objects or collections of space objects. These sources are joined (the cartesian product is calculated) and for each combination of source objects a new data row is produced. The row consists of the defined data elements (e.g. attributes of objects). For each relevant time point the data provider can generate such a data table. Data providers are typically connected to data processors, which use them for fetching the input data. Data processors can come in very different flavors, from a simple file writer to graphical output generators for charts or histograms. The specification of data providers heavily depends on their concrete type so that the main commonality is the data source given by the data provider.

1 Domain Model

In this section the `~domain model~` concepts will be presented in detail. Each concept will be explained using the XML schema parts and additionally with example xml code snippets

1.1 Space Type Definition

All space types are defined with the `\<spacetypes\>` section of Jadex application descriptor. This section is an extension point (using xml-schema any element) and allows for custom space types to be plugged in. These specific space types typically come with their own schema definition and hence use tags within a separate namespace. For EnvSupport the outermost declaration is called `\<env:envspacetype\>` (using here 'env' as prefix for its namespace). The figure below shows the basic elements an environment is composed of.

`~XML schema part for the environment space type~`

It can be seen that a space type is generally specified by declaring `~objecttypes~`, `~avatarmappings~`, `~actiontypes~`, `~tasktypes~`, `~processtypes~`, `~percepttypes~`, `~views~`, `~perspectives~` and a `~spaceexecutor~`. For the domain model we

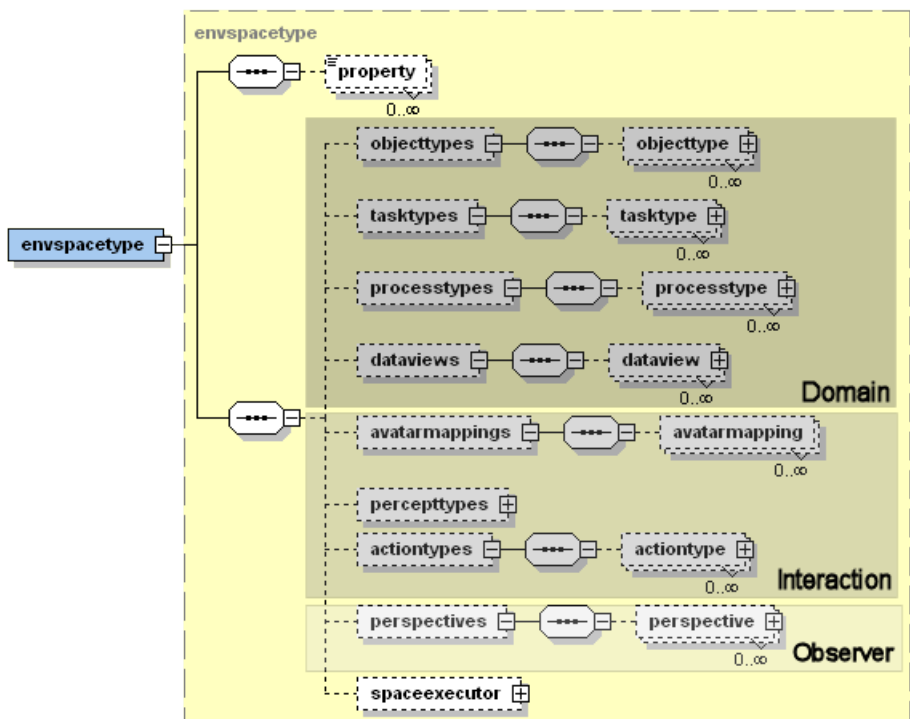


Figure 67:

will first cover the details of `~objecttypes~`, `~actiontypes~`, `~tasktypes~`, `~processtypes~` and the `~spaceexecutor~`. Before explaining these elements a we will have a short look on the `envspacetype` element itself.

In this basic definition the type name, its implementation class and also its extent (using width, height and optionally depth) as well as its border mode (as subtag) can be included. Currently, two different implementations are available: the `~ContinuousSpace2D~` for a continuous 2d world and `~Grid2D~` for a 2d grid world consisting of a discrete raster. Both implementation are contained in the package `'jadex.application.space.envsupport.environment.space2d'`, which has to be imported if the classes are not referenced fully qualified. The border mode determines the behaviour of the space objects at the borders of the 2d field. If set to 'strict' the world ends at the borders, whereas if the 'torus' mode is used the space objects can cross the borders and enter the field on the other side.

1.1.1 Object Type

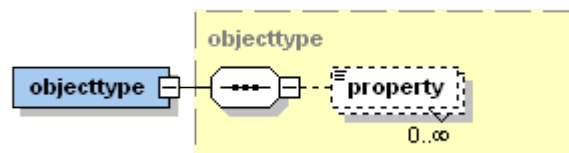


Figure 68:

`~Object type xml schema part~`

Object types are the foundation for space object instances. They represent some kind of simple class (better struct as it cannot contain behaviour) definition for an object. Each object type has to be uniquely identified via a `~name~` attribute. An object type may contain an arbitrary number of property type declarations. A property is then described using attributes for a `~name~` and a `~class~`. Additionally, two flags can be used. The `~dynamic~` flag determines how a property value is interpreted at runtime. If set to false the initial value is evaluated once and the result is stored. Otherwise, the declared value is treated as dynamic expression, which is reevaluated on every access. The `~event~` flag can be used to tell the system that property change events should be generated whenever the property value changes. Please note that also the parametrization of many other elements is done using the property mechanism as described above.

In the following code snippet a short example for an object type is shown. It is the avatar type sentry of the marsworld example. It has properties for its `~vision~` range, `~speed~` and `~position~`.

```
{code:xml}
<env:objecttype name="sentry">
```

```

<env:property name="vision" class="double">0.1</env:property>
<env:property name="speed" class="double">0.05</env:property>
<env:property name="position" class="IVector2" dynamic="true">
  $space.getSpaceObjectsByType("homebase")[0].getProperty("position")
</env:property>
</env:objecttype>
{code}
~Sentry avatar object type~

```

1.1.1 Task Type

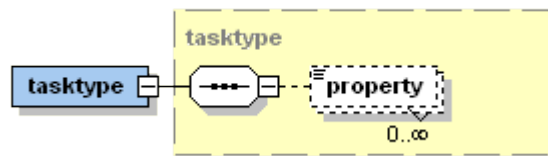


Figure 69:

~Task type xml schema part~

A task type can be used for encoding behaviour that is associated to a specific object at runtime. A type type has to be defined using attributes for a `~name~` and its implementation `~class~`. In addition to these attributes an arbitrary number of properties may be specified as subtags of the tasks. These properties can be used for initialization of task instances of the given task type. The implementation of a task type has to follow the `~IObjectTask~` interface (package `jadex.application.space.envsupport.environment`). It consists of four method signatures (cf. code snippet below). All these methods will be called automatically from the environment support runtime (space executor). The `~start()~` and `~shutdown()~` methods should contain code for the initialization and close down of the object task. The `~execute()~` is called in every step the space executor issues. It should contains the main logic of the task to perform. In order to quit a task automatically the `~isFinished()~` method can be used. As long as the task runs the method will be called to find out is the task has finished its execution. For programmer convenience an abstract task implementation `~AbstractTask~` (package `jadex.application.space.envsupport.environment`) is also available. Extending this abstract task requires only the `~execute()~` method being filled with code (if `~start()~` and `~shutdown()~` are overridden they should call `~super().start()/shutdown()~`). It already implements a solution for finishing tasks. On the one hand it is possible to call a `~setFinished()~` and manually terminate the task and on the other hand the task fetches a boolean condition via `~(IBooleanCondition)getProperty("condition")~` and calls `~isValid()~` on this condition to check if the task should still run.

```

{code:java}
package jadex.application.space.envsupport.environment;

```

```

public interface IObjectTask extends IPropertyObject
{
    public void start(ISpaceObject obj);

    public void shutdown(ISpaceObject obj);

    public void execute(IEnvironmentSpace space, ISpaceObject obj, long progress,
        IClockService clock);

    public boolean isFinished(IEnvironmentSpace space, ISpaceObject obj);
}
{code}
~IObjectTask interface~

```

As an example below a move task from the cleanerworld example is shown. It basically changes the avatar position according to its velocity and direction step by step. It also reduces the amount of available energy according the currently travelled distance. The task is set to finished when the target location has been reached.

```

{code:java}
public class MoveTask extends AbstractTask
{
    public static final String PROPERTY_DESTINATION = "destination";
    public static final String PROPERTY_SPEED = "speed";
    public static final String PROPERTY_VISION = "vision";
    public static final String PROPERTY_CHARGESTATE = "chargestate";

    public void execute(IEnvironmentSpace space, ISpaceObject obj, long progress,
        IClockService clock)
    {
        IVector2 destination = (IVector2)getProperty(PROPERTY_DESTINATION);
        double speed = ((Number)obj.getProperty(PROPERTY_SPEED)).doubleValue();
        double maxdist = progress*speed*0.001;
        double energy = ((Double)obj.getProperty(PROPERTY_CHARGESTATE)).doubleValue();
        IVector2 loc = (IVector2)obj.getProperty(Space2D.PROPERTY_POSITION);

        IVector2 newloc = ((Space2D)space).getDistance(loc, destination).getAsDouble()<=maxdist?
            destination : destination.copy().subtract(loc).normalize().multiply(maxdist).add(loc);

        if(energy>0)
        {
            energy = Math.max(energy-maxdist/5, 0);
            obj.setProperty(PROPERTY_CHARGESTATE, new Double(energy));
            ((Space2D)space).setPosition(obj.getId(), newloc);
        }
        else
        {

```



```

    throw new RuntimeException("Energy too low.");
  }

  if(newloc==destination)
    setFinished(space, obj, true);
}
{code}

```

1.1.1 Process Type

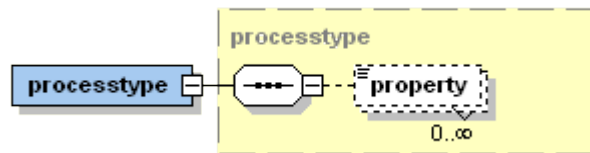


Figure 70:

~Process type xml schema part~

A process type is very similar to a task type. It also represents dynamic behaviour, but on a global level, i.e. it is not connected to a specific environment object but to the environment as a whole. The process type definition is done again using a ~name~ and a ~class~ attribute optionally extended with properties as subtags. A process type implementation has to be compliant to the ~ISpaceProcess~ interface (package `jadex.application.space.envsupport.environment`). This interface is similar to the task interface and also contains methods for starting (~start()~) and terminating (~shutdown()~) the process. The space executor calls ~execute()~ in each step, so that this method should contain the application logic of the process. It does not have a `isFinished()` method as in most cases processes determine themselves that they do not want to run any longer. This can be achieved by calling ~removeSpaceProcess()~ on the environment.

```

{code:java}
package jadex.application.space.envsupport.environment;

public interface ISpaceProcess extends IPropertyObject
{
    public void start(IClockService clock, IEnvironmentSpace space);
    public void shutdown(IEnvironmentSpace space);

    public void execute(IClockService clock, IEnvironmentSpace space);
}
{code}

```

~ISpaceProcess interface~

One frequent use case is the creation of domain objects by an environment process. For this purpose a customizable default implementa-

tion is provided by the `~DefaultObjectCreationProcess~` class (package `jadex.application.space.envsupport.environment`). It can be parametrized using the following parameters:

- `*type:` The type of the object to be created (String, required).
- `*properties:` The initial properties of the object (Map, optional).
- `*condition:` A condition to enable/disable object creation (boolean, optional).
- `*tickrate:` Number of ticks between object creation (double, optional, 0 == off).
- `*timerate:` Number of milliseconds between object creation (double, optional, 0 == off)

1.1.1 Data View

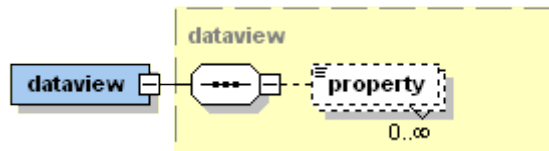


Figure 71:

`~Data view type xml schema part~`

A data view is similar to a data base view, i.e. it defines what cutouts of the environments can be perceived by the one using the view. Currently data views are mainly used by observers that use data views for presenting what can be seen from a specific perspective, e.g. from one avatar in the environment. A data view has attributes for `~name~` and `~class~` and optional for `~objecttype~`. The implementation class has to follow the `~IDataView~` interface shown in the figure below. It has to implement the methods `~init()`, `~getType()`, `~getObjects()` and `~update()`. `Init` is called once on initialization of the view. `Update` is called in each step from the space executor to refresh the view and `~getType()` return the type name of the view. The observer uses the `~getObjects()` method for fetching the current objects of the view.

```

{code}
package jadex.application.space.envsupport.dataview;

public interface IDataView
{
    public void init(IEnvironmentSpace space, Map properties);

    public String getType();
  
```

```

public Object[] getObjects();

public void update(IEnvironmentSpace space);
}
{code}
~IDataView interface~

```

There are two default implementations for views available. The `GeneralDataView2D` simply returns all the objects contained in the space, whereas the `LocalDataView2D` implementation can be used for creating personal views of specific objects. The local view implementation can be parametrized using the attribute `objecttype` specifying the type of objects for which a specific view is applicable and these properties:

- `*object:` The space object instance.
- `*range:` The range for determining objects around the creature.

1 Component Interaction

In this section the interaction concepts between components and space objects are described. The presented elements are part of the space environment type xml part as introduced in section [Domain Model>03 Domain Model]. For convenience the relevant cutout of the xml schema is shown again below.

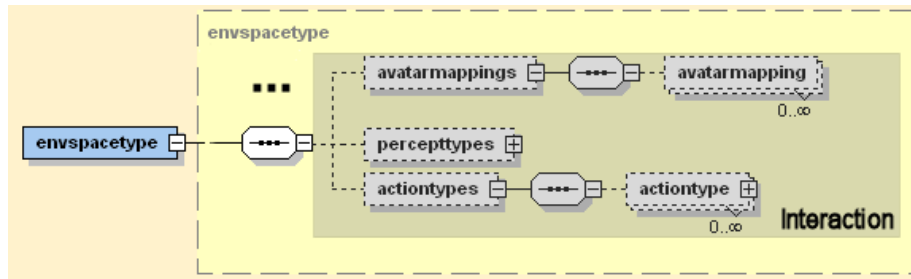


Figure 72:

~Interaction xml schema part of the environment space type~

1.1.1 Avatar Mapping

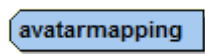


Figure 73:

~Avatar mappings xml schema part~

The avatar mapping is used to define the relation between space objects (avatars)

Name	Type	Use	Default
componenttype	xs:string	required	
objecttype	xs:string	required	
createavatar	xs:boolean		true
createcomponent	xs:boolean		false
killavatar	xs:boolean		true
killcomponent	xs:boolean		false

Figure 74:

and components (i.e. agents). This relationship determines what happens when a new space object of a specific type is created/deleted and on the other side what happens when a component is created/deleted. The connection between both can be established to mimic the behaviour from one side on the other side, e.g. create also an avatar (a space object) when a new component is created. The default values for the four flags `~createavatar~`, `~killavatar~`, `~createcomponent~`, `~killcomponent~` are defined in a way that the component side dominates the simulation world, i.e. whatever happens to a component is also done with its avatar in the simulation. If the creation and deletion of space objects should also lead to the creation and deletion of components this has to be explicitly set with the corresponding flags.

The code snippet below shows a fictitious robot example, which creates robot agents whenever a new robot space object is created. Also, for each newly created robot agent an avatar is initiated automatically.

```
{code:xml}
<env:avatarmapping componenttype="Robot" objecttype="robot" createcom-
ponent="true"/>
{code}
~Avatar mapping example~
```

1.1.1 Percept Types

~Percept types xml schema part~

Percepts are meaningful events for components, i.e. a percept can be seen as some kind of environment event that is addressed towards a specific kind of component. This means that different kinds of components may receive completely different percepts for the same observed events. Hence, a percept is a concept that connects a component with the environment. For this reason percept types include several aspects in EnvSupport. On the one hand, the basic kinds of percepts can be defined (inner percept types tag). On the other hand percept generators and processors need to be specified. A percept generator is responsible for creating percepts (based on the available percept types) and thus attached to the environment. These percepts are then consumed by components using percepts processors, which may depend on the concrete component type. E.g. a

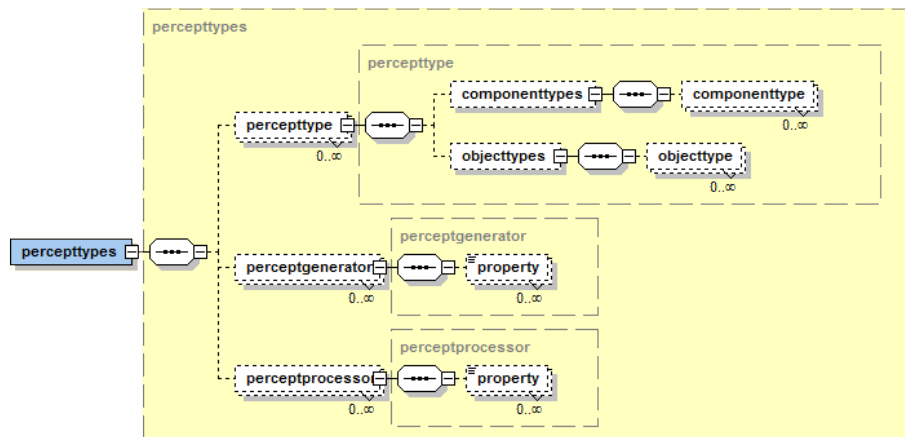


Figure 75:

bdi agent percept processor can directly store newly arriving percepts in fitting beliefs or beliefsets.

A percept type is described using a `~name~` attribute for percept type identification. Using the `~objecttype~` attribute (or tags for multiple types) the underlying meaning of the percept can be defined. The `~componenttype~` attribute (or tags for multiple types) is useful for specifying the component types that can generally perceive such kind of percept. Property tags can be employed for parametrization as for most other elements.

1.1.1.1 Percept Generators

A percept generator is defined using `~name~` and implementation `~class~` attributes. An implementing class has to extend the `~IPerceptGenerator~` interface. Currently, with the `~DefaultVisionGenerator~` (package `jadex.application.space.envsupport.environment.space2d`) a quite powerful default implementation for a percept generator, capable of creating vision range dependent percepts, exists.

```
{code:java}
<env:perceptgenerator name="visiongen" class="DefaultVisionGenerator">
  <env:property name="range">0.1</env:property>
  <env:property name="range_property">"vision_range"</env:property>
  <env:property name="percepttypes">
    new Object[]
    {
      new String[] {"cleaner_moved", "moved"},
      new String[] {"waste_appeared", "appeared", "created"},
      new String[] {"waste_disappeared", "destroyed"},
      new String[] {"wastebin_appeared", "appeared", "created"},
    }
  </env:property>
</env:perceptgenerator>
```

```

        new String[]{"wastebin_disappeared", "destroyed"},
        new String[]{"chargingstation_appeared", "appeared", "created"},
        new String[]{"chargingstation_disappeared", "destroyed"}
    }
</env:property>
</env:perceptgenerator>
{code}

```

~Default vision generator example from cleanerworld example~

In the code snippet above an example for a vision generator specification is shown. The configuration of the default vision generator is done using the `~range~`, `~range_property~` and `~percepttypes~` properties. The range determines the radius of the vision the avatar can perceive (all elements within the radius can be seen). In order to find out what range to use the following steps are performed: 1) try to get the range as property from the avatar using the `range_property` (if not specified "range" is tried) 2) if no range value could be obtained the default range is used by fetching the `~range~` property of the vision generator.

The percept types are defined using String arrays of the form of a perceptname and an arbitrary number of actionnames. In order to find the correct percept type for a specific event consisting of `~componenttype~`, `~objecttype~` and `~actiontype~` the percepttype defined in the space is fetched and it is checked if the `componenttype` and `objecttype` of the event fit to those of the percepttype. Thereafter, the `actiontype` of the event is compared with those declared in the vision processor. Please note the the default vision processor supports the following action types:

- `*created:*` A space object has been created.
- `*destroyed:*` A space object has been destroyed.
- `*appeared:*` A space object came into the vision range and was not seen before.
- `*disappeared:*` A space object moved out the vision range and was seen before.
- `*moved:*` A space object changed its position.

1.1.1.1 Percept Processors

A percept processor is specified using `~componenttype~` and implementation `~class~` attributes. The first is used to define in component types the percepts shall be injected. The latter has to extend the `~IPerceptProcessor~` interface shown below.

```

{code:java}
package jadex.application.space.envsupport.environment;

public interface IPerceptProcessor extends IPropertyObject
{
    public void processPercept(IEnvironmentSpace space, String type,
        Object percept, IComponentIdentifier component, ISpaceObject avatar);
}

```

```

}
{code}
~Percept processor interface~

```

For BDI agents a default implementation called `~DefaultBDIVisionProcessor~` exists.

```

{code:java}
<env:perceptprocessor componenttype="Cleaner" class="DefaultBDIVisionProcessor"
>
  <env:property name="percepttypes">
    new Object[]
    {
      new String[]{"cleaner_moved", "remove_outdated", "wastes"},
      new String[]{"waste_appeared", "add", "wastes"},
      new String[]{"waste_disappeared", "remove", "wastes"},
      new String[]{"wastebin_appeared", "add", "wastebins"},
      new String[]{"wastebin_disappeared", "remove", "wastebins"},
      new String[]{"chargingstation_appeared", "add", "chargingstations"},
      new String[]{"chargingstation_disappeared", "remove", "chargingstations"}
    }
  </env:property>
</env:perceptprocessor>
{code}

```

`~Default vision processor example from cleanerworld example~`

In the code snippet above an example for a vision processor specification is shown. The configuration of the default vision processor is done using the `~range~`, `~range_property~` and `~percepttypes~` properties. The first two range properties are only of importance for the `"remove_outdated"` action as here the objects that are not seen any longer need to be determined. The concrete range is determined in the same way as explained in the text of vision generators. The syntax for specifying the `percepttypes` is `~perceptname, action, belief(set)name, conditionname~`. It means that the action will be executed when the named percept occurs and the condition evaluates to true. The condition itself has to be specified as named property of the vision processor. The available actions are shown below.

- `*add:*` Add a percept to a beliefset.
- `*remove:*` Remove a percept to a beliefset.
- `*remove_outdated:*` The `remove_outdated` action checks all entries in the belief set, if they should be seen, but are no longer there.
- `*set:*` Set the percept as fact of a belief.
- `*unset:*` Set a belief to null.

1.1.1 Action Type

`~Action type xml schema part~`

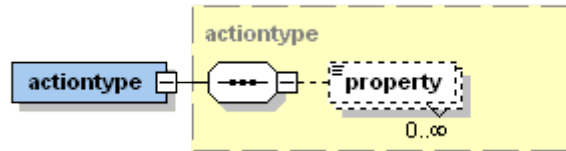


Figure 76:

Action types are used for specifying which kinds of action can be issued on an environment. It is also uniquely defined by a type `~name~` and additionally an action `~class~`. This action class has to implement the `~ISpaceAction~` interface (package ‘jadex.application.space.envsupport.environment’) and is itself an extension of the `~IPropertyObject~` (package ‘jadex.commons’) interface. The interface `ISpaceAction` only contains one method that has to be implemented by all action types. It is named `~perform(Map parameters, perform(Map parameters, IEnvironmentSpace space)~` and should contain the procedural code for the action. It has access to the space itself via the `~IEnvironmentSpace~` interface (you can cast if you need a concrete subclass) and to a map of action type specific parameters (name - value pairs). The `~IPropertyObject~` interface requires the action type to have getter and setter methods for properties. This allows a very flexible way of action type configuration from the xml. The declared property values in the xml can directly be accessed via the `~getProperty(String name)~` method.

1 Visualization

In this section the visualization concepts between components and space objects are described. The presented elements are part of the space environment type xml part as introduced in section [Domain Model>03 Domain Model]. For convenience the relevant cutout of the xml schema is shown again below.



Figure 77:

`~Visualization xml schema part of the environment space type~`

The EnvSupport visualization is clearly decoupled from the domain and interaction aspects. It is defined using `~perspectives~`, whereby multiple perspectives can be specified for the same domain model. At runtime between these perspectives can be switched as needed. A perspective has an identifying `~name~` attribute and is composed of `~property~` elements as well as arbitrary many

drawables and pre- and postlayers.

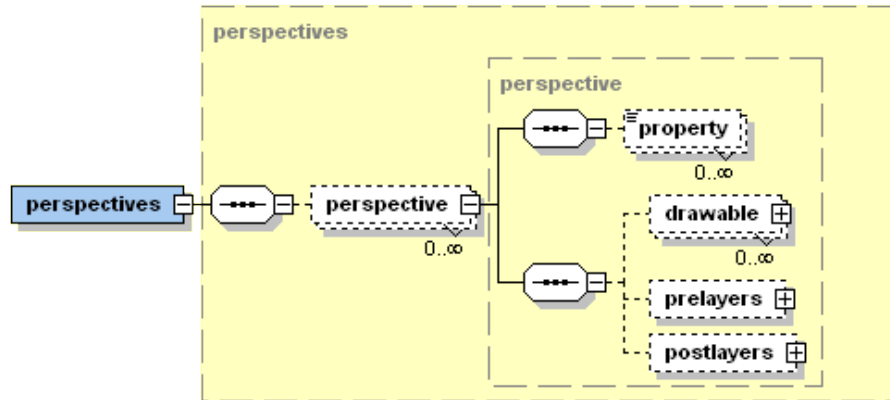


Figure 78:

~Perspectives declaration part of XML schema~

1.1.1 Drawable

~Drawable declaration part of XML schema~

A drawable represents the visual counterpart of a space object type. As can be seen in the schema part above it consists of ~property~ elements and arbitrary many geometrical shapes composing the look of the object (triangle, rectangle, ellipse, regularpolygon, texturedrectangle and text). The drawable itself is further specified by associating it to a specific ~objecttype~. Additionally, the width, height and rotation can be set. Each geometrical shape contained in the drawable is of type ~drawableelement~ and has the attributes described in the list below.

- ***layer:*** An integer for specifying the drawing layer (0 is default).
- ***position:*** An IVector2 or String for the 2d position of the object (relative to the drawable size). If it is a String it must be a property name of the drawable the value is bound to.
- ***size:*** An IVecor2 or String for the shape size (relative to the drawable size). If it is a String it must be a property name of the drawable the value is bound to.
- ***rotation:*** An IVecor3 or String for the rotation according to x, y, z axis (relative to the drawable rotation). If it is a String it must be a property name of the drawable the value is bound to.
- ***x:*** The x value of the position (double or int, relative to the drawable x).
- ***y:*** The y value of the position (double or int, relative to the drawable y).
- ***width:*** The width value of the position (double or int, relative to the drawable width).

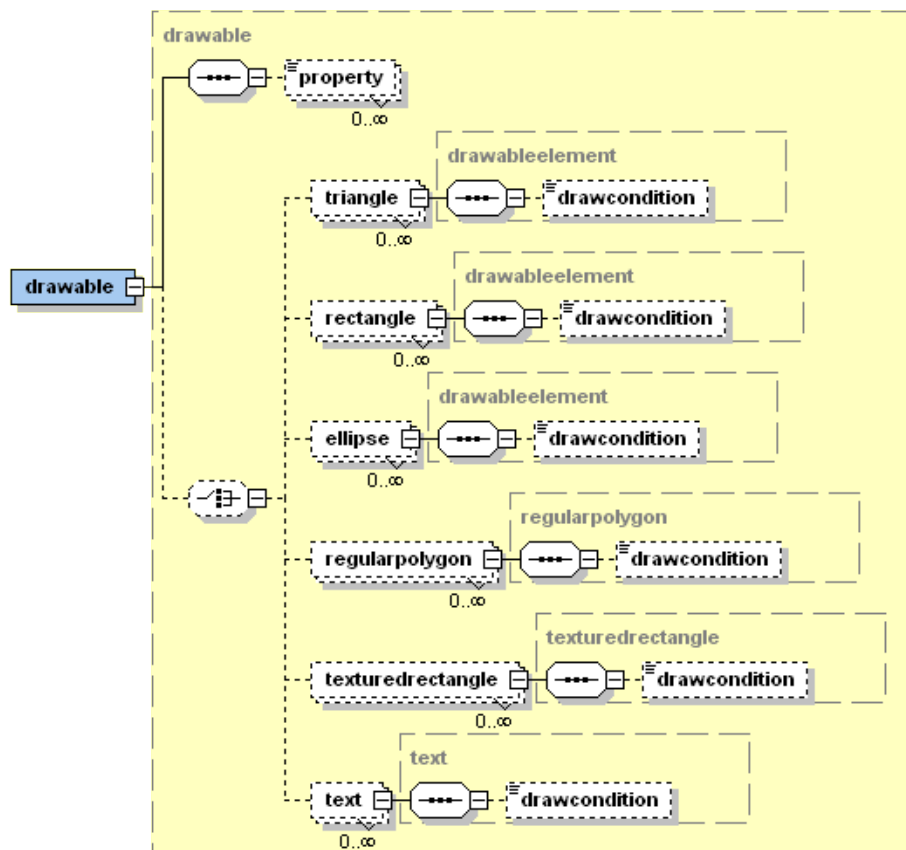


Figure 79:

- `*height:` The height value of the position (double or int, relative to the drawable height).
- `*rotatex:` The rotation x value of the position (double or int, relative to the drawable rotation x).
- `*rotatey:` The rotation y value of the position (double or int, relative to the drawable rotation y).
- `*rotatez:` The rotation z value of the position (double or int, relative to the drawable rotation z).
- `*abspos:` Absolute position value (IVector2 or String).
- `*abssize:` Absolute size value (IVector2 or String).
- `*absrot:` Absolute rotation value (IVector2 or String).
- `*color:` Color value of the shape (String).

Moreover, some elements have certain special properties, which are not applicable to all Drawables. The easiest way to find out about the properties is to use eclipse's auto-complete to gain an overview. The Text drawable for example may be further customized with these attributes:

- `*font:` The name of the font.
- `*style:` An integer representing the font's style. Use one of `Font.PLAIN`, `Font.BOLD`, or `Font.ITALIC`
- `*size:` An integer for specifying the text's size (12 is default).
- `*align:` The alignment can either be left, right, or center.

1.1.1.1 Example

```
{code:java}
<env:drawable objecttype="prey" width="1.0" height="1.0">
  <env:texturedrectangle layer="1" width="0.9" height="0.9" imagepath="jadex/micro/examples/hunterprey/in
  />
  <env:rectangle layer="-1" width="3" height="3" color="#ff00007f" />
  <!--red-->
  <env:rectangle layer="-1" width="1" height="1" x="-2" y="0" color="#00ff007f"
  /> <!--green-->
  <env:rectangle layer="-1" width="1" height="1" x="2" y="0" color="#0000ff7f"
  /> <!--blue-->
  <env:rectangle layer="-1" width="1" height="1" x="0" y="-2" color="#ffff007f"
  /> <!--yellow-->
  <env:rectangle layer="-1" width="1" height="1" x="0" y="2" color="#00ffff7f"
  /> <!--turquoise-->
</env:drawable>
{code}
```

~Drawable example taken from the hunter prey scenario~

In the example above, one drawable is declared. As you can see in the first line, this drawable is mapped onto objects of type prey and the whole drawable spans exactly one grid cell in width and height. As this drawable represents a prey and its vision, two layers are defined. One layer on which the icon is drawn and

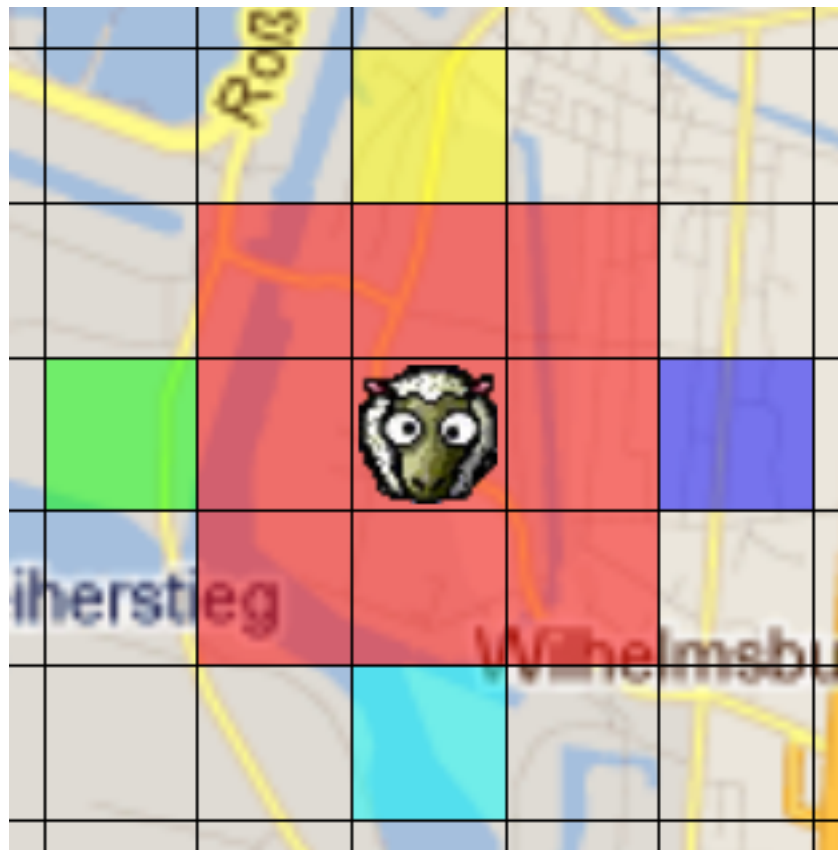


Figure 80:

below this a layer for the vision. The icon shall be scaled to fill 90% of the grid cell (width=0.9 and height=0.9). The vision consists of five rectangles, a big one spanning 3x3 grid cells and four smaller ones. The x- and y- coordinates represent the horizontal and vertical offset. The color attribute declares, how a rectangle shall be filled, e.g. #00ff007f reads as 00 (=red), ff (=green), 00 (=blue) and 7f (= alpha, i.e. transparency).

1.1.1 Pre and post layers

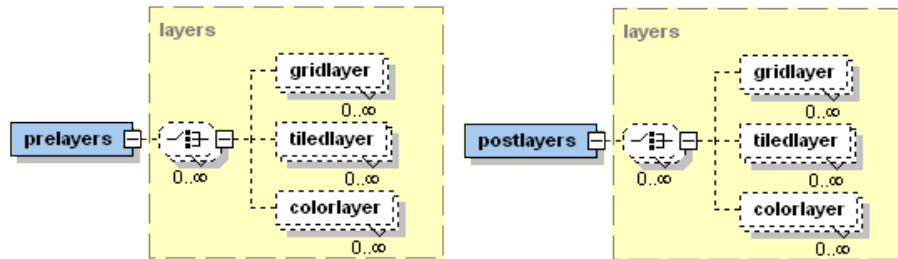


Figure 81:

~Pre and postlayer declarations part of XML schema~

Prelayers can be used for displaying graphical background elements. These layers are painted before the drawables and postlayers. On the contrary, postlayers are drawn after all other elements. For both layer groups the above depicted kinds of layers can be defined. A ~gridlayer~ can be used to paint a grid structure, typically for visualization of Grid2D environments. It is specified using attributes for the ~color~ of the grid and the ~width~ and ~height~ of the grid cells. In contrast, a ~tiledlayer~ can be used to paint an image repeatedly. It is declared using an ~imagepath~ for the image file name, ~width~ and ~height~ attributes for the tile size and additionally a ~color~ can be defined. The color is modulated on the tiles and changes the underlying look of the tiles. The simplest layer type is the ~color~ layer, which is defined by setting ~color~ attributes. A color layer changes the background color to the selected color value.

1.1.1.1 Example

```
{code:java}
<!--Draw the background-->
<env:prelayers>
  <env:tiledlayer width="0.4" height="0.4" imagepath="jadex/micro/examples/hunterprey/images/background.png"/>
</env:prelayers>

<!--draw a black grid upon everything-->
<env:postlayers>
```

```

<env:gridlayer width="1.0" height="1.0" color="black" />
</env:postlayers>
{code}

```

~Pre- and postlayer example taken from the hunter prey scenario~

In the example above, the prelayer is used to draw the background. In this case an image, which spans only 40% of the overall grid. Because of this, the image is repeatedly drawn horizontally and vertically. If, for example, a street map shall be used, just set width and height to 1.0. The post layer, in contrast to the tiled background layer, is a grid, spanning the whole environment. The row and column count of the grid correspond to the width and height of the ~envspacetype~.

BEGIN MACRO: box param: cssClass="floatinginfobox" title="Contents"

END MACRO: box

3D Visualization

In this section the 3D visualization concepts are described. This section only focuses the **use** of existing 3D-objects in the Enviroment. The creation of new 3D Objects for Jadex 3D is a complex topic which will later in a different chapter



A running 3D Example

Coordinate System

The coordinate system consists of:

- The origin, a single point in space.
 - This point is always at coordinate (0,0,0)
- Three coordinate axes that are mutually perpendicular, and meet in the origin.
 - The X axis is “right/left”
 - The Y axis is “up/down”
 - The Z axis is “towards you/away from you”

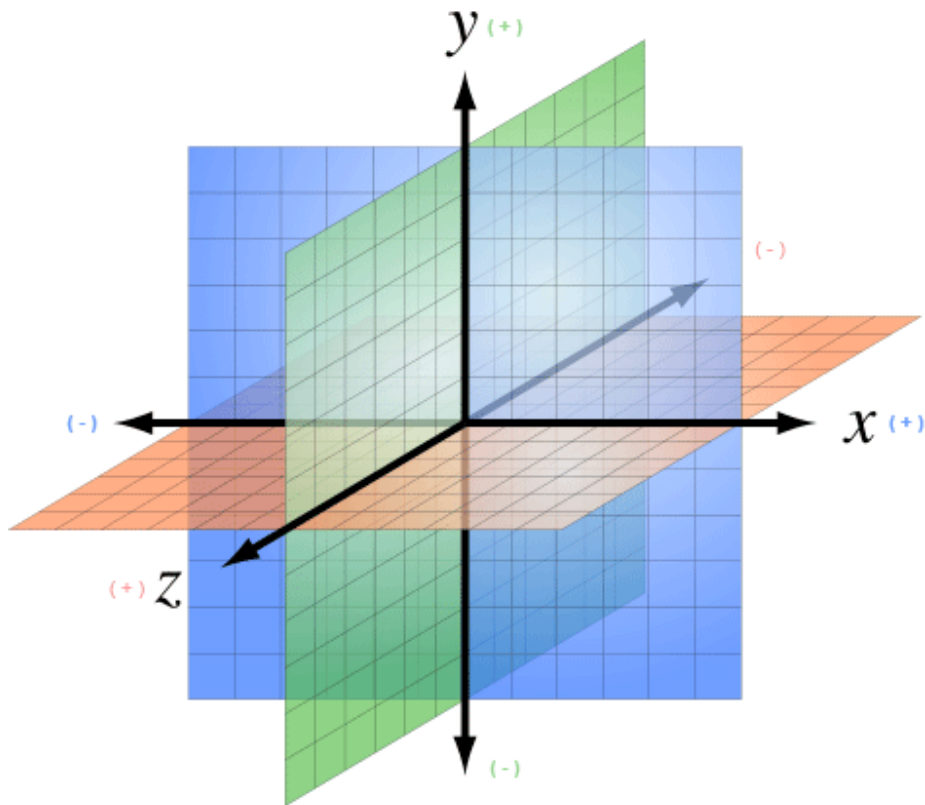


Figure 82:

Every point in 3D space is defined by its (x,y,z) coordinates. The data type for vectors is `Vector3Double` or `Vector3Int`.

Drawable3d declaration part of XML schema

A **drawable3d** represents the visual counterpart of a space object type. As can be seen in the schema part above it consists of property elements and arbitrary many geometrical meshes composing the look of the object (sphere, box, cylinder, dome, torus, object3d, arrow, text3d, sky, terrain, rndterrain and sound3d).

The primitive mesh-elements are sphere, box, cylinder, dome, torus and object. Complex mesh types are object3d (which loads a complex 3d mesh from a file) and terrain (generator for a 3d-terrain ground). Text3d, sky and sound3d can be considered as special types. We will describe all these types in detail later. The drawable3d itself is further specified by associating it to a specific objecttype. Additionally, the width, height and rotation can be set. Each geometrical mesh based shape contained in the drawable is of type **drawabeelement** and has the attributes described in the list below.

For all Elements, the **Drawable3d** and the Visual Objects inside have at least the three Attributes Position, Size and Rotation.

For instance, if you set the Size for the **Drawable3d** all Objects inside are influenced. If you set the Size for just one Visual Object only the Object is influenced relative to the size value of the **Drawable3d**.

Name	Default Value	Description	Type	position	Vector3Double(0,0,0)	An IVector3 or String for the 3d position of the object (relative to the drawable position)
------	---------------	-------------	------	-----------------	----------------------	---

Use this or x, y, z instead (then you have to set all three) Vector3Int

Vector3Double	x	0	The x value of the position	int / double	y	0	The y value of the position	int / double	z	0	The zvalue of the position	int / double
---------------	---	---	-----------------------------	--------------	---	---	-----------------------------	--------------	---	---	----------------------------	--------------

Name	Default Value	Description	Type	size	Vector3Double(1,1,1)	An IVecor3 or String for the shape size (relative to the drawable size).
------	---------------	-------------	------	-------------	----------------------	--

Use this or width, height, depth instead (then you have to set all three).

If it is a String it must be a property name of the drawable the value is bound to Vector3Double *Use this or width, height, depth instead (then you have to set all three).*

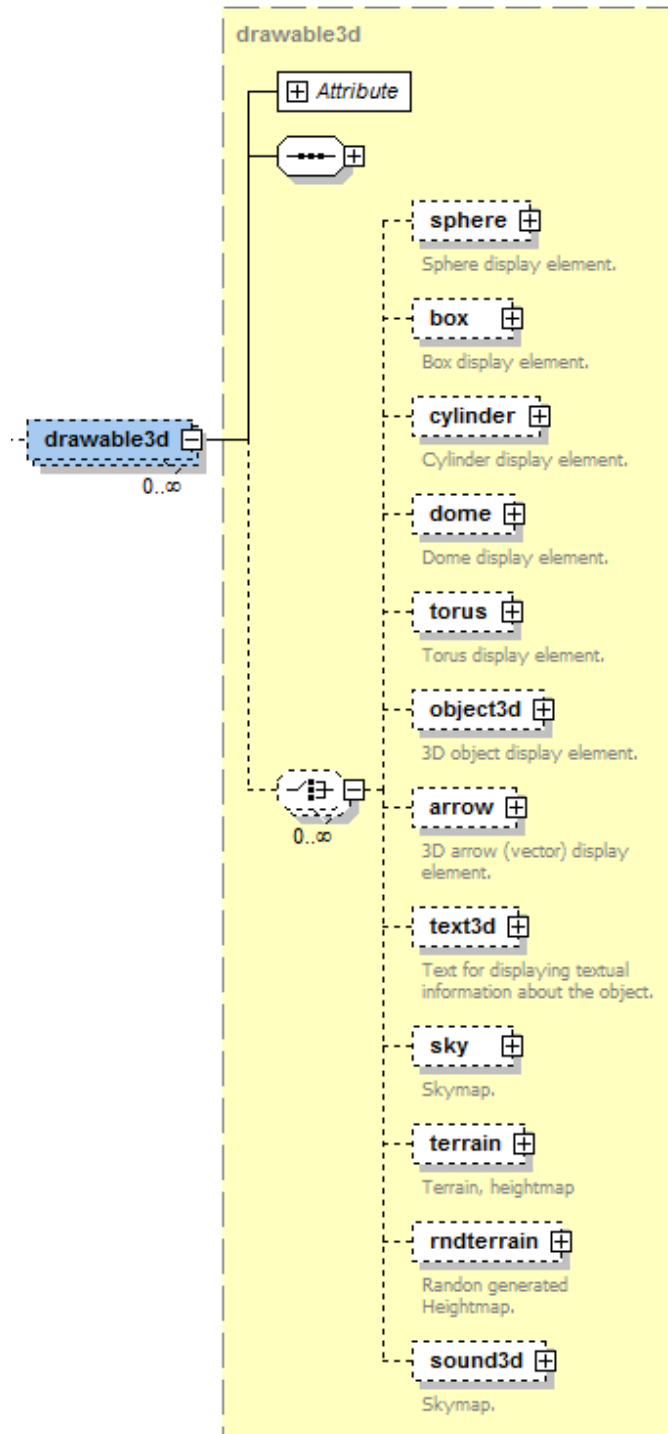


Figure 83:
393

width 1 The height value of the size int / double
 height 1 The depth value of the size int / double
 depth 1 The depth value of the size int / double

Name	Default Value	Description	Type
rotation	0	An IVecor3 or String for the rotation according to x, y, z axis (Default is (0,0,0) the drawable rotation) in radians. <i>Use this or rotate_x, rotate_y, rotate_z instead (then you have to set all three).</i>	String or Vector3Double
rotation _x	0	The rotation value along the x axis	int / double
rotation _y	0	The rotation value along the y axis	int / double
rotation _z	0	The rotation value along the z axis	int / double

Beside this very basic Attributes all the Visuals (not the Drawable3d itself) have this three attributes:

Name	Default Value	Description
shadowType	“Off”	A String that indicates if this object should be rendered with shadows. Va
color	“#FFFFFF”	The color of the visual object. Notation is in RGB, # following by the Rec
texturepath	“”	If you want to use a texture instead of just a color for the primitive put th

Moreover, some elements have certain special properties, which are not applicable to all Drawables. The easiest way to find out about the properties is to use eclipse's auto-complete to gain an overview. We will describe this special Attributes in detail for each visual Object if necessary in the list of predefined visuals below.

Rotation in Detail

You can use frequently used predefined rotations for every angle. The predefined values are 45, 90, 135, 180, 235 and 270 degree. To use it just type `rotation="$DEG{value}{x|y|z}"`.

For example if you want to rotate 180 degree on the y-angle just type `rotation="$DEG180y"`.

To define the rotation by yourself just remember its defined in radians. To use degree you have to calculate it by $\text{Pi}/180 \cdot \text{value}$. For example if you want to rotate 40 degree on x, 170 degree on y and 80 degree on z just type:

```
rotation="new Vector3Double((Math.PI/180)*40, (Math.PI/180)*170, (Math.PI/180)*80)"
```

The three Dimensions are explained in the Picture below:

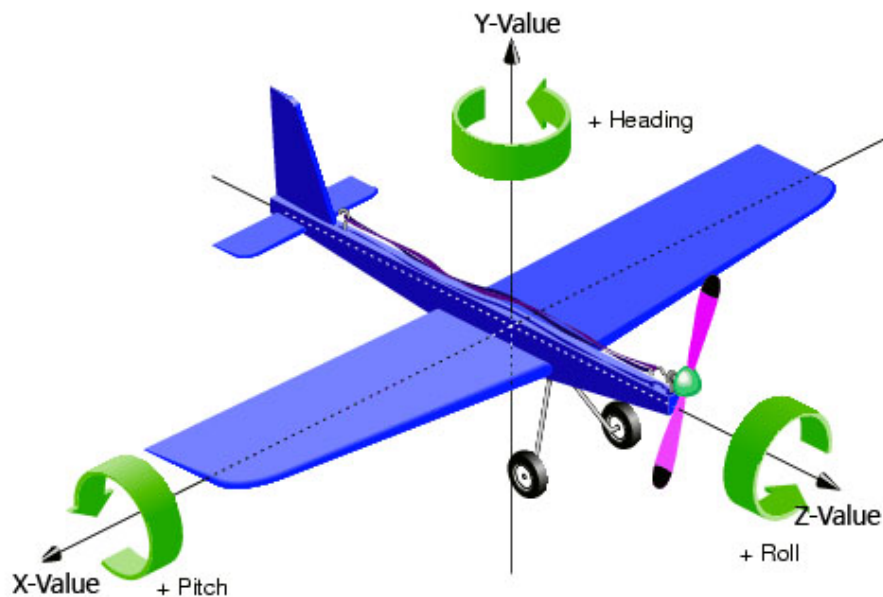


Figure 84:

ShadowType Example

In this example we have four objects who cast shadows and one plane below this objects to Receive the shadows.
You can see the result in the rendered Picture.

```
<env:drawable3d objecttype="character" hasSpaceobject="true" width="1" height="1" depth="1"
  <env:box width="10" height="0.01" depth="10" x="0" y="1" z="0" color="#FFFFFF" shadowtype="C"
  <env:dome width="1" height="1" depth="1" x="0" y="1.75" z="0" color="#7FFF00" shadowtype="C"
  <env:dome width="1" height="1" depth="1" x="5" y="1.75" z="5" color="#FFFF00" shadowtype="C"
  <env:cylinder width="1" height="1" depth="1" x="5" y="1.75" z="0" color="#0000FF" shadowtyp
  <env:cylinder width="1" height="1" depth="1" x="0" y="1.75" z="5" color="#FF0000" shadowtyp
</env:drawable3d>
```

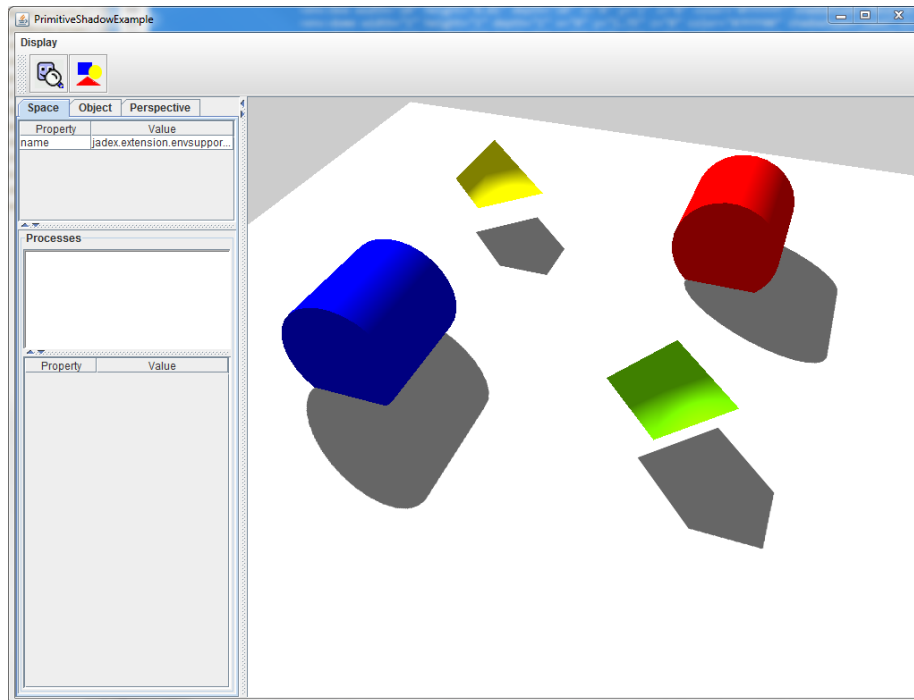


Figure 85:

List of usable predefined 3D-Primitives

Now we describe the easiest Objects you can use to make an jump-start for you Application possible. These Objects are sphere, box, cylinder, dome, torus and

arrow. You can replace them later to use complex 3d-Objects you create or get from external sources.

Sphere

Additional Values: (none)

Name	default value	Description	Type
------	---------------	-------------	------

Example:

```
<env:drawable3d width="1" height="1" depth="1" rotation3d="true">  
  <env:sphere width="1" height="1" depth="1" color="#FF0000">  
  </env:sphere>  
</env:drawable3d>
```

Box

Additional Values: (none)

Name	default value	Description	Type
------	---------------	-------------	------

Example:

```
<env:drawable3d objecttype="character" width="1" height="1" depth="1" rotation3d="true">  
  <env:box width="1" height="1" depth="1" color="#0000FF">  
  </env:box>  
</env:drawable3d>
```

Cylinder

Additional Values: radius

Name	default value	Description
radius	1	The Radius of the Cylinder. Default is 1. You can change the apperance for the cyl

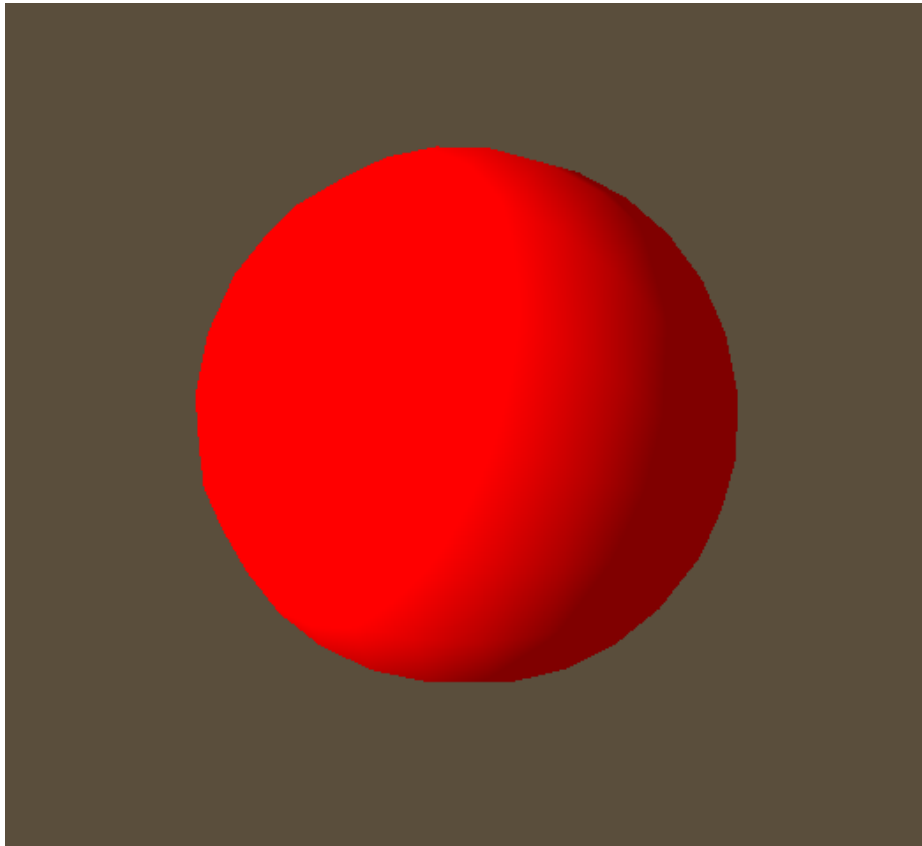


Figure 86:

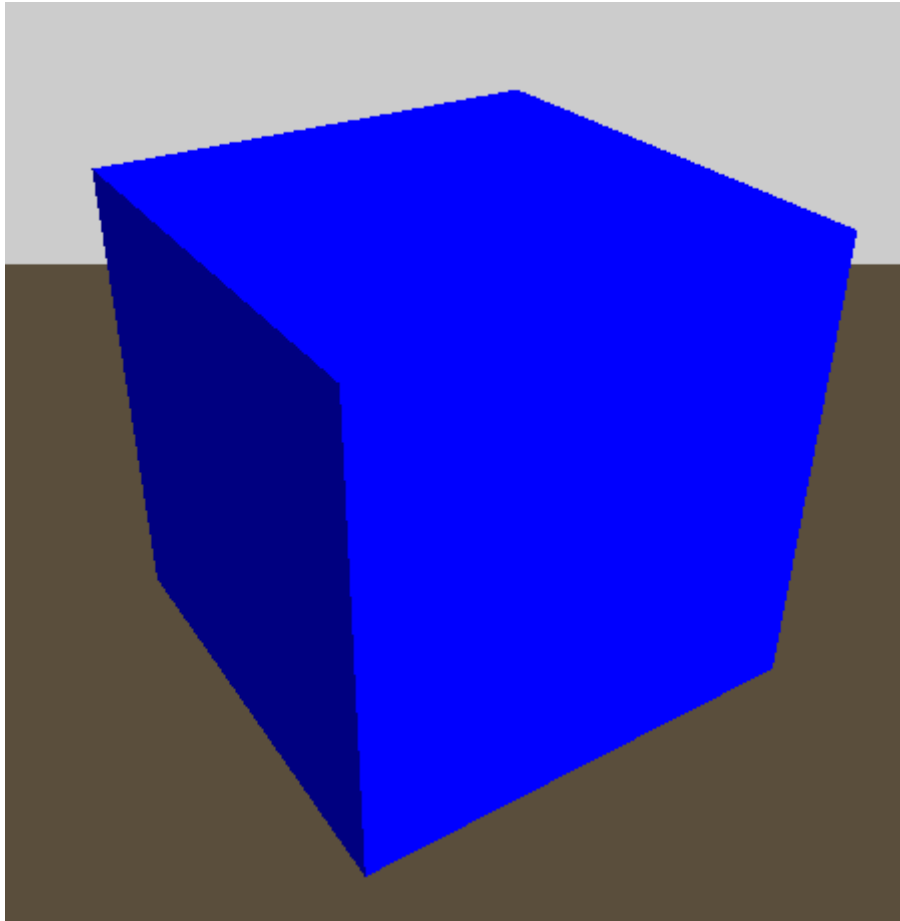


Figure 87:

As you can see from the example, you have to rotate a Cylinder 90 degree on the x oder z axis to get it in a “standing” position.

You only need the **height** and the **radius** value to define a Cylinder.

Example:

```
<env:drawable3d objecttype="character" width="1" height="1" depth="1" x="5" y="0" z="5">
  <env:cylinder height="1" radius="2" rotation="$deg90x" x="0" y="1.75" z="0" color="#000000">
  <env:cylinder height="2" radius="0.3" rotation="$deg90y" x="5" y="1.75" z="5" color="#FF0000">
  <env:cylinder height="1" x="5" y="1.75" z="0" color="#0000FF"/> <!-- Blue one -->
  <env:cylinder height="5" radius="0.5" rotation="$deg90x" x="0" y="1.75" z="5" color="#FF0000">
  <env:cylinder height="0.01" radius="10" rotation="$deg90x" x="0" y="1" z="0" color="#FFFFFF">
</env:drawable3d>
```

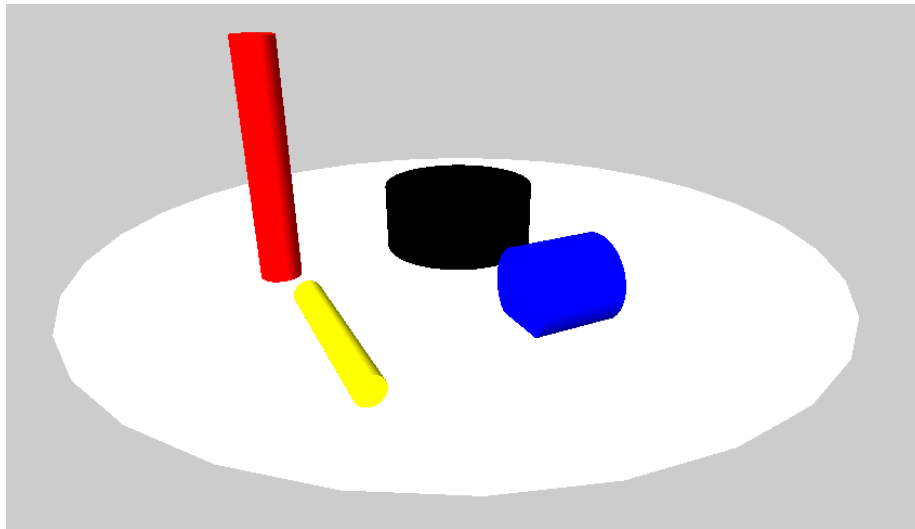


Figure 88:

Dome

Additional Values: (none)

Name	default value	Description
radius	1	The Radius of the Dome.
planes	1	How rounded the Dome is at the sides.
samples	4	The samples. It means how round the Domes Ground is. So it ´s more like a pyram

Look at the Picture to understand the influence of planes and samples.

Example:

```
<env:drawable3d objecttype="character" width="1" height="1" depth="1" x="5" y="0" z="5">  
<env:dome width="2" height="2" depth="2" planes="30" samples="30" x="0" y="1.75" z="0" color="#FF0000">  
<env:dome width="1" height="2" depth="1" radius="1" rotation="$deg45y" x="5" y="1.75" z="5" color="#FF0000">  
<env:dome width="2" height="1" depth="2" samples="10" x="5" y="1.75" z="0" color="#0000FF">  
<env:dome width="1" height="1" depth="1" planes="10" radius="2" x="0" y="1.75" z="5" color="#0000FF">  
<env:dome width="1" height="1" depth="1" samples="6" radius="2" x="3" y="1.75" z="7" color="#0000FF">  
<env:cylinder height="0.01" radius="200" rotation="$deg90x" x="0" y="-2" z="0" color="#FF0000">  
</env:drawable3d>
```

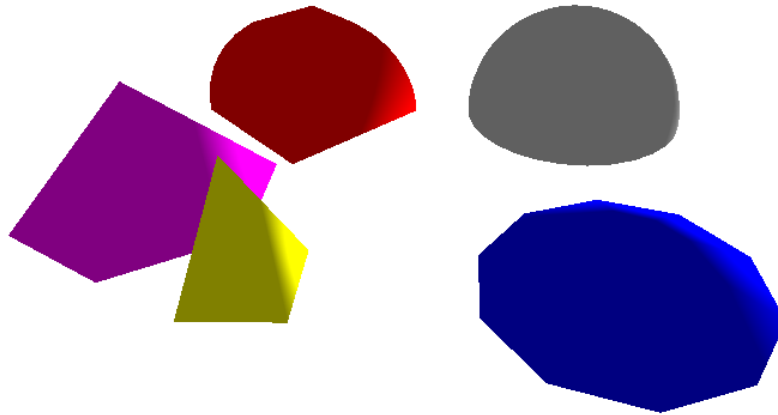


Figure 89:

Torus

Additional Values: (none)

Name	default value	Description
outerRadius	1	The outer Radius
innerRadius	0.1	The inner Radius of the Torus-Ring
circleSamples	40	How many Samples used for rendering the circle from the outer Radius. If s
radialSamples	20	The Samples for the Torus-Ring

Example:

```

<env:drawable3d objecttype="character" width="1" height="1" depth="1" x="5" y="5" z="5">
  <env:torus width="2" height="2" depth="2" innerRadius="0.1" outerRadius="2" circleSamples=
  <env:torus width="2.3" height="2.3" depth="2.3" innerRadius="0.1" outerRadius="2" circleSam
  <env:torus width="2.7" height="2.7" depth="2.7" innerRadius="0.1" outerRadius="2" circleSam
  <env:torus width="3" height="3" depth="10" innerRadius="0.1" outerRadius="2" circleSamples=
  <env:torus width="3" height="3" depth="3" innerRadius="1" outerRadius="2" circleSamples="30
  <env:torus rotation="$deg90x" width="10" height="10" depth="10" innerRadius="0.02" outerRac
  <env:cylinder height="0.01" radius="200" rotation="$deg90x" x="0" y="-5" z="0" color="#A0F0
</env:drawable3d>

```

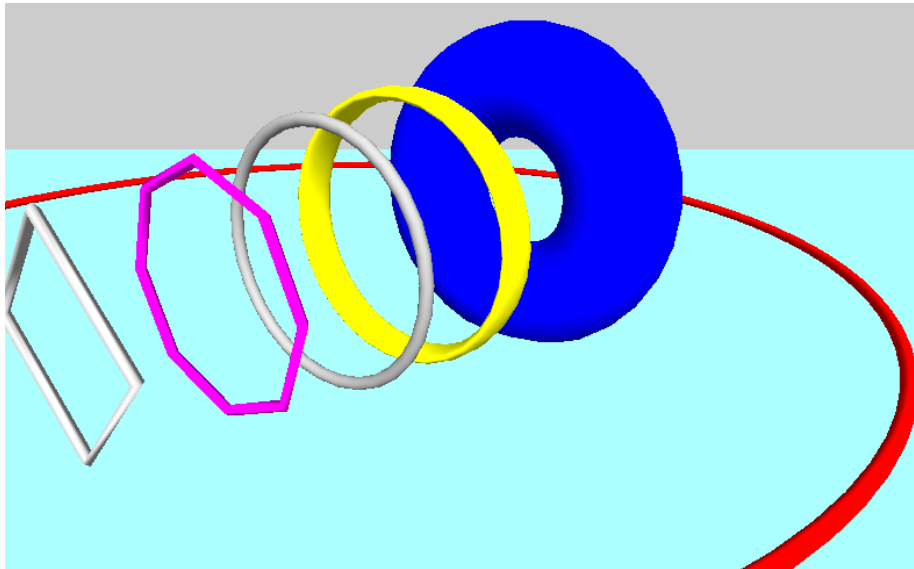


Figure 90:

Arrow

Additional Values: (none)

The Arrows are always one pixel wide. So the only thing you can set is the Direction (width, height, depth) and the origin Point(x,y,z).

Name	default value	Description	Type

Example:

```

<env:property name="rotate45" dynamic="false">new Vector3Double(0, (Math.PI/180)*45, 0)</env:

```

```

<env:box width="100" height="0.1" depth="100" x="-50" y="-1" z="50" rotation="rotate45" col
<env:arrow width="0.01" height="0.2" depth="0" x="0" y="0" z="0" color="#FFFFFFF"></env:ar
<env:arrow width="0.2" height="0" depth="0.01" x="0" y="0" z="0" color="#FF0FFFF"></env:ar
<env:arrow width="0" height="0.01" depth="0.2" x="0" y="0" z="0" color="#FFF00FF"></env:ar

</env:drawable3d>

```

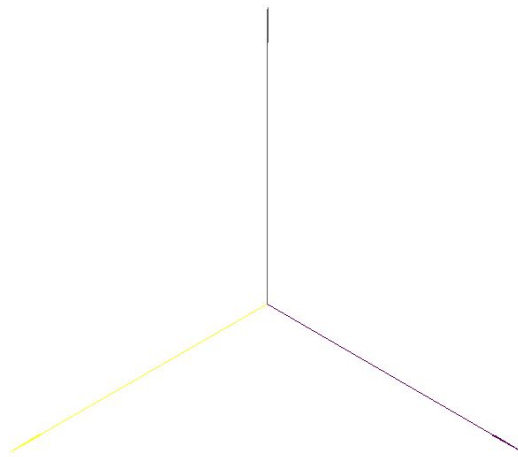


Figure 91:

Working with Materials

A Geometry like all them above is just the shape of the object. The Engine cannot render a shape without knowing anything about its surface properties. You need to apply at least a color or better a texture to the surface to make them visible. Colors and textures are represented as Material objects. You can just add a color or one picture as Texture and the implementation creates the Material Object automatically.

If you want to have more influence and want to use complex Materials with Normal maps or shiness you have to create the Material files by yourself. But it is not complicated if you understand the mechanics.

Simple Case: One Imagefile as Material

If you just want to change the “basic” look of an surface, just add an Picture (can be JPG or PNG) as Texture. You cant make any additional Settings to the Material like offset or how it is rendered on the Object. This way is only recommended for very basic objects.

```
<env:box width="0.2" height="0.2" depth="0.2" color="#FFFFFFF" texturepath="jadex3d/textures"
```

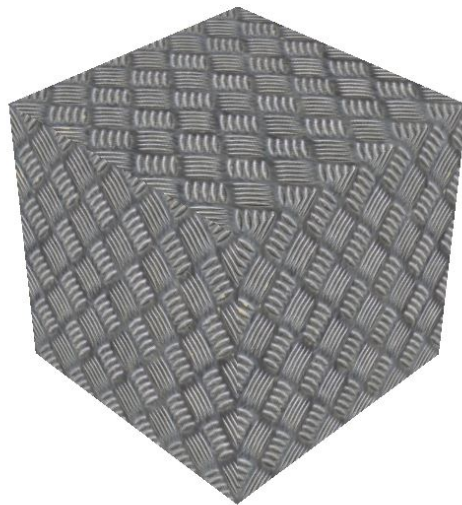


Figure 92:

Complex Case: Using Materials with .j3m Files

You can define small Textfiles with additional informations about the Materials. How this is working is perfectly described by the official JMonkey Tutorials here: [JMonkey Materials Definition](#)

In the Code you just Link to the Material File you Created.

```
<env:box width="0.2" height="0.2" depth="0.2" materialpath="jadex3d/textures/solid/Iron.j3m"
```

In this small example we use a normal map to add some structure on the object without additional polygons.

```
Material Iron : Common/MatDefs/Light/Lighting.j3md {
  MaterialParameters {
    DiffuseMap : jadex3d/textures/solid/metalfloor.jpg
    NormalMap: jadex3d/textures/solid/metalfloor_normal3.jpg
    Specular : 1.0 1.0 1.0 1.0
    Diffuse : 1.0 1.0 1.0 1.0
    Ambient : 1.0 1.0 1.0 1.0
    GlowColor : 1.0 1.0 1.0 1.0
    VertexLighting : false
    UseMaterialColors : false
  }
  AdditionalRenderState {
    FaceCull Back
  }
}
```

With this as Result (You can see the NormalMap Effect):

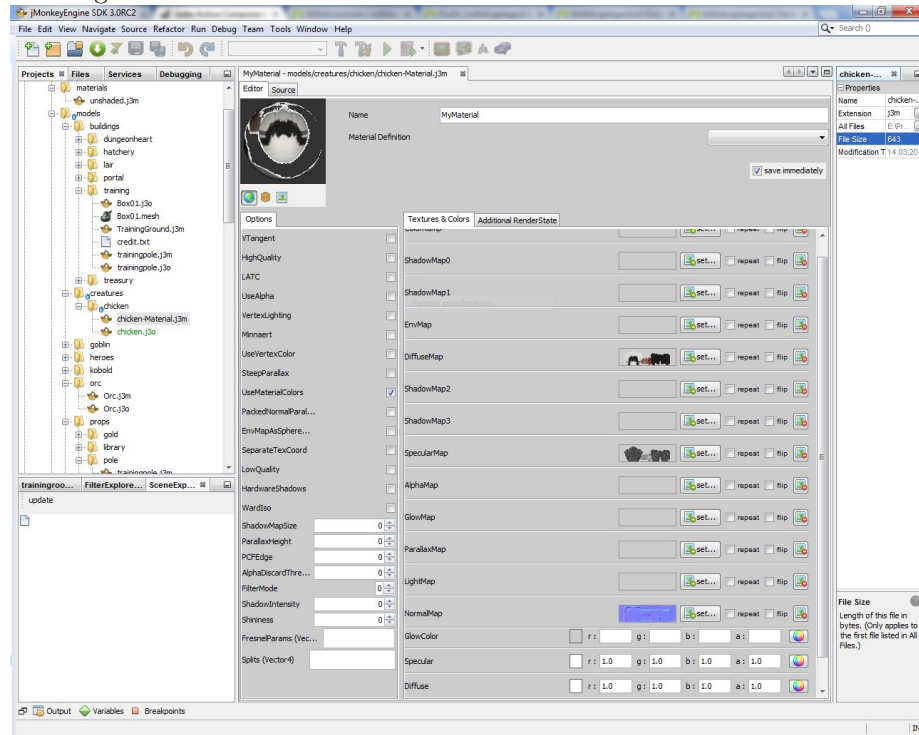


Figure 93:

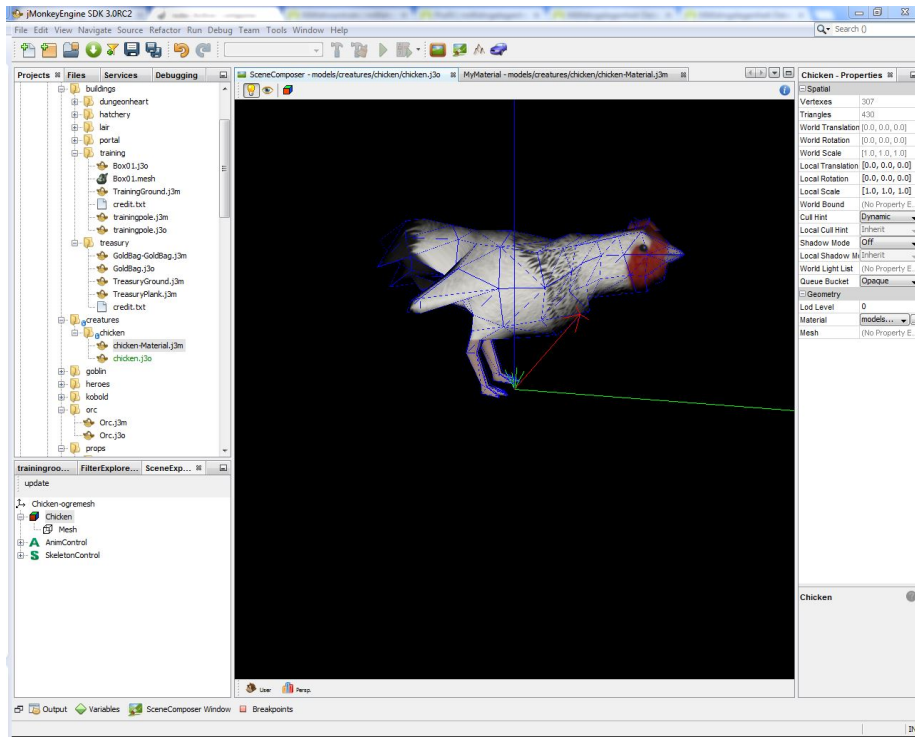
Complex Case 2: Defining and using Materials with the jMonkey SDK

If you want to use the jMonkey SDK it is pretty simple to create complex j3me Files with an Wizard. Look into the jMonkey SDK Documentation (TODO) to see how this SDK can help you alot using it along with Eclipse.

Defining the Material:



Adding it to an Asset:



Working with complex 3d Objects

todo

3D Objects in general

todo

JMonkey 3D Object Format

todo

Using 3D Objects in Jadex

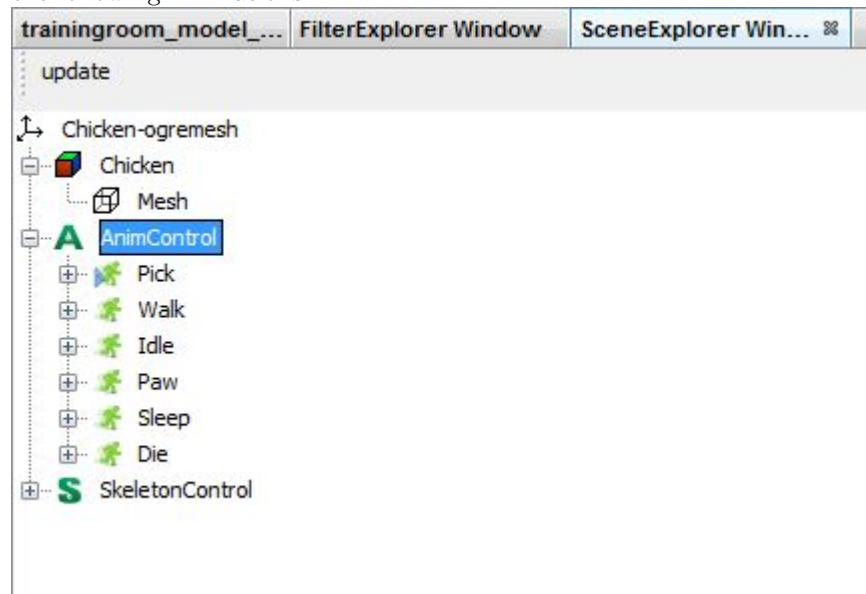
todo

Working with Animations

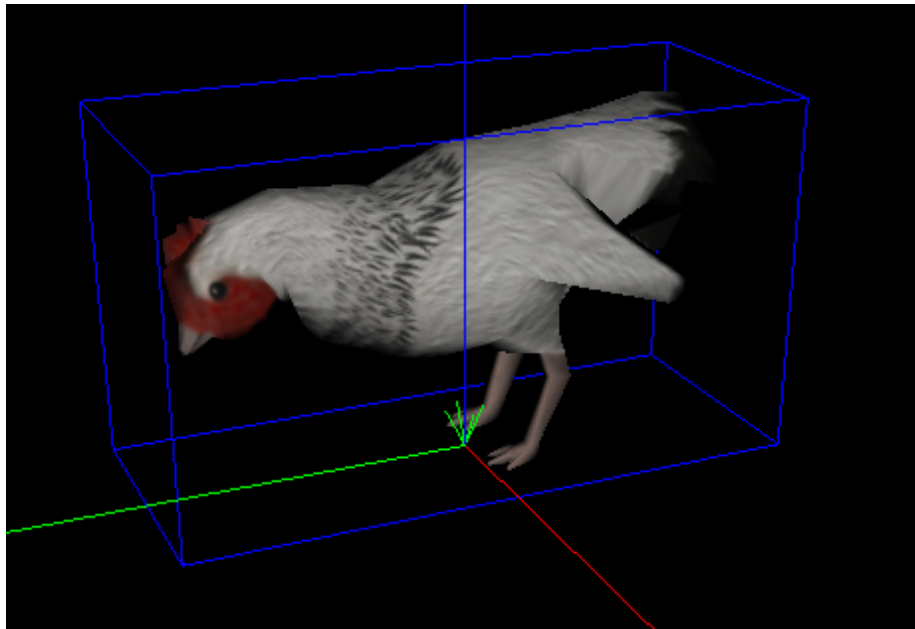
You create animated models for complex Objects with a tool such as Blender. Take some time and learn how to create your own models in Tutorials. If you downloaded an Model with Animations, the first Step is to find out the Names of the Animation the model contains.

Step 1: Finding the Animations Names

If you downloaded a Model and have it in the correct .jme Format you can open the Model in the jMonkey SDK. You can see all Animations listed in the Scene Explorer Window. Click on the Animation-Control. For the Chicken we have the following Animations:



The Chicken play the Pick Animation:



Step 2: Using the Animations

Inside an `object3d` tag you can define Animations. For example if you just want to play the “idle” Animation from the chicken all the time, you can simply write this:

```
<env:animation name="Idle" channel="chanA" loop="true" speed="1.5"/>
```

You can choose different parameters:

Name	default value	Description	Type
name	null	The Name has to be the Same from Step 1	String
channel	null	Some models can have several Animations on different Channels	String
loop	true	If you set this to false the Animation is only played once	Boolean
speed	1.0	The Playback speed of the Animation	Integer

If you want to create more complex animation-logics you have to define an `animationcondition`. The next example shows you how to use that. It uses Properties from the Environment which are used and influenced by your Java code.

```
<env:drawable3d objecttype="chicken" width="1" height="1" depth="1">
```

```

<env:object3d width="0.6" height="0.6" depth="0.6" x="0" y="0" z="0"
  modelpath="models/creatures/chicken/chicken.j3o"
  materialpath="models/creatures/chicken/chicken-Material.j3m" >

  <env:animation name="Pick" channel="chanA" loop="true" speed="1.5">
    <env:animationcondition>$object.getProperty("status").equals("Pick")</env:animationcondi
  </env:animation>

  <env:animation name="Idle" channel="chanA" loop="true" speed="1.5">
    <env:animationcondition>$object.getProperty("status").equals("Idle")</env:animationcondi
  </env:animation>

  <env:animation name="Paw" channel="chanA" loop="true" speed="1.5">
    <env:animationcondition>$object.getProperty("status").equals("Paw")</env:animationcondi
  </env:animation>

  <env:animation name="Sleep" channel="chanA" loop="true" speed="1.5">
    <env:animationcondition>$object.getProperty("status").equals("Sleeping")</env:animation
  </env:animation>

  <env:animation name="Walk" channel="chanA" loop="true" speed="2.5">
    <env:animationcondition>$object.getProperty("status").equals("Walk")</env:animationcondi
  </env:animation>

</env:object3d>
</env:drawable3d>

```

The Program sets the “status” Property. For example, if the chicken is moving, it sets the status on “Walk” and the Enviroment Triggers this status with the “Walk” animation because of the animationcondition.

This example can look like this in the 3d Enviroment, if you have lots of chickens who either walk or do randon animations.



1 Environment Instance

This section describes how an environment space instance can be created based on an environment space type definition. Space instance definitions are part of the application instance section in the application descriptor. Basically, in the instance part arbitrary many elements of the defined types can be created. In addition, it is possible to set-up evaluations, which can be used to collect and process data from the environment. In the Figure below the corresponding cutout of the XML schema is shown.

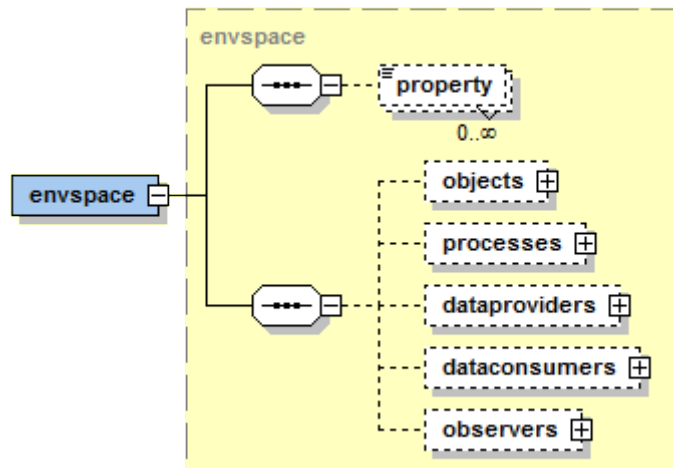


Figure 94:

~The environment instance schema definition~

An environment space has two mandatory attributes. The `~name~` is used for identifying the space instance and the `~type~` relates the environment space type that is the basis for the space instance. Additionally, the original `~width~`, `~height~` and `~depth~` attributes of the space type can be overridden at the instance level via attributes. Furthermore, space `~property~` elements can be defined for storing arbitrary user data. The additional elements - objects, processes, data providers, data consumers and observers - are described in the following sections.

1.1 Objects

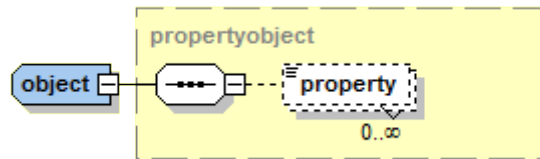


Figure 95:

~Object schema definition~

An object in an instance of a space object type. It is defined using the `~type~` attribute that must relate to an existing object type. For convenience, it is possible to specify the `~number~` attribute, which has the effect that more than one object instances are instantiated with a single object declaration. If necessary `~property~` tags can be used as usual for further object customization.

1.1 Processes

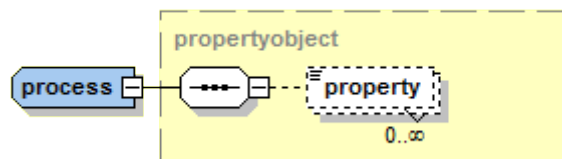


Figure 96:

~Process schema definition~

A process instance is defined very similar to an object. In the same way the `~type~` attribute has to be used to declare the underlying process type. Also properties can again be used for passing further parameter values to the process instance.

1.1 Observers

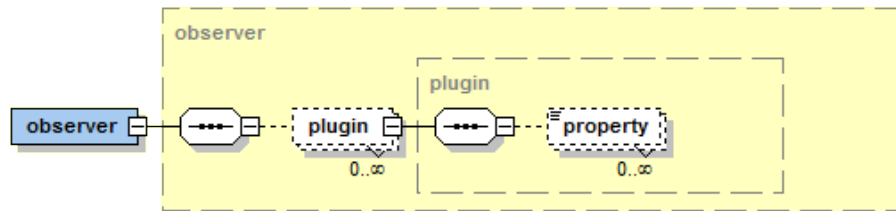


Figure 97:

~Observer schema definition~

An observer represents the user interface part of the simulation environment. It is configured using mandatory attributes for the `~name~`, `~dataview~` and `~perspective~`. Please note that the `dataview` and `perspective` attributes only represent the default settings of the observer window. The user can change these at runtime as she likes. Additionally, the `~killonexit~` flag can be used to determine, if the application should be killed when the gui is closed. The observer offers an extension mechanism that allows the user interface being modified by adding new plugins. A plugin is represented with its own button on the top left of the observer. When activated the plugin can display its own view on the control area (lower left). A plugin is defined using `~name~` and `~class~` attributes. Plugin classes have to implement the `~jadex.application.space.envsupport.observer.gui.plugin.IObserverCenterPlugin~`. The interface is shown below and basically has methods for start and shutdown as well as fetching information such as name, icon and the view to display.

```

{code:java}
package jadex.application.space.envsupport.observer.gui.plugin;

public interface IObserverCenterPlugin
{
    public void start(ObserverCenter main);

    public void shutdown();

    public String getName();

    public String getIconPath();

    public Component getView();

    public void refresh();
}
  
```

```

}
{code}
~IObserverCenterPlugin interface~

```

The evaluation of environment data is done by specifying data providers and data consumers. A data provider can be used to define which data should be collected and a data consumer then can operate on this collected data in order to process it in different ways.

1.1 Data Providers

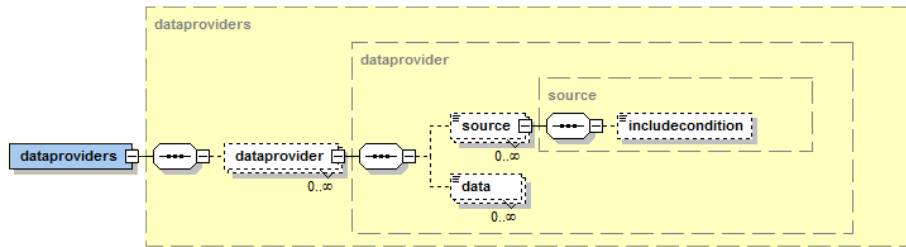


Figure 98:

~Data provider schema part~

A data provider is similar to a database query, which is executed once every step the spaceexecutor performs. This means a data provider does not store a history of entries but only provides the data that has been collected at the current point in time. In order to use a data provider from data consumers it is necessary to equip it with a specific `name` attribute. The query itself is specified using arbitrary many `source` and `data` elements.

A named `source` element determines which space objects are of general interest for the query (corresponds to the ‘from’ part of an SQL query). It requires to state, which space objects of an underlying `objecttype` should be considered. As possibly not all objects of a given type should be included an `includecondition` can be employed to explicitly state what dynamic property an object has to posses in order to be considerd. If not the object itself is of primary interest for the query but only one of its properties this can be expressed using an expression inside the tag. Using the `aggregate` flag it is possible to determine if the source data is interpreted as multiple elements (for each included object) or as one element (list of objects). The result data is calculated via a join over all sources, i.e. the cartesian product of all source elements (similar to the ‘select’ of an SQL query). An aggregated source provides always only one element to the join.

Finally, the concrete data model is specified using columns similar to a relational table model. With each `data` tag one named column is defined (`name` attribute). In the tag content an arbitrary Java expression can be specified for stating the desired property that should be recorded. The objects from the

different sources are available via predefined variables directly using the source name. In addition, the predefined variables ‘\$time’ and ‘\$tick’ are available as time series represent an often occurring use case.

1.1 Data Consumers

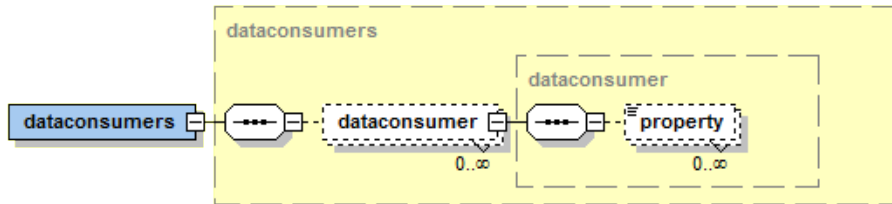


Figure 99:

~Data consumers schema part~

Data consumers can be used to process data in various ways. As data consumers may be very different, the configuration is kept very general. A data consumer is specified using the ~name~ and ~class~ attributes. The class has to extend the interface ~jadex.application.space.envsupport.evaluation.ITableDataConsumer~, which is shown below.

```
{code:java}
package jadex.application.space.envsupport.evaluation;

public interface ITableDataConsumer extends IPropertyObject
{
    public void consumeData(long time, double tick);
}
{code}
```

~Interface ITableDataConsumer~

The interface contains the method ~consumeData()~, which is automatically called in each execution step (determined by the space executor). The method then typically has the purpose to fetch the data from a (or more) connected data provider(s) and process it in a specific way, e.g. visualize it as chart. The connection to a data provider as well as other characteristics are defined using ~property~ tags. Predefined data consumers available in the distribution are TimeChartDataConsumer, XYChartDataConsumer, HistogramDataConsumer, CategoryChartDataConsumer and a CSVFileDataConsumer. In the following the XYChartDataConsumer, the HistogramDataConsumer and the CSVFileDataConsumer will be described in more details. The other consumers are very similar to the XYChartDataConsumer and only differ in the way the x-axis is interpreted.

1.1.1 XYChartDataConsumer

The xy chart consumer can be used to display arbitrary many chart series in an area. The class of this consumer is `~jadex.application.space.envsupport.evaluation.XYChartDataConsumer~`. The following table shows the properties that can be specified for a xy chart data consumer. It is based on the JFreeChart class xy line chart.

name	class	description
dataproducer	String	The name of the data provider to use.
title	String	The title of the chart consumer
labelx	String	The label for the x-axis
labeley	String	The label for the y-axis
bgimage	String	The filename of a background image.
maxitemcount	int	The maximum number of items kept. If the max is reached the oldest entries are removed.
autorepaint	boolean	True for automatic repaint on changes (possibly slow if case of frequent changes).

{table}

~Basic chart properties~

name	class	description
seriesid	String	The name of the id property. If an id is used it is a multi series definition.
seriesname	String	The name of the series.
valuex	String or IExpression (then dynamic must be true)	The property name for x value fetching or an expression for evaluation.
valuey	String or IExpression (then dynamic must be true)	The property name for y value fetching or an expression for evaluation.

{table}

~Series chart properties~

Please note that it is possible to define a `~normal series~` and a `~multi series~`. A normal series is identified by a series name, whereas a multi series consists of a multitude of series that are identified by the 'id' property. In case of a multi series for each 'id' value (fetched by the id property name) a separate chart curve is drawn.

If several normal series should be used they should follow a name scheme with appended '`_\<no\>`', e.g. `seriesname_0`, `valuex_0` and `valuey_0`, `seriesname_1` etc.

1.1.1 HistogramDataConsumer

1.1.1 CSVFileDataConsumer

Introduction

Jadex Android is a framework for developing software agents running on the Android platform. Agent-oriented Software Engineering (AOSE) is a software development paradigm especially suited for distributed Systems as the main building blocks are constituted by software agents, whose outstanding characteristics are - among others - autonomy, message-based and asynchronous communication, re- and proactivity, social abilities and cooperation, etc.



On the one hand, AOSE allows to model active objects, meaning that e.g. the user and her needs can naturally be represented by an agents, taking the responsibility to autonomously achieve the user's goals. On the other hand AOSE abstracts from lower-level details such as multithreading and communication issues, as in this respect the Jadex framework takes care of all these.

- Chapter 2 - Installation describes how to install the required tools and libraries and start development of Jadex Android applications.
- Chapter 3 - Using Jadex on Android gives an overview of available API functions

State of Implementation

Except for the UI, multiple Jadex modules were ported to android, replacing incompatible libraries and calls with ones that are compatible with android. Although we try to keep up with new jadex features, some things are still missing on android.

You can follow the release notes below to get an impression of the ongoing development, the latest implementation can be found in the download section.

Supported Modules

This is a list of all supported jadex modules as of version 2.4:

- jadex-kernel-base
- jadex-kernel-bdi

- jadex-kernel-bdiv3 (only with compile-time generation, see here)
- jadex-kernel-bpmn
- jadex-kernel-bdibpmn
- jadex-kernel-component
- jadex-kernel-micro
- jadex-platform-extension-webservice (REST client only)

Release Notes

2.5-SNAPSHOT

- mainly working on PlatformApp/ClientApp Separation

2.4

- Maven Plugin to generate BDIV3 Agent code at compile time -> enables the use of BDIV3 on Android.
- JadexAndroidEvents for dispatching events from Agent to Service/Activity
- synchronous getService() method in JadexService
- awareness set to enabled by default, as it is default on desktop platforms
- new PlatformApp/ClientApp functionality: separate Jadex Platform into a standalone app.

2.3

- API Changes! Please refer to the example project on how to start the platform.
- fixed problems with BDI Agents
- added REST client api + demo (working since 2012-11-14)
- new demo applications project
- included chat application

2.2.1

- No specific changes

2.1

- provides a simple control center application (see example project)
- since 2012-05-31: based on modular jadex distribution instead of separate artifacts (NOTE: Your Android applications will require different dependencies now!)

- adjusted version numbering to Jadex' Version
- Jadex-Android uses android xml pull parser now instead of woodstox, reduces memory footprint
- Jadex-Android stores settings in Android Shared Preferences now. It will, however, prefer properties stored in a default.settings.xml (/data/data/<application package name>/files/default.settings.xml)

0.0.5

- fixed bug, preventing service calls from Desktop-Jadex to Android-Jadex. Please note, that due to the Android emulator's virtual network environment, it is still not possible to call services offered by Desktop-Jadex. You have to manually create a ProxyAgent on the Android device to communicate. On real devices, if you are not in a private Wifi network you need to use Jadex' relay server as broadcasts are generally not supported over the Internet.

0.0.4

- updated maven projects to maven-android-plugin-3.0.0-alpha-14 -> supports ADT R15
- communication between platforms fixed, so remote mobile platform components are visible in JCC
(This requires the HTTP Relay Transport to be enabled if running in an emulator)
- added AndroidSettingsService for File Access on Android Devices
- introduced AndroidContextService to provide access to android files (TODO: properties)
- Security Service is active by default. The generated Plattform Password will be written to LogCat and saved in
/data/data/<packagename>/files/<platformname>.settings.xml.
To disable the Security Service, just uncomment the Service in your platform.component.xml

0.0.3

- uses Woodstox XML Parser instead of broken StaX reference implementation

This document will guide you through the setup that is necessary to develop applications using Jadex-Android.

Please report any difficulties or errors in this document as well as in the provided *jadex-android* libraries.

Installation

You have multiple Options:

- If you don't use eclipse or maven, just follow step 1.
- If you want to use eclipse, but not maven, follow step 1 and 2.
- If you want to use eclipse and maven, follow step 1, 2 and 3.
- To compile the example project with eclipse and maven, follow steps 1-4.
- To compile the example project with eclipse without maven follow steps 1,2 and 5.

You can also compile the example project without using eclipse by following steps 1 and 3 and then manually starting the maven build (*mvn package*).

Step 1

To develop Android applications with jadex-android you need to:

- Install the Android SDK (from <http://developer.android.com/sdk/index.html>](<http://developer.android.com/sdk/index.html>))
- Download the SDK Platform API for Android 2.2 or higher using the *Android SDK Manager*
- Download and extract *jadex-android* (Download Section)

Proceed to the next step if you want to use Eclipse OR start using jadex-android by adding the libraries in the extracted folder *lib* to your projects build path

Step 2

In order to develop Android applications using Eclipse (with or without maven), you need:

- Eclipse 3.6 or newer (<http://www.eclipse.org/downloads/>](<http://www.eclipse.org/downloads/>))
- The Android Developer Tools (ADT), Eclipse Update Site: <https://dl-ssl.google.com/android/eclipse/>](<https://dl-ssl.google.com/android/eclipse/>)
- In Eclipse, set the path to the Android SDK in *Window -> Preferences -> Android* and click *Apply*

Proceed to the next step if you wish to use maven OR to step 5 if you don't.

Step 3

If you want to use Maven as Build System (required for the included example project):

- set the `$ANDROID_HOME` environment variable to the directory where the Android SDK is located
- install a JDK (not JRE), available at <http://www.oracle.com/technetwork/java/javase/downloads/index.html>](<http://www.oracle.com/technetwork/java/javase/downloads/index.html>)
- configure Eclipse to run with a JDK (<http://matsim.org/node/372>](<http://matsim.org/node/372>)) and set a JDK as default JRE (*Window -> Preferences -> Java -> Installed JREs*, add your JDK here and check it)
- install m2eclipse plugin in eclipse (for eclipse 3.6 users, use the old update site: <http://download.eclipse.org/technology/m2e/releases>](<http://download.eclipse.org/technology/m2e/releases>))

Since Jadex 2.2 the jadex-android artifacts are available through the maven central repository. Eclipse will download these as necessary for your Jadex Android projects.

Step 4

To import the maven sample project in eclipse (after following steps 2 and 3)

- extract the *jadex-android-example-projects.zip*, which is included in the jadex-android distribution
- copy the *jadex-android-example-project-maven* directory to your workspace
- choose *File -> Import -> Maven -> Existing Maven Projects*
- select your workspace folder, select the *jadex-android-example-project-maven* directory and click *next / finish* until import is completed
- you will be prompted to install some Eclipse Plugins (*m2e android connector*) and to restart eclipse
- after restarting, the project should build without any errors
- Starting from Version 2.1, the eclipse build will work (just click *run as Android Application*), if eclipse hangs see Hints on the bottom of this page
- to build the project with maven (**required for jadex-android < 2.1**), use the included launch config *Build example project with maven*
- to run the maven build, use the included launch config *Run example project with maven*, which will deploy and run the project on any android devices plugged in or emulators running.

Step 5

To import the non-maven sample project in eclipse (after following step 1 and 2)

- extract the *jadex-android-example-projects.zip*, which is included in the jadex-android distribution
- select *File -> Import -> General -> Existing Projects into Workspace*
- locate the extracted *jadex-android-example-project* directory

- optionally check *copy projects to workspace*
- click Finish
- copy the jadex-android libraries to the *libs* directory of the project and add them to the projects build path (you could skip jadex-android-bdi, bpmn and bdibpmn as the sample only uses a MicroAgent)
- project should compile without errors
- Right-click on the example project and choose *Run As -> Android Application* to start the application.

Hints

- if Eclipse cannot find a suitable M2E connector and you already have an older version of m2e-android installed, try updating it manually using the Eclipse Installer and the following Update Site: <http://rgladwell.github.com/m2e-android/updates/> (<http://rgladwell.github.com/m2e-android/updates/>)
- if building is slow or if you get Exceptions related to memory (“Out of Heap Space”, “GC overhead limit”) during compilation, try setting *-Xms128m -Xmx1024m* in your *eclipse.ini*
- bug in m2e-android (<https://github.com/rgladwell/m2e-android/issues/31%29>) (<https://github.com/rgladwell/m2e-android/issues/31%29>)
- NIOTCP Transport doesn't work in a 2.2 emulator, see <http://code.google.com/p/android/issues/detail?id=9431> (<http://code.google.com/p/android/issues/detail?id=9431>) — **Update (Feb. 2012)**: This is now fixed, NIOTCP was now successfully tested with FroYo (Android 2.2) and Gingerbread (Android 2.3).

Using Jadex on Android

Once you have installed the necessary tools, the **jadex-android-example-project** can be helpful to get started.

This guide, however, does **not** assume you are using the example project, but instead introduces the API of Jadex for Android step by step.

To understand the basics of android application development, please take a look at <http://developer.android.com/guide/> (<http://developer.android.com/guide/>)

This guide and all included demo applications are currently using the API Level 8, which is supported in android 2.2 and above.

We assume that you created a basic Android Application to start with.

Differences to the desktop version of Jadex

While programming components is the same on the standard jadex distribution and the android version, everything else is different.

We try to list some of the differences here to avoid confusion.

- **No JCC:** First, there is no JCC (Jadex Control Center). This has a simple reason: There are no Java Swing components included in the Android runtime libraries. But, when you develop Android Applications, you'll want to have a native UI anyway.
- **UI as entry point:** In Desktop Jadex, you have Active Components that create their own UI. On Android, the entry point of an application IS the UI (e.g. an Activity). Because of this, components can never create the UI on Android.
- **The Jadex Platform runs inside an Application:** In consequence, instead of running the Jadex Platform and then starting applications, on Android, you will first start your application which then can launch a Jadex Platform. The Platform is also not shared across applications by default.
- **UI can be paused any time:** Because an Android UI Component can be paused or destroyed at any time, it is recommended to let the Jadex Platform run in an Android Service.

Start a Jadex Platform

To use Jadex components, you first have to start a Jadex Platform. This can be done in different ways, both of them are described below.

Use the Jadex Platform directly in an activity

This method is only recommended for simple applications, e.g. if you **don't care** about running in **background**.

Also, the jadex **platform will shutdown** if the application gets destroyed, for example when **turning the device**.

Skip this section if you don't like that.

If such limitations doesn't matter or you simply want to test jadex on android, you can have your main activity extend the *JadexAndroidActivity* class.

This replaces the Android Activity class and provides additional, jadex-related functionality.

The Jadex-Android-Example-Project shows how to use this method in the *jadex.android.exampleproject.simple* package.

The following code shows how to set-up a jadex platform.

```

public class HelloWorldActivity extends JadexAndroidActivity
{
public HelloWorldActivity()
{
    super();
    setPlatformKernels(JadexPlatformManager.KERNEL_MICRO);
    setPlatformAutostart(true);
    setPlatformName("HelloPlatform");
    setPlatformOptions("<insert normal jadex options here>");
}
}

```

Available platform options are documented here .

Setting up an activity like this will start a jadex-platform during the *onCreate()* phase and inform the activity about the progress in the two methods

`onPlatformStarting()`

and

`onPlatformStarted()`

:

```

@Override
protected void onPlatformStarting()
{
    super.onPlatformStarting();
    // own logic here
}

@Override
protected void onPlatformStarted(IExternalAccess result)
{
    super.onPlatformStarted(result);
    IComponentIdentifier platformId = result.getComponentIdentifier();
    // own logic here
}

```

The jadex platform will also be automatically be terminated during the *onDestroy()* phase, e.g. when the activity is terminated by the user.

If you use

`setPlatformAutostart(false)`

, the platform can be started manually by calling

`startPlatform()`

Regardless of which method is being used, the platform can always be stopped by calling

```
stopPlatforms()
```

Use the Jadex Platform in a service

If you need to create a more complex application, which should perform background tasks or should at least keep a jadex platform running in background, you should use the jadex platform in an android service .

The *Jadex-Android-Example-Project* shows how to use this method in the *jadex.android.exampleproject.extended* package.

For your service class, extend

```
JadexPlatformService
```

like this:

```
public class MyJadexService extends JadexPlatformService
```

By default, the service will autostart a jadex platform on creation.

To adjust Jadex Platform behaviour, use the methods

```
setPlatformAutostart()
```

```
,
```

```
setPlatformKernels()
```

```
,
```

```
setPlatformName()
```

and

```
setPlatformOptions
```

in the service constructor, like below:

```
public MyJadexService()
{
    setPlatformAutostart(false);
    setPlatformKernels(JadexPlatformOptions.KERNEL_MICRO);
    setPlatformName("JadexAndroidExample");
    setPlatformOptions("-awareness false");
}
```

Create Agents/Components

When you implement Agents that should run on an Android device, use the specific Classes

`AndroidMicroAgent`

, ... (available since 2.4).

These classes offer additional functionality, such as event dispatching (see next section).

A Component is started by the method

`startComponent()`

. As Parameters, you have to specify a component name and the path to it's model (xml) file. In case of MicroAgents, you can specify the Class of the Agent directly, e.g.

```
startComponent("HelloWorldAgent", MyAgent.class).addResultListener(new DefaultResultListener()
    public void resultAvailable(IComponentIdentifier result)
    {
        System.out.println("Agent started");
    }
});
```

With the Result Listener you will be informed when the Agent is created.

For more advanced scenarios, you can pass a

`CreationInfo`

object to the Agent which can contain additional parameters.

Agent <-> Android Service Coupling

The communication between Component and Service can be handled using a **ProvidedService** and **JadexAndroidEvents**.

The Combination of the two offer an easy way of agent-service coupling.

The idea is that the Android Service can invoke methods on the Agent through an interface, while the Agent can inform the Android Service about what's happening through events. Both ways are described below.

Communication with the activity will not be mentioned in this tutorial, as it is recommended to **handle all agent-based communication in a service**, since activities are pause/resumed/destroyed on a regular basis and thus are not reliable.

The image below shows an overview of the communication model:

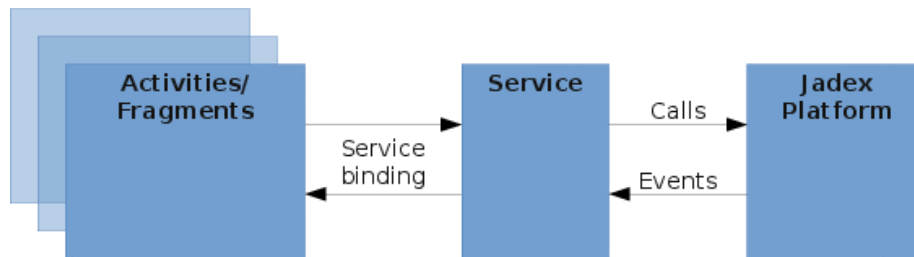


Figure 100:

Android Service to Agent

To make a Component accessible by the Android Service, first create the required interface:

```

public interface IAgentInterface
{
    void callAgent(String message);
}
  
```

Now let your Component implement that interface and add it to the Provided-
Services declaration:

```

@ProvidedServices({
    @ProvidedService(name="agentinterface", type=IAgentInterface.class)
})
@Service
public class MyAgent extends AndroidMicroAgent implements IAgentInterface
{
    public void callAgent(String message) { ... }
}
  
```

To gain access to the Component via the interface you created, use the following code in your Android **Service**:

```

IAgentInterface agent = getService(IAgentInterface.class);
agent.callAgent("Hello Agent!");
  
```

If you are using Jadex inside a JadexAndroidActivity, call `getPlatformService()` first:

```

IAgentInterface agent = getPlatformService().getService(IAgentInterface.class);
  
```

Agent to Android Service

Agent to Android Service communication is done event-based.
First, create a custom Event Type that extends

JadexAndroidEvent

:

```
public class MyEvent extends JadexAndroidEvent
{
    private String message;

    public String getMessage()
    {
        return message;
    }

    public void setMessage(String msg)
    {
        this.message = msg;
    }
}
```

For your Service to listen to Agent events, it has to register an EventReceiver
(you can do this in

`onCreate()`

):

```
registerEventReceiver(new EventReceiver<MyEvent>(MyEvent.class)
{
    @Override
    public void receiveEvent(final MyEvent event)
    {
        handler.post(new Runnable()
        {

            @Override
            public void run()
            {
                System.out.println("received message: " + event.getMessage());
            }
        });
    }
});
```

To dispatch events in an Agent, just use the

```
dispatchEvent()
```

method (you need to extend `AndroidMicroAgent` for this method):

```
MyEvent myEvent = new MyEvent();
myEvent.setMessage("Hello Service!");
dispatchEvent(myEvent);
```

Starting Components

To simply start a (bdi, bpmn, ...) component, use

```
startComponent()
```

in your Activity or Service and provide a component name and a valid model path:

```
startComponent("HelloWorldAgent", "jadex/android/applications/demos/bdi/HelloWorld.agent.xml")
    .addResultListener(bdiCreatedResultListener);
```

For micro agents, please use

```
startMicroAgent()
```

and provide the class of the agent.

Accessing the platform

The provided methods for accessing the platform depend on whether you are using `JadexAndroidActivity` (running Jadex inside an activity) or `JadexPlatformService` (running Jadex inside a service).

You can, however, get the internal platform service from a `JadexAndroidActivity` by calling `getPlatformService()`. The returned object should contain all methods listed here.

- `isPlatformRunning()`: Checks whether the platform is running or not.
- `getService()`: Gets a service of a component running on the platform (asynchronously).
- `getService()`: As before, but blocks until the service is found (synchronously).
- `startComponent()`: Starts a component on the platform.
- `shutdownJadexPlatform()`: Terminates the platform.

If you want to access the platform manually, for features not covered by provided methods, you can use the following methods:

- `getPlatformAccess()`: returns the external access to the platform

Using remote services

Jadex was designed for distributed systems and Jadex-Android supports all the distribution features, too.

To use a remote service, just declare the required service like usual in the agent:

```
@RequiredServices(  
    {@RequiredService(name = "myservice", type = IMyService.class, binding = @Binding(scope =
```

Be sure to use the same Interface on both the service consuming and the service providing application, e.g. *use the same package and class name* for the service interface.

If Binding scope is set to global, services running on a desktop platform will be discovered by android devices, too.

Using the jadex control center

Jadex-android provides a simple replacement for the Desktop-JCC to configure security and awareness settings:

AwarenessManagement

Delete On Disappearance

Discovery Settings

Info send delay
Current value: 20

Fast startup awareness

Discovery Mechanisms

Includes/Excludes

vsispool31
included

Discovery Info

vsispool31_25d
Last info: 18-12-2012 14:04:59
Has Proxy

ISecurityService

Local Password Settings

Use Password

To use it, your AndroidManifest.xml has to include the following line:

```
<activity android:name="jadex.android.controlcenter.JadexAndroidControlCenter"/>
```

And *jadex-runtimetools-android-<version>.jar* should be included in your project build path.

The Control Center can then be launched from any jadex-android application as follows:

```
Intent i = new Intent(this, JadexAndroidControlCenter.class);  
i.putExtra("platformId", (ComponentIdentifier) platformId);  
startActivity(i);
```

This is also part of the example-project.

Using BDIv3 on Android

The BDI v3 programming model heavily depends on code-generation based on java annotations.

It's using the ASM Bytecode manipulation framework to generate the code.

Android not only uses a different virtual machine than any Java SE environment, the Dalvik Virtual Machine (DVM), it also uses a different bytecode representation, which is not supported by ASM.

As runtime bytecode generation is slow on android anyway, we use a different mechanism to make BDIv3 work: Generating the bytecode at compile time, which is then processed and converted by standard android tools.

For this method to work, however, we need to include an additional compile step, which is done using maven.

So you **need to setup a maven project to use BDIv3** Components!

Additionally, you need to have a copy of the jadex-android-maven-plugin, so this is the first step

jadex-android-maven-plugin

The simplest way to get the plugin is to insert the following code in your pom.xml, which adds my jadex repository:

```
<repositories>  
  <repository>  
    <id>jadex-android</id>  
    <url>http://jadex.julakali.org/nexus/content/groups/public/</url>
```



```
</repository>
</repositories>
```

Then, in the build section of your pom, define that you want to use the plugin:

```
<build>
  <plugins>
    <plugin>
      <groupId>net.sourceforge.jadex</groupId>
      <artifactId>jadex-android-maven-plugin</artifactId>
      <executions>
        <execution>
          <goals>
            <goal>generateBDI</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

If you compile your project with maven now, the plugin will run, detect all BDIV3 Agents and will enhance the classes as needed by the Jadex runtime.

On desktop systems, Jadex provides a platform for other (user) applications. Those applications can be started using the JCC and don't contain any Jadex libraries, as those are provided by the platform.

On Android devices, however, it's not simple to separate the user application from the Jadex platform, because of the way Android deals with loading Activities and Services.

So past jadex-android versions just included all needed platform libraries in the packaged user-application.

Nevertheless, the mentioned separation is desirable, because if user applications don't contain the whole set of jadex platform libraries, there will be some advantages:

faster application packaging time: always dex-ing the jadex libraries is quite time-intensive

smaller applications: because user applications doesn't contain the jadex platform

Since version 2.4 (which is currently only available as night build) of jadex-android, this separation is supported.

The Platform and all it's libraries are now packaged into a standalone Android APK .

This document describes how to develop applications using the *ClientApp* model.

Installation of the Platform App

1. Download the Jadex platformApp from nightly builds .
2. Install it on your phone or emulator. For emulator, use `adb install jadex-android-platformapp-2.4-SNAPSHOT.apk`.
For installation on your phone, enable the *unknown sources* setting, located in *settings > security* , download the APK to your phone and execute it.

The Platform App will create a Startup icon just like any other android app. However, selecting this icon will not start the platform, instead, it is started by launching client applications.

Functionality of separated Platform and Client

When a client application is started, the following steps are performed as illustrated in the image below:

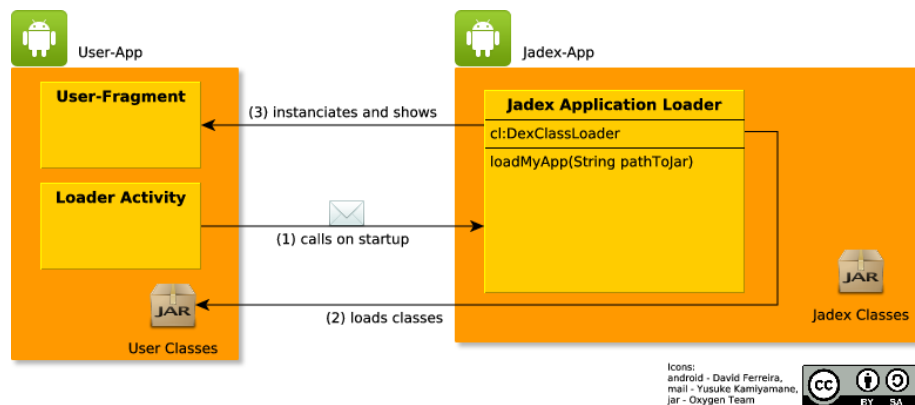


Figure 101:

1. The started user application or rather the included Loader Activity will call the Jadex Platform application by posting an intent.
2. The Jadex Platform application parses the intent data, which include the name of the class the User App wants to display on the screen, and loads the user classes.
3. The User Fragment will be instantiated and shown on the screen, while all jadex classes are present and can be used by the user application.

The next step will explain how to create client applications.

Developing a Jadex ClientApp

Prerequisites

Important: Currently, it is only possible to develop Jadex ClientApps using maven!

Before you can start developing, please follow steps 1-3 from here to install Eclipse and the Android SDK.

Using the ClientApp example

For a quick start, use the example project *jadex-android-clientapp-example-maven* included in the *jadex-android-example-projects* file.

- extract the *jadex-android-example-projects.zip*, which is included in the *jadex-android* distribution
- copy the *jadex-android-clientapp-example-maven* directory to your workspace
- choose *File -> Import -> Maven -> Existing Maven Projects*
- select your workspace folder, select the *jadex-android-clientapp-example-maven* directory and click *next / finish* until import is completed
- to build the project with maven **which is required as of jadex-android-2.4**, use the included launch config *Build clientapp example*
- to run the maven build, use the included launch config *Run clientapp example* which will deploy and run the project on any android devices plugged in or emulators running.

The ClientApp

In Contrast to a normal android application, each Jadex ClientApp has an entry point (Activity) that **has** to extend *JadexApplication*. This Class only needs to overwrite one Method, *getClassName()*.

The String it returns should be the fully-qualified class name of a *ClientAppFragment*.

ClientApp Fragments

Because Fragments can be added dynamically to views, we use Fragments to display the developers application content.

That means, instead of creating *Activity* classes, you should instead extend from *ClientAppFragment*, which offers mostly the same functionality.

This is the biggest difference to normal application development: **No Activities, use ClientAppFragments!**

If access to the concrete Activity is needed, call *getActivity()* in your Fragment.

We also added one Lifecycle Phase: *onPrepare()*. This is called **before** any other android lifecycle methods are called, so nothing is initialized. It is needed to perform tasks that have to execute *before* the Fragment enters the view, such as requesting Window Features.

Other Features work like before: binding Services, starting other Activities/Fragments.

Chapter 1 - Introduction

This tool guide gives an overview about the Jadex tool suite and explains how they can be used to administrate, manage, and debug your infrastructure and applications.

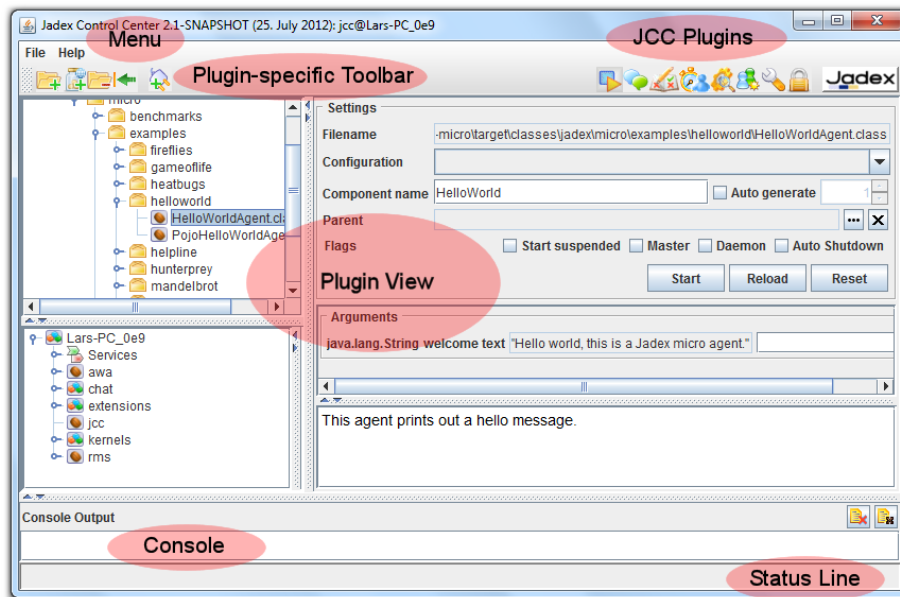
- Chapter 02 JCC Overview [(02%20JCC%20Overview)] : Getting around the Jadex Control Center (JCC) window
- Chapter 03 Starter [(03%20Starter)] : A JCC view for starting and stopping components on the platform
- Chapter 04 Awareness Settings [(04%20Awareness%20Settings)] : A JCC view for administering automatic platform discovery features
- Chapter 05 Security Settings [(05%20Security%20Settings%20)] : A JCC view for specifying local and remote platform passwords and trusted networks
- Chapter 06 Library Center [(06%20Library%20Center)] : A JCC view for adding and removing Java resources (i.e. Jar files or Maven artifacts)
- Chapter 07 Deployer [(07%20Deployer)] : A JCC view for remote file system access
- Chapter 08 Chat [(08%20Chat)] : An application that allows worldwide chatting and exchanging files with Jadex users
- Chapter 09 Simulation Control [(09%20Simulation%20Control)] : A JCC view for switching execution between continuous and simulation clocks
- Chapter 10 Component Viewer [(10%20Component%20Viewer)] : A JCC view for showing custom user interfaces of components and services
- Chapter 11 Test Center [(11%20Test%20Center)] : A JCC view for executing component unit tests.
- Chapter 12 Debugger [(12%20Debugger)] : A JCC view for inspecting components and step-wise component execution
- Chapter 13 Cli Email Signer [(13%20Cli%20Email%20Signer)] : A JCC view for inspecting components and step-wise component execution
- Chapter 14 Relay Server [(14%20Relay%20Server)] : A rendezvous server for cross-network platform discovery and communication

- Chapter 15 ADF Checker](15%20ADF%20Checker) : An eclipse plugin for consistency checking of component descriptions

Legacy tools that are still supported but whose applicability is partially reduced due to the new active component approach:

- Chapter A1 Conversation Center](A1%20Conversation%20Center) : A JCC view for manual sending and receiving of component messages
- Chapter A2 Communication Analyzer](A2%20Communication%20Analyzer) : A JCC view for monitoring message exchange between components
- Chapter A3 Directory Facilitator](A3%20Directory%20Facilitator) : A JCC view for old-fashioned service registrations

Chapter 2 - Jadex Control Center Overview



JCC Overview

The Jadex Control Center (JCC) is the heart of most Swing based runtime tools. The JCC itself uses an extensible plugin-based mechanism to offer different functionalities to the user. As can be seen in the screenshot above it consists of different areas that are explained in the following:

Menu: In the menu general settings as well as plugin-specific ones can be found, i.e. the menu may change when the plugin view is changed. The general menu has a 'File' and a 'Help' section. In the first the platform and user interface settings can be saved or restored. Furthermore, in the help section a link to the Jadex online documentation can be found.

Toolbar: Similar to the menu also the toolbar consists of a plugin-specific (left-hand side) and a general part (right-hand side). The general part consists of icons for all currently loaded plugging. By clicking on another icon the corresponding plugin will be activated and show in the lower area. By right clicking on a free space in the toolbar a popup menu can be activated that allows for adding new plugins at runtime. A plugin chooser allows for selecting the plugin to load. After successful initialization a new icon will be shown for the plugin at the right hand side of the toolbar. In addition to adding new plugging, also existing ones can be hidden or their ordering in the toolbar can be changed. For this purpose rightclick on a plugin icon. A popup menu will appear that allows you to move the icon to the left or right or hide it completely. Hidden plugins can be made visible again just by right clicking on a free space in the toolbar and selecting the hidden tool in the popup menu.

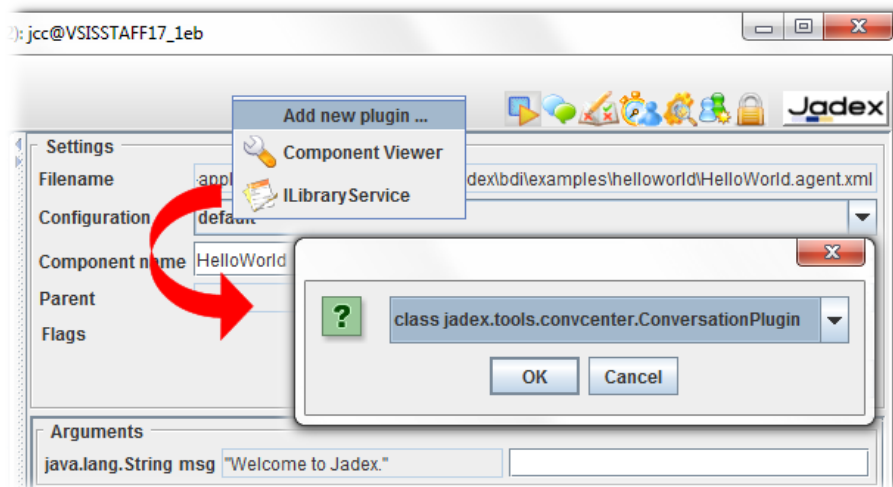


Figure 102:














Adding a tool plugin via popup menu on toolbar

Plugin: The main plugin view in the middle of the JCC show the user interface of the currently selected plugin.

Console: The console can be used to display the Java System.out and System.err streams of the platform. Especially, when administrating remote platforms it can be used to make visible the print outs of the otherwise inaccessible streams.

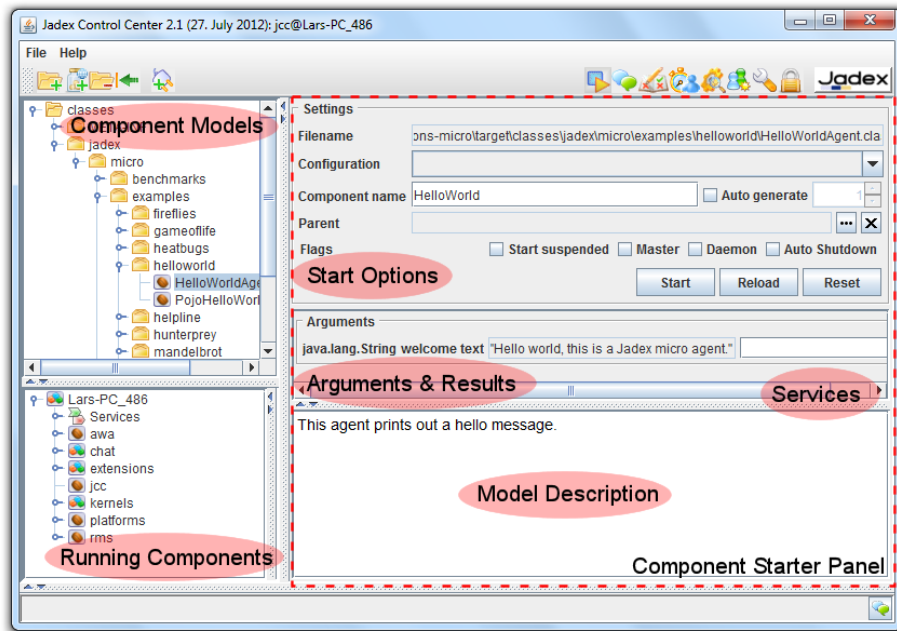
Status line: The status line is used by the plugins and the JCC itself to announce recent events to the user. The messages will be displayed for several seconds and vanish automatically. In additiona, on the right hand side of the status line current processing activities may be visualized.

Overview of Plugins

- : The **starter plugin** for starting and stopping applications and components.
- : The **awareness plugin** that can be used to configure awareness options such as in- or excluded nodes.
- : The **security plugin** allows for configuring platform security settings.
- : The **library plugin** can be used to add custom resources (jars or directories) to the Java classpath.
- : The **deployer plugin** is similar to a ftp tool and can be used to transfer files from/to other platforms.
- : The **chat plugin** can be used to chat with users from other platforms.
- : With the **simulation plugin** the execution mode and time progress can be controlled.
- : The **component viewer** can be used to view components and services that expose user interfaces.
- : The **test center** is a front end for executing junit like component based test cases.
- : The **debugger** allows for executing components in step mode and inspecting their state.
- : The **conversation center** can be used to manually send messages to specific components.
- : The **communication analyzer** is helpful for tracking message based communication between components.
-  (deprecated): The **directory facilitator GUI** can be used to inspect the state of the DF and manually add or remove subscriptions.


Chapter 3 - Starter

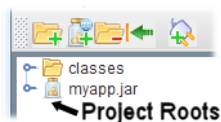
The starter plugin is one of the core tools for managing applications. It allows for starting and stopping applications and components and also provides a view of the currently running components. The basic layout of the starter is shown in the screenshot below. It basically consists of three different areas. The left upper part contains a file view of component models. When a user selects a new model it will be loaded and displayed on the right hand side of the tool. Finally, in the lower left part the platform with the currently running components is displayed.



Starter screenshot




Component Models

In the component models section a file system view on available active component models is displayed. To be able to view and select your own models here, you will first have to add the root path or jar file of your project. This can be done by clicking the add resource button  from the menu bar or by right clicking within a free space of the panel. This will also give you a popup menu with an add resource option. Activating add resource will give you a file chooser to select the project folder or jar file to add. After having confirmed the choice the folder will be displayed in the panel. You can now browse its contents and select a model to start. Please note that you cannot add any folder to the starter, as it has to be the classpath root of the contained resources, i.e. you should always add a bin or classes folder but not an internal package directly. Jadex will add the new resource automatically as classpath entry and if its not the correct folder you may encounter `ClassNotFoundException` when selecting models.



Root entries are classpath entries

In order to remove an unwanted entry from the model tree you have to select


the topmost entry (e.g. the classes directory) and use the remove resource action  from the toolbar or popup menu. For easy navigation in the model tree you can collapse the whole tree by clicking on the corresponding action  from the toolbar. The add global resource action  can be used to add a maven resource via its artifact id. Clicking the action opens a dialog in which you can browse maven repositories and search for specific artifacts. Having selected an artifact, Jadex will automatically download the resource including all its dependencies and make it available in the model tree.


Note: The add global resource action  is only available in the Jadex Pro version.


Active Component Types


The model tree as well as the component instance tree (lower left) use different icons to display different component types. This makes it easy to quickly see which internal architecture is used by a component. The following component types are currently supported:

: The basic **component type** is defined using an XML schema. Such components do not have an explicit internal architecture (simple steps can be defined).


: The **application type** is rather equivalent to the XML component type described above. The application type is considered as deprecated and will probably be removed in future releases.

: The **micro agent type** is defined using annotated Java class files. Micro agents have a three-phased internal architecture consisting of an init, a body and a shutdown phase.

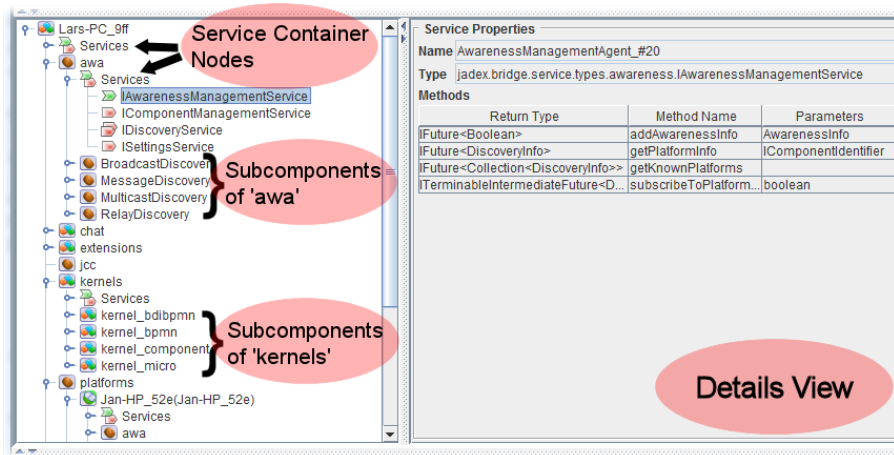
: The **BDI agent type** is defined using both XML and Java class files. BDI agents have a cognitive internal architecture that allows for constructing agents with belief, goals, and plans.

: The **BDI capability type** is a module concept for BDI agents. These modules cannot be started separately.

: The **BPMN process type** is defined using the business process modelling notation. BPMN components are modelled using a graphical BPMN tool.

: The **GPMN process type** is a goal oriented workflow variant. It combines a goal view with a BPMN activity perspective. GPMNs are thus defined using a graphical editor and a BPMN editor.

Running Components



Component instance tree

In this tree view the platform itself is displayed as root (here the platform has the name `Lars-PC_9ff`). The children of a component may consist of two node types: a service container node and further child component nodes. Looking at the ‘`awa`’ component you can e.g. see that it has a service container node and four components as direct children named ‘`BroadcastDiscovery`’, ‘`MessageDiscovery`’, ‘`MulticastDiscovery`’ and `RelayDiscovery`’. By looking at the icon of these child components you can also see that they are micro agents. The opened service container node of ‘`awa`’ reveals that this component has one provided service of type `IAwarenessManagementService` and three required services with interfaces `IComponentManagementService`, `IDiscoveryService`, and `ISettingsService`. Please note that the required service `IDiscoveryService` has a multiplicity, i.e. this required service binds all service instances of the given type. On the right hand side of the tree panel a details view is provided. Double clicking on a node in the component instance tree will activate the details view and present more information about this node. In the screenshot, details of the provided service `IAwarenessManagementService` are shown, i.e. the signatures of the methods comprising the service interface. If you select a component instead of a service the details view will show more information about the component including such as its current state or the creator.

Component Starter Panel

Having selected a component model from the model tree, Jadex will attempt to load the file and display details about it on the right hand side of the screen. In the upper part the **start options** are displayed. Below, an optional part is shown containing the properties for component **arguments and results**. In

the panel at the bottom the **model description** of the component is presented.

Start Options

Filename: The filename of the selected model, i.e. it will show the physical location of the model on the hard drive.

Configuration: The configuration in which the component should be started. Configurations are named property settings declared within a component. A configuration e.g. describes with which subcomponents a component should be started.

Component name: The component instance name. This name has to be unique, i.e. there must not exist any component with the same name within the platform. This means, if you start a component two times without changing the instance name, you will encounter an error message stating that the component could not be created because it already exists. If you don't care about the instance name or intend starting multiple components you can select the auto generate option and set the number of components to create.

Parent: The optional parent component. Using the '...' button a component selector can be brought up and a parent component can be chosen by clicking the corresponding component. Using the 'x' button a previous selection can be deleted.

Flags:

- **Start suspended:** If selected, the component and all its children will be started in suspended mode, i.e. the component will not start executing unless it is resumed. This option is helpful e.g. for debugging purposes, because it allows for starting suspended, switching to the debugger and executing a component piecewise without having to stop it manually.
- **Master:** If a component is set to be master, it is a mandatory component for the component it is contained in. If this master component is destroyed the super component will be destroyed as well.
- **Daemon:** If a component is set to daemon it will not prevent autoshtutdown of the super component (given the super component has autoshtutdown turned on).
- **Auto shutdown:** If enabled, this component is tracked to be autoshtutdowned if it has no more child components. In order to allow autoshtutdown even if there specific child components still exist, these components can be set to be daemons.

Buttons:

- **Start:** Will attempt to start the selected component type.
- **Reload:** Will reload the currently selected model.
- **Reset:** Will clear the selection.

Arguments and Results

Figure 103:

In the arguments and results section it is shown which arguments can be provided and which results are produced by the component. Each argument type is presented with its *type*, its *name* and its *default value*, which is used if nothing else is specified. In the example shown above, the type of the argument is *int*, the name is *testcnt* and the default value is *2*. On the right hand side of each argument a text field is displayed, which can be used to enter a new argument value. It has to be noted that the string entered in this text field is treated as a Java expression, i.e. it is evaluated. In the screenshot above a user entered *test*, which cannot be parsed as Java expression. If such a wrong value is entered it will automatically be detected and indicated by a red underline (as in the screenshot).

In the results section below similar details about the results are shown. For each result a row with the result type, its name, the default value and the real result is shown. As results are valid only after termination of the component, the right most text field will be empty for the model. In order to see results of component instances, it is necessary to activate the **store results** checkbox. After the component has been destroyed the 'select component instance' choice can be used to select a specific component. Its result values are shown in the results section from above. Please note that result values are not automatically deleted if 'store results' is turned on. This can be achieved manually by clicking the 'clear results' button.

Provided and Required Services

Required Services		
Name	Interface	Multiple
clockservice	IClockService	false
regservice	IRegistryServiceE3	false
chatservices	IChatService	true

Provided Services	
Interface	Creation Expression
IChatService	jadex.micro.tutorial.ChatServiceD5

Figure 104:

In this area details about the required and provided services of a component are depicted. For example, in the screenshot above, a component with three required and one provided service is shown. For each required service its *logical (local) name*, its *interface type* and its *multiplicity* are stated. In the example, a required service with name `clockservice`, interface `ILockService` and no multiplicity exists. In case of provided services only the *interface type* and the *creation expression* are provided. The creation expression either is a Java expression such as e.g. a constructor call to the service implementation or in most simple cases just the class name of the service implementation class. In the example a provided service of type `IChatService` is listed, which is created via its implementation class `jadex.micro.tutorial.ChatServiceD5`.

Model Description

The model description area contains the developer made description of a component model. The description may be simple text or HTML text. HTML may be used to improve the readability of the text due to the possibilities of using headings and different style attributes. The model description of a component model is the topmost comment in case of XML files or comes from the description annotation in case of Java files (Java comments are not accessible from class files).

Chapter 4 - Awareness Settings

Awareness is the mechanism used in Jadex to automatically discover other platforms. The awareness settings allow for customizing how awareness is realized and which other platforms one wants to ignore. The general mechanism how awareness works is described in the corresponding tool guide chapter . The main idea is to represent remote platforms as local components (called proxy components). Thus, for the platform there is no specific mechanism to search for remote services. Instead these remote services are discovered automatically if a proxy component of that platform exists locally and the search reaches the proxy. The awareness component automatically creates platform proxy components if it is notified by one of its awareness mechanisms that such a platform exists. Hence, platforms participating in awareness regularly send awareness infos over the network. These infos include the sender id of the platform as well as a lease time that tells the receiver the validity of the received packet. If this time passes by and the lease has not been renewed by the platform the proxy will be automatically deleted (assuming some buffer time).

The awareness settings contain the following sections: **Proxy Settings**, **Discovery Settings**, **Discovery Mechanisms**, **Includes and Excludes**, and **Discovery Infos**. The meaning of these settings is explained in the following.

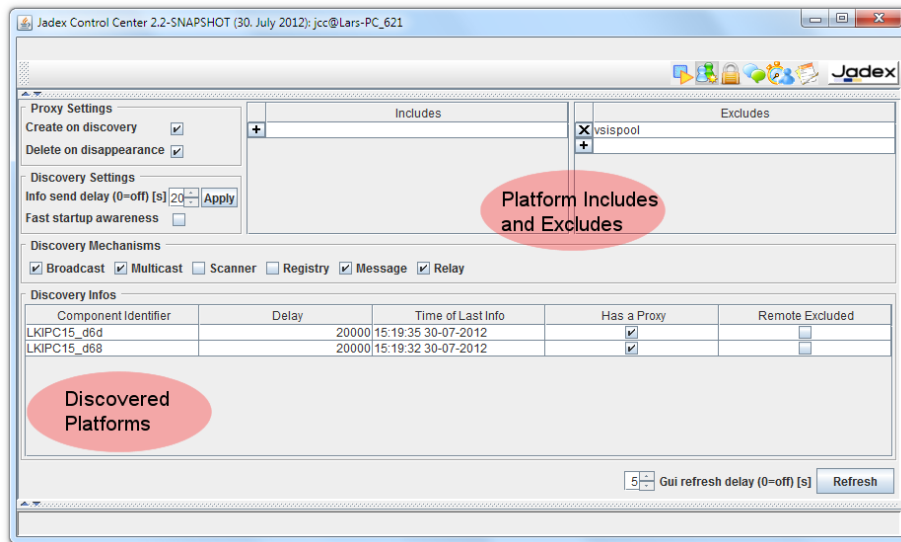


Figure 105:

Proxy Settings

The proxy settings determine how the awareness management component should behave when new information becomes available. Using **create on discovery** one can control if a new proxy component should be created in case a new platform was discovered. If this feature is turned off, no new proxies will be generated any more. The **delete on disappearance** option can be used to determine if proxies are automatically deleted when contact to the remote platform has been lost.

Discovery Settings

The discovery settings allow for changing the announcement properties of the own platform. The **info send delay** can be used to specify the delay between platform announcements sent by the active awareness mechanisms (in seconds). The delay defined here will be set in all running awareness mechanisms. The **fast startup awareness** option can be used to turn on / off a specific protocol on platform startup. If fast awareness is on, the platform will initiate an immediate awareness discovery to find other platforms as fast as possible.

Discovery Mechanisms

Discovery of other Jadex platforms is difficult to achieve as it depends on details of the network configuration, e.g. if multicast is allowed in a local network. In order to provide a robust discovery experience for Jadex applications, multiple awareness mechanisms may work concurrently in order to discover as much as possible from the surrounding environment. In general, Jadex supports two kinds of techniques: a) local mechanisms that work only in LANs and b) global mechanisms which allow for Internet scale discovery. To the first category belong *broadcast*, *multicast*, and *scanner* and to the latter belong *registry*, *message* and *relay*. Details about the mechanisms can be found in the awareness chapter of user guide. The checkboxes can be used observe which mechanisms are activated and can additionally be used to turn them on or off.

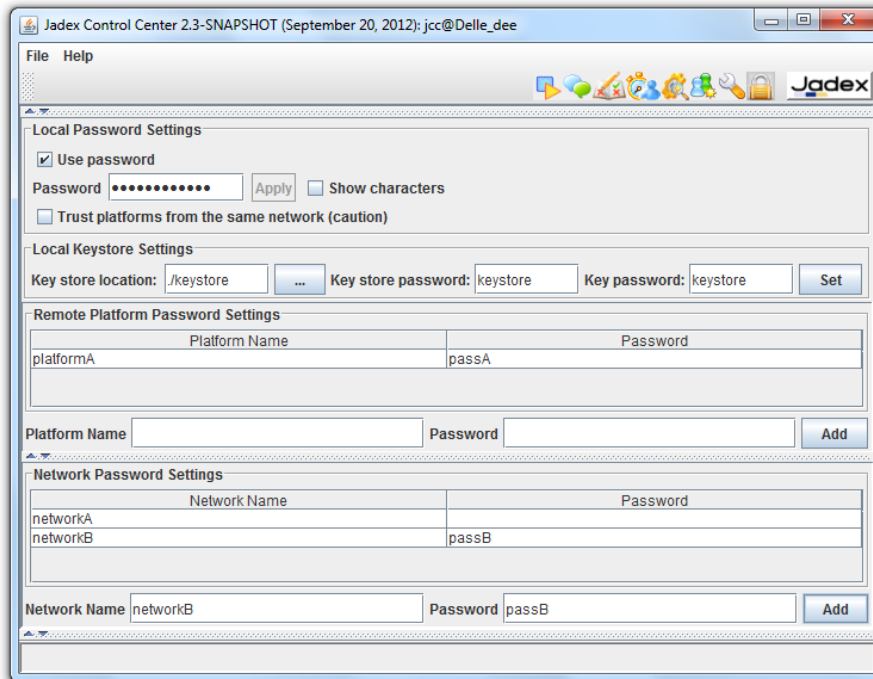
Includes and Excludes

The includes and excludes list allow for customizing the general awareness behavior in the context of specific platforms. Adding a platform name to the **exclude list** results in ignoring all discovered platforms that *start with a string* of a exclude list entry. E.g. if 'otago' is put to the exclude list and 'otago' or 'otagoBa16' is discovered, neither for the first nor for the second entry a proxy will be created. Instead of the logical platform name also IP addresses of hosts can be added to the exclude list. In this case all platforms that have a transport address containing the banned IP will be ignored. On the other hand using the **include list**, one can define platform names and IP addresses for which proxy creation is allowed, i.e. if the include list is not empty and a newly discovered platform name or IP is not present in the include list, no proxy will be created. In this respect the exclude list is a blacklist and the include list a whitelist mechanism. If you combine both lists, a newly discovered entry has to satisfy both, i.e. must be contained in the include list and must not be contained in the exclude list to not being ignored.

Discovery Infos

The table of discovery infos contains all currently valid awareness infos received from other platforms. The table shows for each entry the **component identifier** of the platform that sent the awareness info, its **delay** in sending awareness infos, the **local time of the last received info** from that source and additionally, if a **proxy is available** for the platform and if it is **remote excluded**. In the screenshot it can be seen that a platform with id 'LKIPC15_d6d' sends awareness infos every 20 seconds. The last info was received at 15:19:35 on 30-07-2012 and it current has a proxy and is not excluded.

Chapter 5 - Security Settings



Security settings panel

Platform security in Jadex is based on platform passwords and networks. Please refer to the security chapter in the user guide to read about these concepts. The security settings panel consists of four different areas that are explained in the following sections.

Local Password Settings

Use password: The use password setting turns on or off local password protection. If turned off any other platform potentially has full access towards all services of the unprotected platform.

Password: The password textfield and can be used to view (if show characters is turned on) or change the current password. To change the password the new one has to be entered and the 'Apply' button has to be pressed.

Trust platforms from the same network: Activating this option inspects the current network of the host and tries to find out the network IP name, i.e. the network prefix length is retrieved and the computer specific part is deleted. For example the host has the IP 134.100.11.77 and the prefix length is 24 (cf. figure

below). In this case a class C network is assumed and 134.100.11.0 is added as network name (without password). As all computers within the same physical network will share this network identifier, with this option platforms can be made easily talk to each other. Please note that this option should only be used for testing purposes. Attackers that guess or get knowledge about the used network name can intrude the network. To prevent this you should always equip the network names with a password. Please remember, for an attacker it is sufficient to know one unprotected network name (i.e. one network name without password) to get access to the platforms.

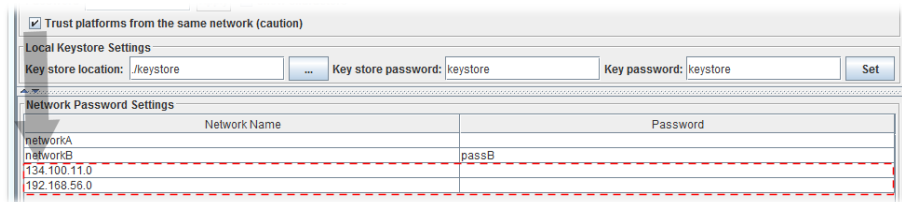


Figure 106: 05 Security Settings @network.png

Local Keystore Settings

The local keystore settings are used for certificate handling of the platform which are currently used by the SSL transport only (only part of Jadex Pro version). This is important if you e.g. want Jadex to use a specific certificate already available. In this case you can tell Jadex to use the local keystore from the location provided. The keyword to the store is needed to access the store and the keyword to a password is required to let Jadex automatically generate a self-signed certificate if none is provided.

Note: Currently, SSL transport accepts self-signed certificates so that no strong form of platform authentication is enforced. This may change in future versions.

Remote Platform Password Settings

The communication with password protected platforms is possible if you know the password of that remote platform. In the remote platform password table the passwords of remote platforms can be entered by writing the platform name and password in the corresponding input fields and pressing the 'Add' button. Please note that it is also possible and possibly more convenient to add the password of a remote platform directly within the component instance tree by activating the popup menu on a proxy component node and choosing 'Set/change remote platform password' (as shown below). To delete an entry just activate the popup menu on the table row and choose 'Remove entry' from the popup menu.

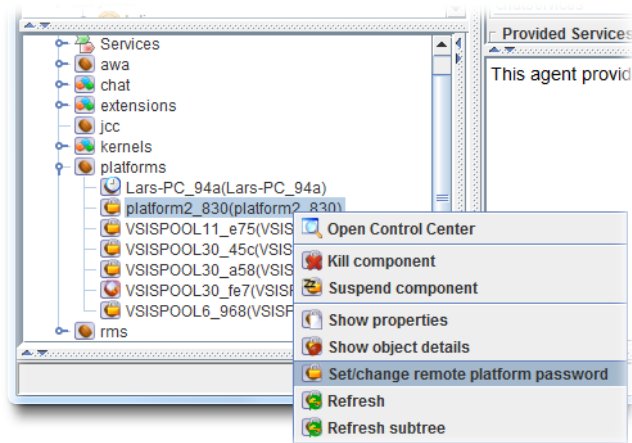


Figure 107: 05 Security Settings @set_password.png

Setting the password of a remote platform

Network Password Settings

Entering passwords of individual platforms becomes quickly tedious when the number of platforms increases, because if the network should consist of n platforms, in each platform $n-1$ passwords have to be deposited. In this case, it is more convenient to use a network name and password combination that all platforms of the network share. In this case only entry has to be made in each of the n platforms. Entering a network name and password can be done by using the input fields and pressing the 'Add' button. Removal can again be achieved by using the popup menu (as shown below).

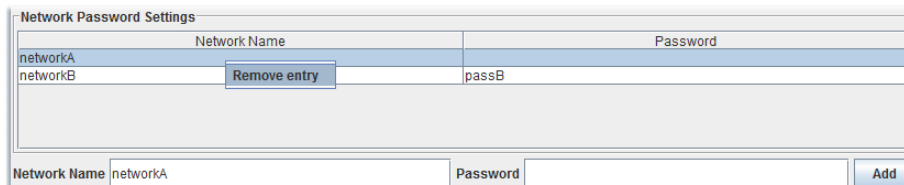


Figure 108: 05 Security Settings @network_rem.png

Chapter 6 - Library Center

Library center screenshot

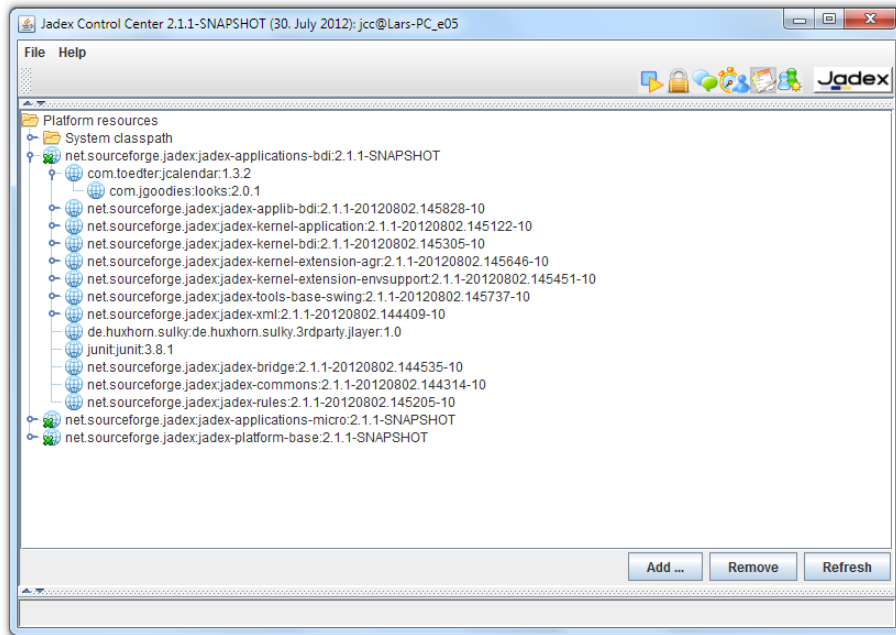


Figure 109: 06 Library Center@libservice.png

The library center presents a deployment oriented resource view of the platform. Platform resources are organized in a resource tree. The meaning is that a resource with child resources depends on those child's resources and needs them for execution. Internally, resources correspond to Java classloaders (each resource is represented by exactly one classloader) so that the hierarchy also shows how resource fetching is performed. The library center can be used to inspect the current deployment setting and also allows for adding or removing resources at arbitrary levels within the resource tree. In order to understand better the concrete setup of the resource tree a conceptual model is shown below.

Resource tree

The root of the resource tree contains basic system resources directly as classpath entries and all further resources via dependencies to subresources. It can be seen that each resource is described via a resource identifier (RID), which has the purpose to uniquely identify a resource. Resource identifiers come in two flavors. Global resource identifiers use a globally unique id represented by a maven artifact id to identify a resource. Local resource identifiers use a local URL to refer to a file representing the resource. Global resource identifiers are automatically resolved at runtime to a concrete local path to the resource and are hence enriched with a local resource identifier.

Note: Maven dependency support is part of the Jadex Pro version.

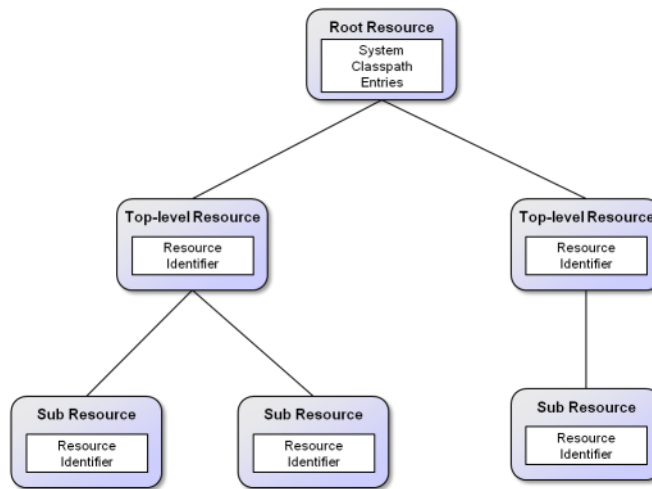
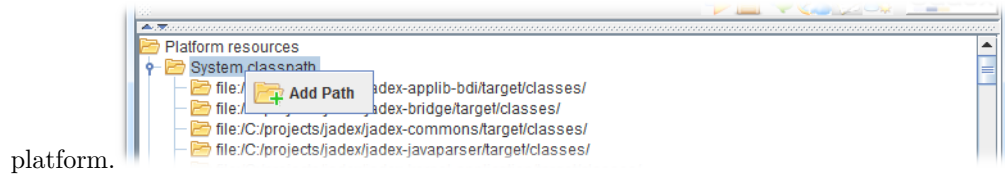


Figure 110: 06 Library Center@resourcetree.png

Basically, two options for modifying resources are available to modify the resource tree, which are described in the following.

Adding/Removing System Level Classpath Entries

Changing the global classpath allows for adding resources to all resource on the platform, i.e. a new resource is visible to all top-level and subresources. A system level classpath entry can be added by activating the popup menu 'Add path' option on the 'System classpath' node (as shown below). To remove a classpath entry the 'System classpath' node needs to be opened and the corresponding child has be selected and removed via 'Remove Path' from the popup menu. Please note that only manually added entries can be removed (indicated by the small ✖). Furthermore, removal of system classpath entry will take effect only after a restart of the



platform.

Adding a system level classpath entry

Adding/Removing Resources in the Resource Tree

A new resource can be added to the resource tree by choosing the ‘Add Path’ or ‘Add RID’ action from the popup menu of a tree resource or by choosing the ‘Add’ button from the bottom of the panel. If no node in the tree is selected while pressing the ‘Add’ button, new resources will be added as top-level resources. The difference between the ‘Add Path’ and ‘Add RID’ actions is that the first adds a local path, while the latter adds a maven artifact id. Please note that a resource is a deployment artifact that can be referenced at multiple places within the tree. If you change a resource by adding or removing a resource all usages will automatically see this change. A resource can be removed by activating the ‘Remove Path’ option from its resource identifier node (see screenshot below).

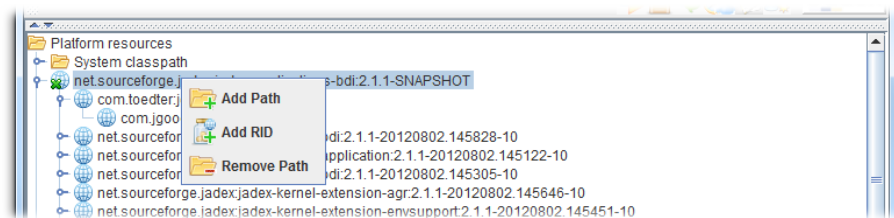


Figure 111: 06 Library Center@addremres.png

Chapter 7 - Deployer

The deployer tool can be used to transfer files between computers. In this sense it is very similar to a typical ftp tool but it could be convenient to use the Jadex deployer tool for the following reasons:

- Can be used to transfer files between arbitrary hosts within your Jadex network, i.e. given you are on host A you can transfer A to B or vice versa but you can also transfer files between B and C.
- The deployer tool allows for transferring files between hosts that do not have a direct TCP connection as it relies on the Jadex platform transport mechanisms, i.e. it can use a relay server to bridge different unconnected networks.
- The deployer will automatically try to find the most efficient connection to the target and will also tolerate if one (of several available connections) to the target diminishes.
- There is no need to install further software on the hosts with the platform, i.e. no server and/or client.

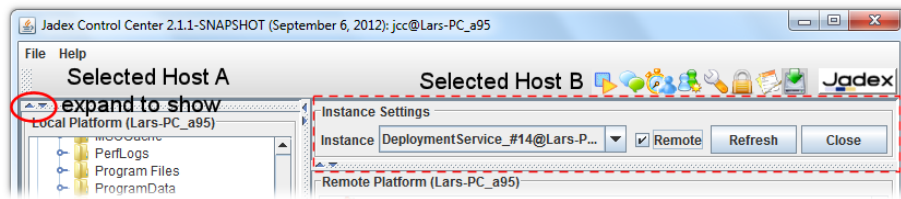
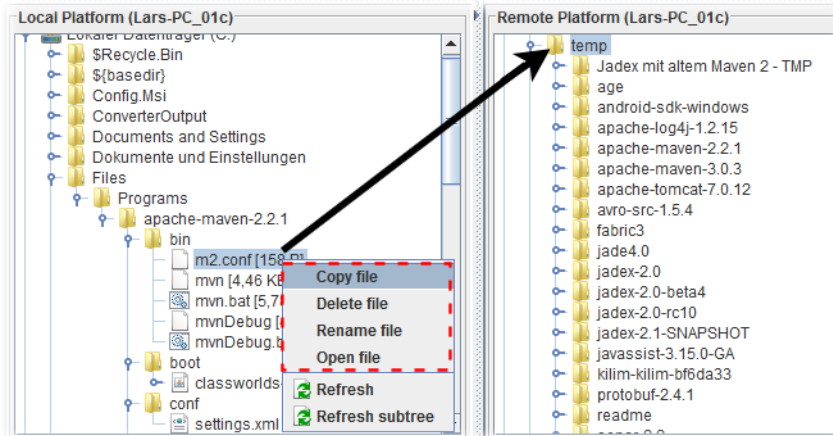


Figure 112: 07 Deployer@host_selection.png

Transferring Files

Select the hosts you want to use by choosing the corresponding Jadex platforms (more concisely deployer services) from the Instance Settings panel. The Instance Settings panel can be made visible by clicking in the left upper area the arrow button (see screenshot above). Having expanded the panel the instance can be selected via the choice box. To discover new remote hosts, click the Remote checkbox and then the Refresh button. The discovered platforms will be added to the choice box. By selecting a new instance the tree view below will show the file system of the corresponding host.



Copying a file can be achieved in two ways. The first option is to select the source file that should be transferred and either use drag and drop to pull it directly to the target folder on the other side. The second option is not based on drag & drop. Instead, first the target folder has to be selected and then the source file has to be chosen. To start the copy process the Copy file action from the popup menu has to be used as shown in the screenshot above.

Further Commands

In addition to copying files between hosts, the deployer also allows for renaming, deleting and opening files. In order to initiate such actions, select a file in the file tree and choose it from the popup menu.

Chapter 8 - Chat

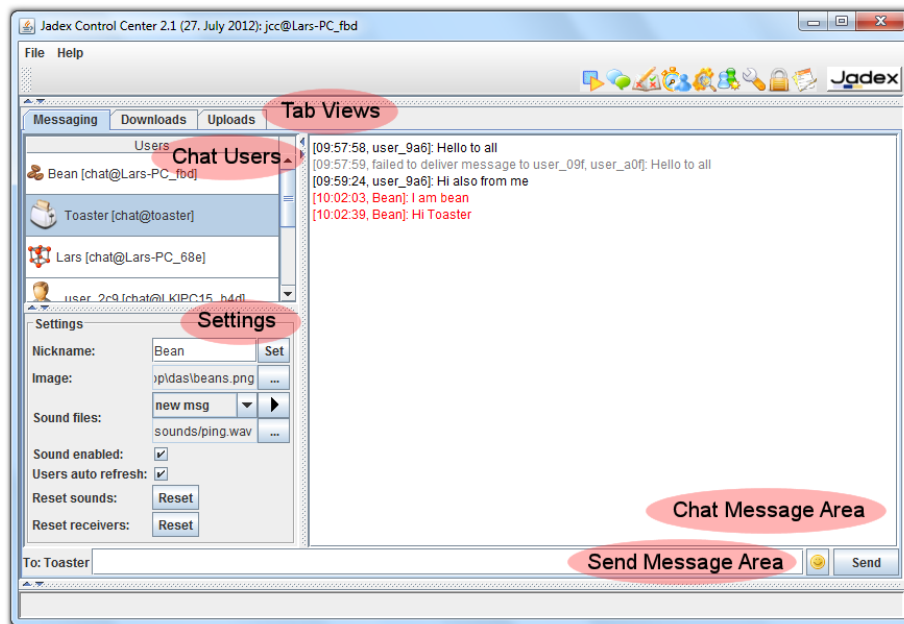


Figure 113: 08 Chat@chat.png

Screenshot of chat plugin

The chat tool is a typical messenger style JCC plugin that can be used to exchange messages and files with other chat users. In contrast to other messengers like ICQ, Yahoo, or MSN, the Jadex chat is conceptually a peer-to-peer chat, i.e. one will see all those users online that have access to the same network. If global awareness is configured via the relay awareness, in principle one can see all other global users. The chat tool itself has three different tab views named **Messaging**, **Downloads**, and **Uploads**, which refer to sending/receiving text messages and binary files respectively.

Messaging

The messaging view contains a **chat user list** in the upper left area, a **chat settings panel** on the lower left, a **chat message area** on the right and a **send message area** at the bottom of the window.

Chat Users

The chat user list contains a list of the chat users currently online. Each user is displayed with an its **image**, **nickname**, and **component identifier**. The image and nickname can be changed in the settings panel below. The component identifier shows the id of the component hosting the chat service used. The chat user list is periodically refreshed and users that cannot be contacted any longer are first displayed as unconnected with an icon shown greyed. If still unconnected during the next refresh the user will be deleted from the list.

Chat Settings

The chat settings panel allow to customize the chat behavior. Currently the following options are available:

- **Nickname:** Can be used to change the user's nickname.
- **Image:** Can be used to change user's image.
- **Sound files:** Important chat events are underlayed with characteristic sounds that can be changed here. First, the event type should be selected using the choice box. The available event types are **new msg** (message), **new user**, **msg failed**, **new file**, **file abort**, and **file complete**. Using the '...' button one can bring up a file chooser to select a new sound file. To test the new sound the play button ▶ can be used.
- **Sound enabled:** Enable or disable to turn on or off all notification sounds.
- **Users auto refresh:** Enable or disable to turn on or off the auto refresh of the chat user list.
- **Reset sounds:** Pressing the reset button will reload the default sounds settings.
- **Reset receivers:** Pressing the reset button will set the chat receivers to 'all'. An alternative to change the receivers to all consists in deselecting one or more selected users by holding down ctrl and clicking the corresponding users in the chat user list.

Send Message Area

In the send message text messages can be entered and sent to other users. On the left hand side the receivers of the text message are displayed. Per default a message is sent to all users currently online (To: all). If you want to send a

private message to one or multiple chatters, you can select those from the chat user list (use shift or ctrl for multi selection). The names of the selected chatters will appear at the left hand side of the panel (such as in the screenshot above 'To: Toaster'). In the middle of the area the text message can be entered into the input field. If you want to use emoticons, you can either use the well known abbreviations, e.g. :-), or select a smiley direction from the corresponding choice on the right. The message will be sent by either pressing return directly in the input field or by hitting the 'Send' button. A message icon will be displayed next to each targeted user in the users list for indicating that message sending is in progress. If a message could not be sent to one of the target users, a corresponding note will be displayed in the message area.

File Transfer

File transfer is performed via a point to point connection between a sender and a receiver. To initiate a file transfer the receiver first has to be selected in the user list. Afterwards the 'send file' action can be used by activating the popup menu (as shown in the screenshot below). The action will open a file chooser to let the user select a file to be sent. On the receiver side a dialog is opened that a new file transfer has been initiated. The receiver can choose to accept or deny the file transfer request within some time interval. The user is presented the filename and size of the source file and can choose a local filename and storage directory. If no choice was made within the interval the transfer will automatically be rejected. In case of a successful transfer start, details about the transfer can be found in the downloads and uploads tab for each side.

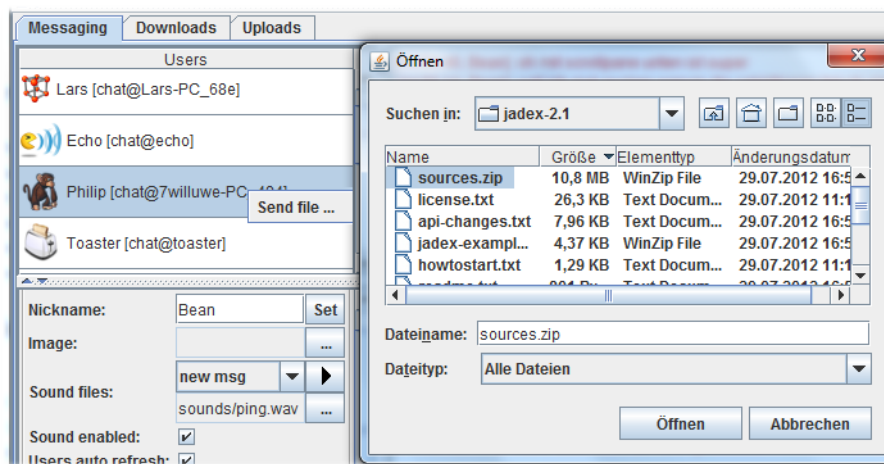
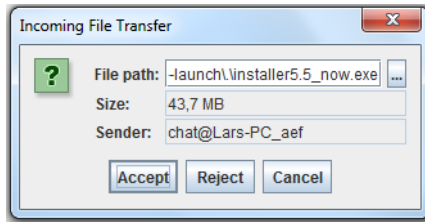


Figure 114: 08 Chat@sendfile.png

File upload via popup menu



File transfer acceptance dialog on receiver side

Downloads

In the downloads tab a table view of downloaded files is shown. The following columns are displayed:

- **Name:** The filename that is used to save the file on the target.
- **Path:** The local path for storing the file.
- **Sender:** The origin of the file. The origin is identified by the component identifier of the source.
- **Size:** The size of the file.
- **Done:** The file portion already downloaded (in bytes, kb, MB).
- **%:** The percentage of the file that already has been downloaded.
- **State:** The state of the transmission. Is one of the following: waiting, rejected, transferring, cancelling, completed, aborted, error.
- **Speed:** The current speed of the download.
- **Remaining time:** The estimated remaining time given that the speed keeps rather constant.

Name	Path	Sender	Size	Done	%	State	Speed	Remaining Ti...
installer5.5_now...	n/a	chat@Lars-PC...	43,7 MB	0B	0%	Aborted		0
Corel_PaintShop...	C:\projects\jade...	chat@Lars-PC...	170 MB	2,64 MB	1%	Transferring	55,3 KB/sec	0:51:40

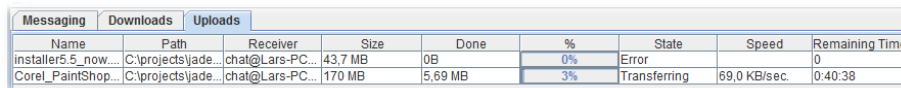
Figure 115: 08 Chat@downloads.png

Download tab

Depending on the state of the downloads different actions can be performed on them. All actions are available via popup menu. If the transfer is not finished yet the only available action will be 'Cancel transfer'. In case the transfer is not running any longer the available actions are 'Open', 'Open folder' and 'Remove'. The first option tries to interpret the file according to its file type, e.g. by starting a viewer for a pdf file and showing it. The second action tries to open the folder in which the downloaded file was stored and finally the thirist option removes the download entry from the list.

Uploads

The uploads panel contains transfer information about uploaded files. The information that is shown the upload tab is equivalent to the one used in the download tab. Despite these similarities two natural differences exist. First, instead of the target file name and path, the original source file name and path are presented to user. Second, instead of the sender, the file receiver is shown.



Name	Path	Receiver	Size	Done	%	State	Speed	Remaining Time
installer5.5_now...	C:\projects\jade...	chat@Lars-PC...	43,7 MB	0B	0%	Error		0
Corel_PaintShop...	C:\projects\jade...	chat@Lars-PC...	170 MB	5,69 MB	3%	Transferring	69,0 KB/sec.	0:40:38

Figure 116: 08 Chat@uploads.png

Upload tab

Chapter 9 - Simulation Control

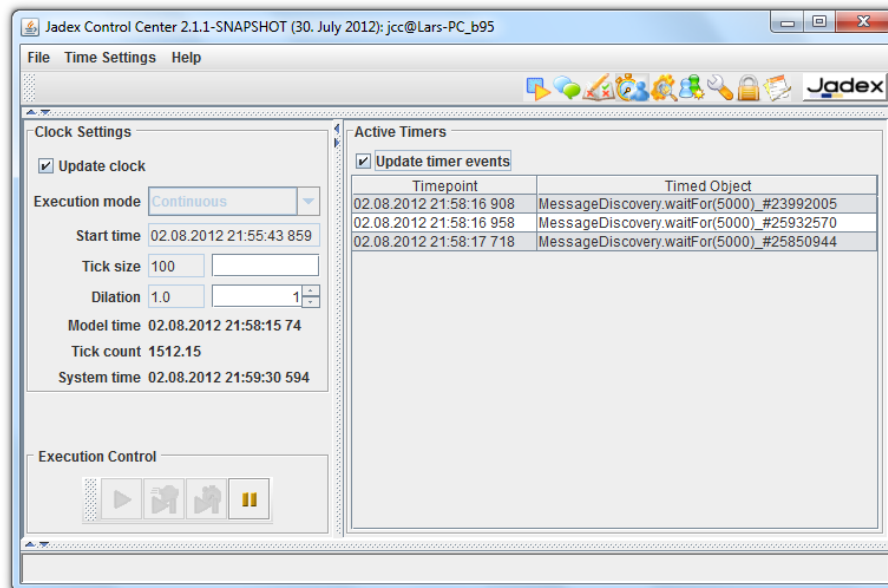


Figure 117: 09 Simulation Control@simulation.png

Simulation control settings screenshot

A fundamental concept realized in Jadex is **simulation transparency**, i.e. applications written in Jadex can be simulated or executed in real time without

code changes. This is realized with a clock abstraction that is used for all timing aspects within an application. By exchanging the clock the timing behavior can be transparently changed. In the simulation control plugin, the clock type and its concrete settings can be defined. Looking at the screenshot above you can see that the simulation control panel consist of three main areas named **Clock Settings**, **Execution Control** and **Active Timers**, which are explained in the following.

Clock Settings

The clock settings area shows details of the clock currently used by the platform.

Update clock: If activated the clock settings are automatically updated.

Execution mode: The execution mode is determined by the clock type that is used. Please note that the clock time can be changed at runtime. For this purpose, first execution has to be paused by pressing the corresponding button from the execution control panel. Currently, the following clock types are available:

- *System clock:* directly corresponds to the computer clock and delivers a time value that corresponds directly to Java `System.currentTimeMillis()`.
- *Continuous clock:* similar to the system clock, but allows for pausing the clock and also using a dilation that makes the clock run faster or slower than the system clock.
- *Event-driven clock:* The event-driven clock uses timing events to advance the clock value, i.e. components that register timing events determine how fast the model time advances. Using event-driven clock as-fast-as-possible simulations are possible, i.e. the whole computation time is used and no unnecessary waiting times exist any longer.
- *Time-driven clock:* The time-driven clock advances the model time by adding a constant (and definable) tick size. In this way the simulation time is advanced regularly which may lead to unnecessary time events, i.e. time points at which no component has something to do. On the other hand this simulation type leads to more realistic observable behavior as no irregular time jumps are produced.

Start time: The init time of the clock.

Tick size: The tick size is important for time-stepped simulations in which the tick size determines the time span between two clock ticks. In Jadex all clock types support ticks, i.e they emit a tick after the tick size amount of time has elapsed. In the field **Tick count** below the number of the current tick is shown. In components it is possible to use the `waitForTick()` method to get notified when the clock has advanced the next tick.

Dilation: The dilation describes the speed of the clock with respect to real time. Values greater one describe faster time progress while values lower than

one indicate a clock slow down. New dilation values can be entered directly into the text field or by using the spinner. The spinner doubles / halves the current value when clicking up / down.

Model time: The model time is the time emitted by the clock. Depending on the clock type and its setting the model time can be completely different from real time. Model time is not advanced if the clock is paused.





Tick count: The number of the current tick. Important only if tick based waiting is used (as described above).

System time: The time of the built-in computer clock.

Note: Although all clock types are very similar from the interface they offer, there is a fundamental difference between simulation and normal clocks. Simulation clocks are passive in the sense that they do not advance time on their own. Instead, time advancement is triggered explicitly whenever all components have finished their execution with respect to the current point in time. In contrast, normal clocks are active and time automatically advances even if nothing else happens or if components are still busy.

Execution Control

The execution control field can be directly used to stop, resume, and step the currently used clock. The following buttons can be used:

- : The start button allows for initially starting or resuming a previously paused clock.
- : The step event button can be used with simulation clocks only. A paused clock can be forced to advance the clock to the next registered time point by pressing the button. As multiple events may have been registered for the same time point this action may not always cause the model time to advance,
- : The step time button is also valid only for simulation clocks. It advances the clock to the next timepoint, i.e. it will activate all timing entries until one timing entry belongs to a time after those of the current clock reading.
- : This button allows for pausing the active clock (the system clock cannot really be paused, but will stop triggering timing events). Having paused the clock applications that rely on timing will not execute further.

Active Timers

The list of active timers shows the registered timing events. This visualizes the next timer entries that will get executed by the clock. On the left hand side of

the table the time point is given at which the timer will be fired and on the right hand side the object that will be executed is shown. As it costs resources to update the timer entry list the check box **Update timer events** can be used to turn off/on the automatic table refresh.

Chaper 10 - Component Viewer

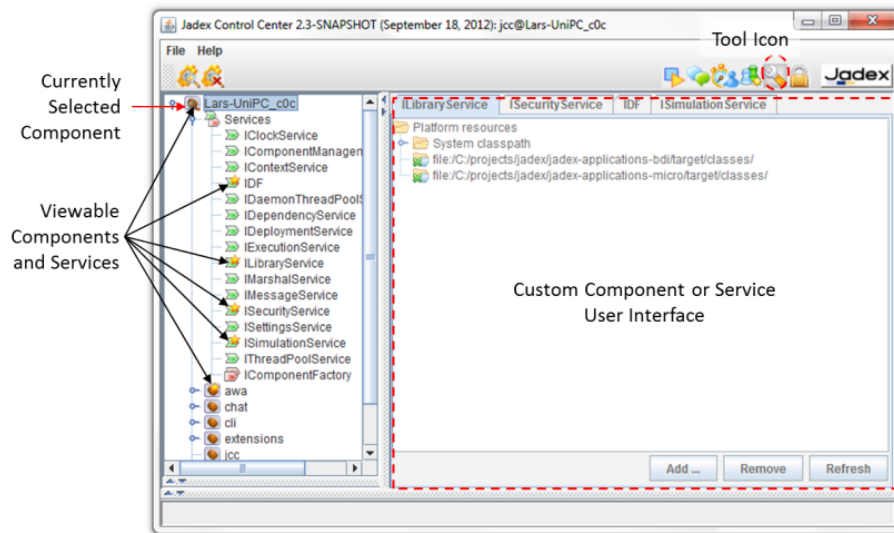


Figure 118: 10 Component Viewer@componentviewer.png

The component and service viewer is a tool that displays custom component or service user interfaces. The tool consists of two areas (as shown in the screenshot above). On the left hand side the component instance tree of the platform is shown. In addition to the normal view, components and services provides a user interface representation are marked with a yellow star. Clicking on such a viewable component or service will open the corresponding visual representation on the right hand side of the tool. As an alternative also the popup action “Open service viewer” can be used to open the user interface. In the screenshot above the platform itself has been selected and as its user interface a tabbed panel of its viewable services is shown. Clicking again on an activated element will close its view.

Please refer to the AC tutorial Exercise F2 - Component Viewer in order to learn about how a user interface for a component can be created.

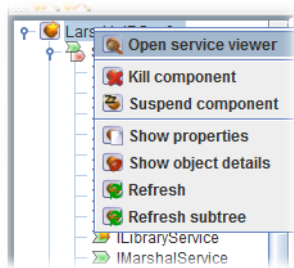


Figure 119: 10 Component Viewer@cvalter.png

Chapter 11 - Test Center

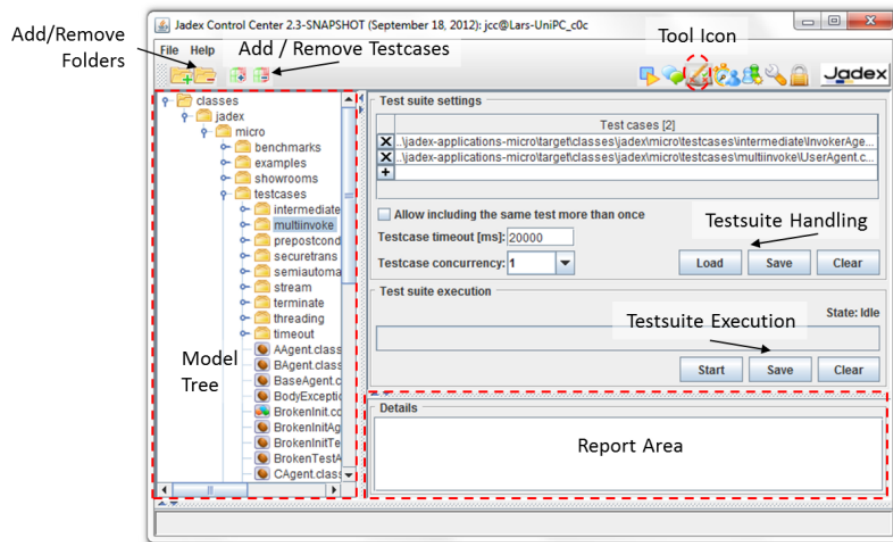




Figure 120: 11 Test Center@testcenter_ov.png

The test center can be used to execute component unit tests within test suites. Therefore, the tool is very similar to the well known visual JUnit test environment. The test center mainly consists of three areas (cf. screenshot above). A model tree on the left, a test suite view on the upper right and a test suite report on the lower right. The general scheme the tool is used consists of the following steps:

1. Select the folders in which your tests reside.
2. Select the test cases and add them to the current test suite.
3. Execute the test suite and inspect the test report.

To include the source folders of the test cases the add folder action  on the model tree can be used (either via the tool bar or via the popup menu). Given that you have included the folders you can now navigate to the folder or file you want to include as test case(s) by selecting the element and afterwards choosing the add testcases action . If the component is a testcase (determined by the fact that it declares a specific return value of type *jadex.base.test.Testcase*), it will appear within the Test Suite Settings panel on the right. Please note that each test case can be included in the test suite only once, because this is a sensible default behavior often desired. If an application case demands including a test case more than once, this can be done by first activating the ‘Allow including the same test more than once’ checkbox and then adding the component as often as needed. The current test suite can be stored on disk by using the Save button in the ‘Testsuite Handling’ area. It will allow you to choose a filename and save the current settings. Such a stored test suite can be fetched from disk by using the corresponding Load button.

Before a test suite is executed, some execution preferences can be changed if necessary. On the one hand the **test case timeout** can be used to alter the maximum waiting time for a single test case. On the other hand the **test case concurrency** determines the degree of test cases that are started at the same point in time. This means that the testcase executor can start n test cases without having to wait for a finish signal, i.e. n=1 means a sequential execution is performed, whereas n=5 means that at most five test cases can be run in parallel. If n=all the test case executor will simply start all test cases at the same time leading to maximum concurrency. To initiate the test case execution the Start button can be used. Once, the test suite is executing, the Start button will be renamed to an Abort button that stops test suite execution.

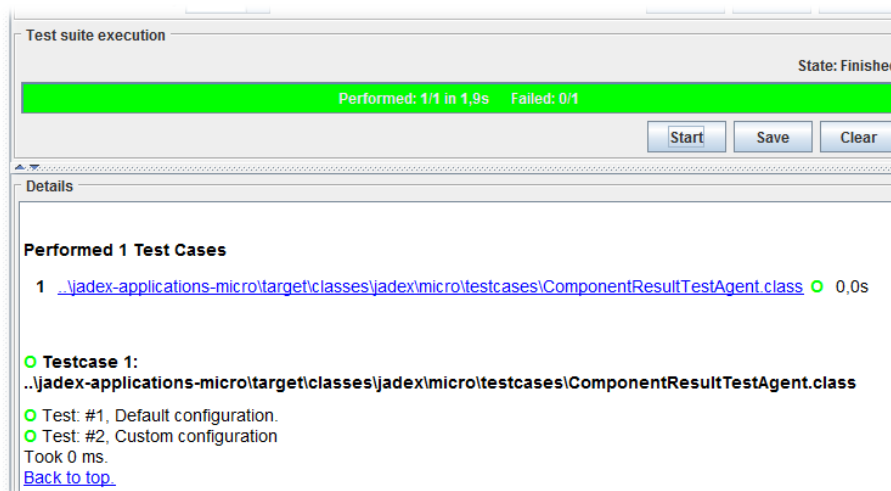


Figure 121:

During test suite execution, the progress is shown with a progress bar that remains green as long as none of the test cases of the suite has failed so far (cf screenshot above). Within the progress bar it is displayed how many test cases already have been performed, how many test cases are in the suite and how long the execution took. In the lower right area the test results are presented as report. The test report consists of an overview area on the top with links to all performed test cases and a details section in which the tests are shown with description and possible the concrete failure reasons. Using the Save button within the Testcase Execution area allows for storing the current report as html file.

Chapter 12 - Debugger

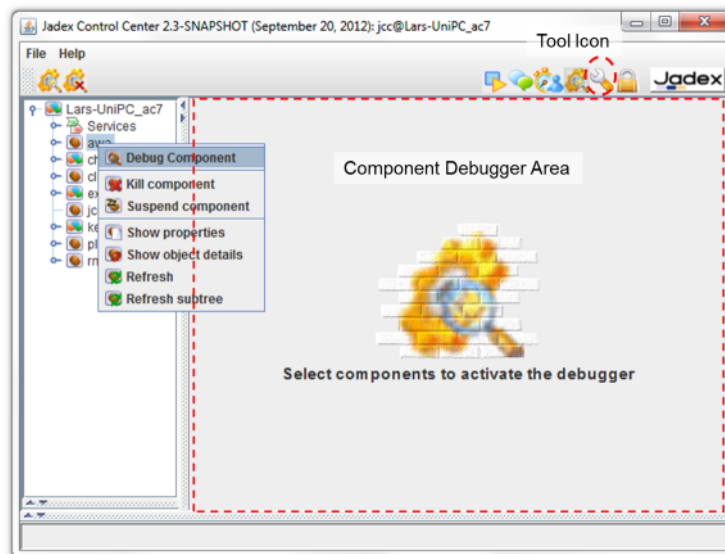


Figure 122: 12 Debugger@debugger.png

Debugger screenshot

The Debugger can be used to inspect components during execution. The Debugger view (as shown in the screenshot above) consists of a tree view at the left and a Component Debugger Area on the right. The tree view shows all currently running components of the platform and can be used to select a component to debug. For this purpose a component can be either double clicked, or the Debug Component action can be chosen via popup menu or from the toolbar. Having selected a component the debugger view will be opened at the right hand side of the window. It has to be noted that the concrete debugger view depends

on the component type being inspected, i.e. a BPMN process has a completely different debugger compared to a BDI agent. Thus in the following the available component specific debuggers will be presented.

BDI Agent Debugger

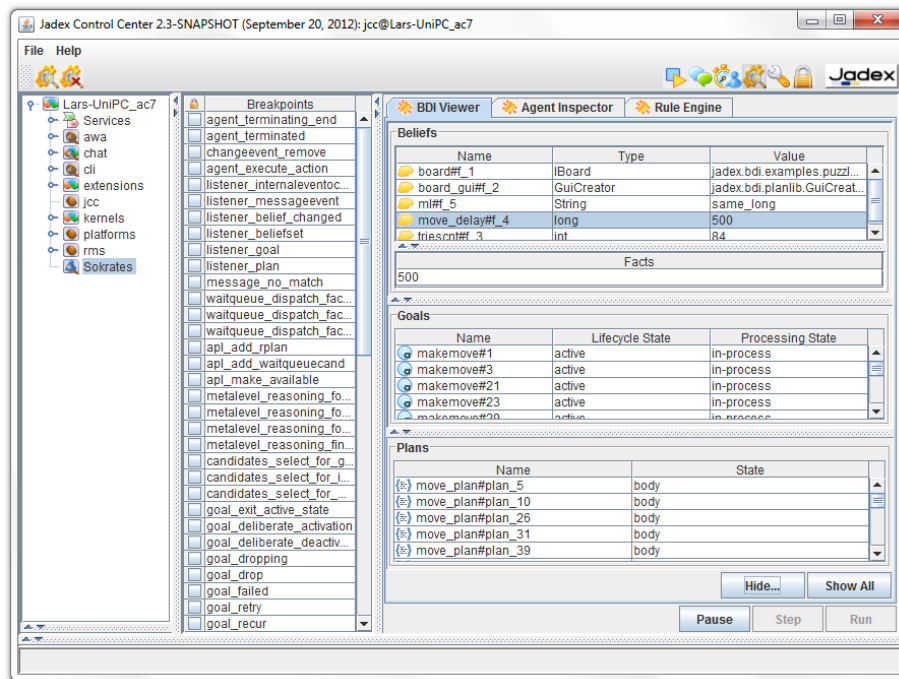


Figure 123: 12 Debugger@bdidebugger.png

BDI Debugger screenshot

The BDI debugger offers four different views:

- **BDI Viewer:** The BDI Viewer shows the beliefs, plans and goals of an agent in the corresponding section. By clicking at a belief or belief set its current value is displayed in the Facts area below. Furthermore, the goal and plan views show the goal and plan instances of the agent and their current state.
- **Agent Inspector:** In the agent inspector the state of the agent can be inspected according to its internal composition. In this tree view more details can be seen about the agent's beliefs, plans, and goals, yet the view is also more difficult to interpret as one has to navigate along the agent structure to get to the elements.

- **Rule Engine:** The rule engine is also a rather internal view that shows which rules make up the behavior of the agent. In addition, the rule engine view contains a visual representation of the rules of the agent as Rete network.
- **Breakpoints:** For a BDI agent in the breakpoints view all rules of the agent are listed, i.e. by selecting a rule the interpreter automatically stops just before a rule of the given type gets executed. One can use the Step, or Run buttons to continue processing.

BPMN Process Debugger

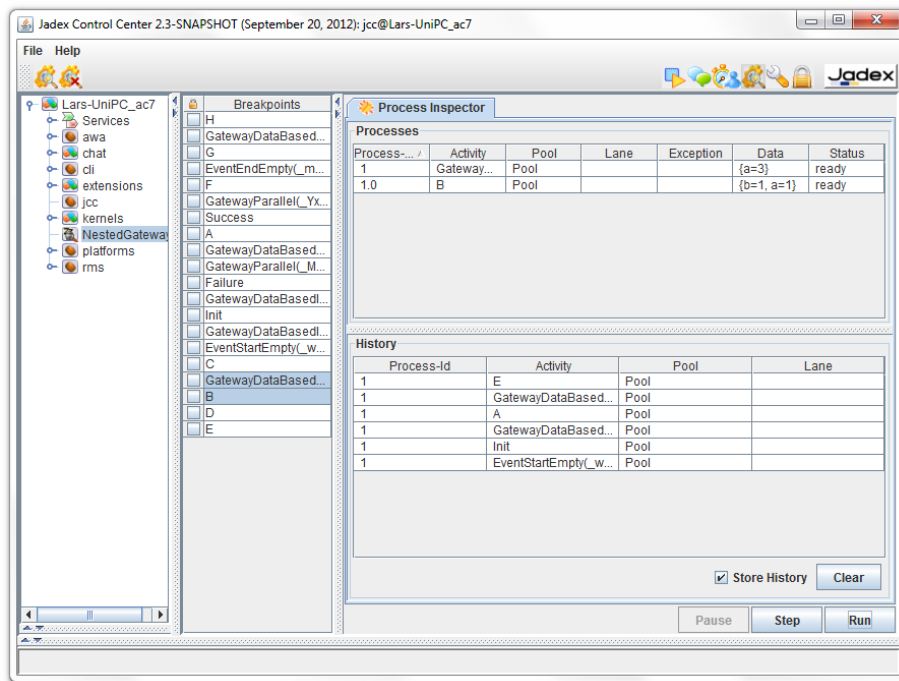


Figure 124: 12 Debugger@bpmndebugger.png

The BPMN process debugger comprises three views:

- **Process threads:** In Jadex, parallel execution within one BPMN workflow are described with (virtual) process threads. A process thread is characterized by the current program counter (the BPMN activity or element that gets executed next) and its own state (variable values within this thread) and execution state (if it is waiting or ready). Concretely the following properties are shown:

- Process id: The unique id of the process thread.
- Activity, Pool, Lane: The activity the thread wants to execute and the location of that activity with respect to the pools and lanes.
- Exception: The exception instance if one has occurred during execution of the process thread.
- Data: A map containing the specific variable values as name, value pairs.
- Status: The current state of the thread. If it is ready the next step can be executed if it is waiting some conditional has to be met before execution can continue.
- **History of steps:** In the history the executed steps and their association to the process thread that executed them are shown.
- **Breakpoints:** In the breakpoint list of BPMN process all its activities, gateways and events are shown. By selecting such an element and executing the process the debugger will suspend execution until such an element is reached.

Micro Agent Debugger

[12 Debugger@microdebugger.png](Breakpoint()* annotation. Additionally, a method has to be implemented that checks if a specific breakpoint has been reached. When implementing a pojo micro agent the @AgentBreakpoint annotation can be used, otherwise the method should override

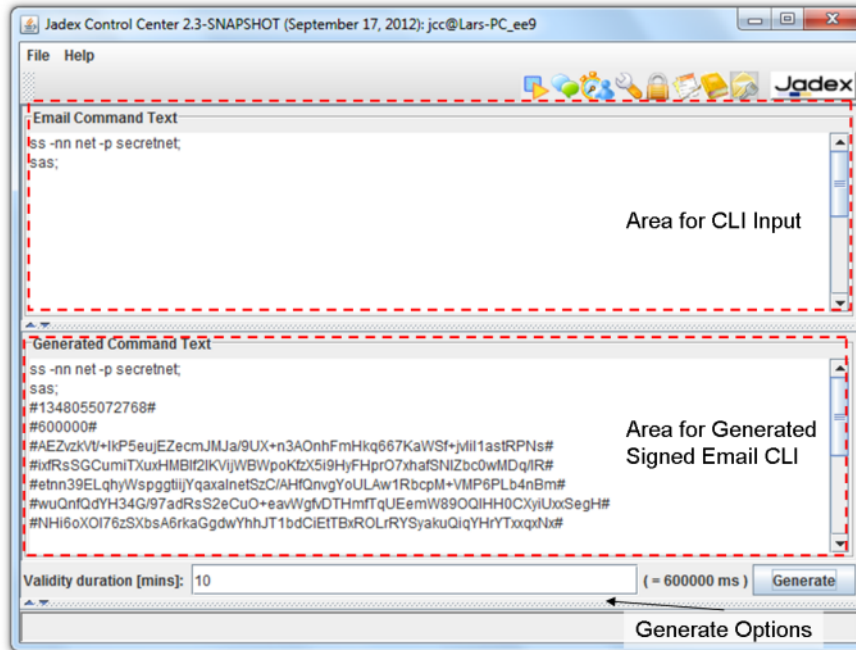
```
public boolean isAtBreakpoint(String[] breakpoints)
```

. See *jadex.micro.testcases.semiautomatic.BeakpointAgent* and *PojoBreakpointAgent* for example code.

Note: Currently, the XML component types ‘component’ and ‘application’ do not possess a dedicated debugger. Instead, if a component of such a type is selected for debugging an Object Inspector view will be shown. This view displays the object structure of the underlying component and can be used to get information about the current state of the object.

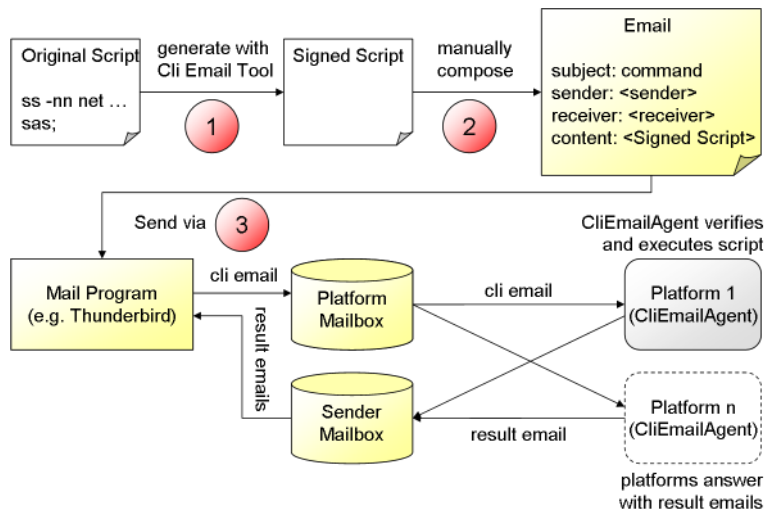
Note: Jadex component debuggers are not meant to replace other existing debugging tools. In contrast, in many cases it is beneficial to combine traditional object oriented debuggers (like the eclipse debugger) with the more high-level component debuggers in order to find reasons for errors in the code.

Chapter 13 - Cli Email Signer



Screenshot of the Cli Email Signer

The Cli (command line interface) Email signer allows for signing command scripts that can be executed by the Jadex Cli platform processor. The idea is that one can use emails to send command scripts to specific Jadex platforms, which then verify that the script comes from a trusted origin and then execute the script and send a reply email with the result of the execution. The singer tool helps to sign scripts making them acceptable for other Jadex platforms. Please note that only platforms will execute a script that share a secret with the sender (cf. platform security mechanisms).



Using the signer tool

The general architecture and workflow is depicted in the architecture view shown above. The steps that need to be manually performed are marked with a red circle and comprise the following:

1. Create a script text that should be executed based on Jadex Cli commands and possibly custom extensions. As an example a script could e.g. start a new application or trigger an update of an application. The script has to end each command with a semicolon. Then the script needs to be copied into the upper text area of the Cli Email Signer. Before hitting the Generate button you may want to change the validity duration of the script. If a platform receives a script after this duration has elapsed it will refuse executing it. Pressing Generate will create a signed script in the lower text area.
2. This step needs to be done with an email client like Thunderbird. Start a new email and copy the generated signed content to the email content area. The subject of the email has to correspond to the email filter used and should be set to command. Finally, the receiver has to be configured. Here a mailbox has to be used that the platforms observe.
3. Send the email via the mail program. If platforms are listening on the mailbox, they will regularly check for new mails. If a mail has a fitting subject it will be executed by the CliEmailAgent and after execution has finished a result mail will be assembled and sent to the initiator. The initiator will receive a result mail from each platform currently listening on the mailbox.

Preparation of the platforms: On each platform that should be enabled to process email scripts an instance of the *jadex.platform.service.cli.CliEmailAgent* has to be started. The agent has to be configured with the email account that should be used, i.e. it has to know details such as the username, password, smtp

and imap server addresses of the account.

Note: The Cli Email Signer as well as the cli processor and the CliEmailAgent are only part of the Jadex Pro distribution.

Chapter 14 - Relay Server

The relay server acts as a rendezvous point for allowing platforms from different protected networks discover and connect to each other.

Relay Web Application



Figure 125: 14 Relay Server@relay-web.png

Screenshot of the global Jadex relay web site

Chapter 15 - Eclipse ADF Checker Plugin

The Jadex ADF Checker is an eclipse plugin that allows checking Jadex *agent definition files* (ADFs), i.e. Jadex source files like *.component.xml*, *.agent.xml*,

.bpmn etc. These files typically contain Java expressions, which are only interpreted by Jadex and thus usually not checked by Eclipse. The ADF Checker closes this gap by attaching itself to the project build process and loading all ADFs in the current project for adding problem markers in eclipse, to show where errors are found in the files.

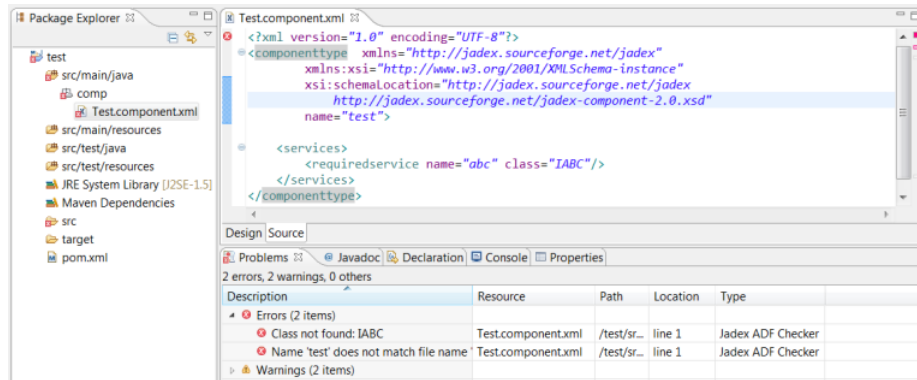


Figure 1: The ADF Checker in action

Requirements

The plugin should work with Eclipse 3.5 or later. It has been tested on Eclipse 3.6 (Helios), Eclipse 3.7 (Indigo) and Eclipse 4.2 (Juno).

The plugin requires that all Jadex libraries, which are used in the project, are available in the project's build path, e.g. as referenced libraries or as maven dependencies. The ADF checker uses whatever version of Jadex is in the build path for loading the ADFs. Therefore, you can choose the Jadex version that you want to use yourself. Due to some changes in the Jadex code base, not all file types are supported for all recent Jadex versions:

- Jadex 2.0 final
 - **.component.xml** supported, but no extensions
 - **.application.xml** supported
 - **.bpmn** supported
 - **.gpmn** not supported
 - **.agent.xml** not supported
 - **.capability.xml** not supported
- Jadex 2.1-SNAPSHOT or later (nightly build or maven snapshot from 02/2012 or newer)
 - **.component.xml** supported
 - **.application.xml** supported
 - **.bpmn** supported
 - **.gpmn** supported
 - **.agent.xml** supported

- `.capability.xml` supported

Installation

The plugin can be obtained from the eclipse update site <http://www2.activecomponents.org/eclipse/update/>](<http://www2.activecomponents.org/eclipse/update/>) . In Eclipse, choose the menu *Help->Install New Software...*, enter the update site address at *Work with:* and select *Jadex Eclipse Plugins->Jadex ADF Checker*.

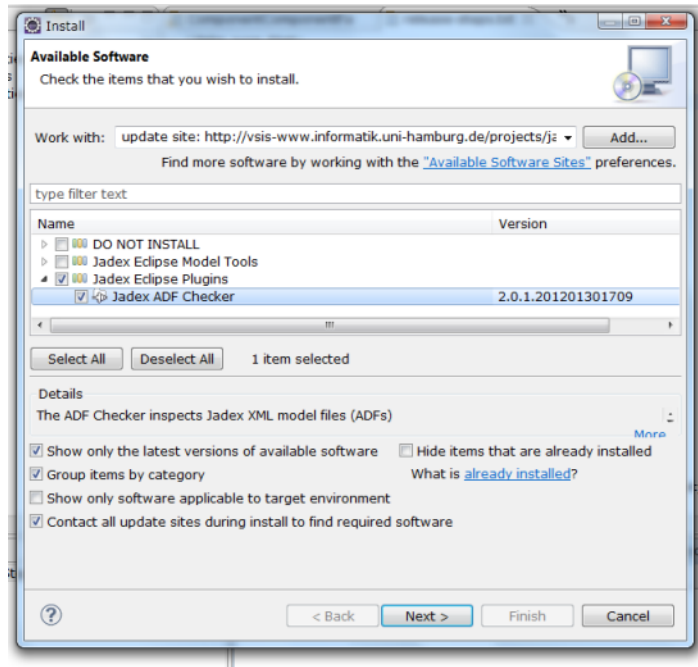


Figure 2: Installing the ADF Checker

Usage

The ADF Checker can be activated and deactivated individually for each Eclipse project. Right-click on the project and choose *Jadex->Enable ADF Checking on Selected Project(s)*.

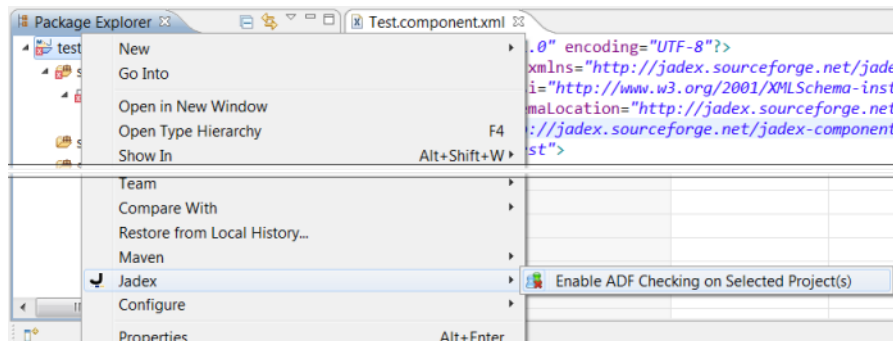


Figure 3: Activating the ADF Checker

The ADF Checker will be executed automatically as part of the Eclipse build process. It will check files that are contained in source folders and that are not excluded from the build path. This assures that it will behave as expected without further configuration for most Eclipse projects.

When ADF checking is enabled on a project, the checking of files is automatically triggered as follows:

- All ADFs will be checked during a complete rebuild, i.e. after doing a *Project->Clean...* in Eclipse.
- Each changed ADF will be rechecked after it is saved, if *Project->Build Automatically* is enabled in Eclipse.

All found problems are reported in the corresponding files editor views as well as in the Eclipse *Problems* view as shown in Figure 1. An info note is added to files that are checked without errors as shown below:



Figure 4: Info Note on Successfully Validated ADF

Known Issues and Limitations

The information reported by the ADF Checker corresponds to the error reports shown in the JCC when loading a broken model. Due to the interpreted nature of Jadex, loading an ADF does not check all aspects. Therefore, runtime errors are still possible, even when the file was successfully checked. We are continuously

trying to improve error reports and add additional checks at load time, and we would be happy to listen to your suggestions on how to do that and what to check.

The following is a list of issues that we are already aware of:

- **Line numbers for errors are not always available.** This is due to models being loaded in more than one pass. Only errors found in the first pass currently have line numbers.
- **Custom component factories are not yet supported.** Currently, only the file types listed in the *Requirements* section above are supported. A configuration option for adding custom component types will be added in the future.
- **Custom extensions are not yet supported.** Currently, only EnvSupport and AGR are supported. A configuration option for adding custom extensiontypes will be added in the future.
- **Overlapping source/resource folders when using Maven and M2Eclipse.** M2Eclipse has a bug to set source includes to `**/*.java` when source and resource folders overlap. Unfortunately, this is a common setting for Jadex projects, because we consider it useful placing e.g. `.component.xml` files in the same folder as corresponding `.java` files. As a result, ADFs are ignored during the build and the ADF Checker is not invoked. To fix this issue, you currently have to manually remove the inclusion pattern from the project setup. Right-click on the project, choose *Build Path->Configure Build Path* and open the *Source* tab. Select the *Included: **/*.java* entry and click *Remove*.

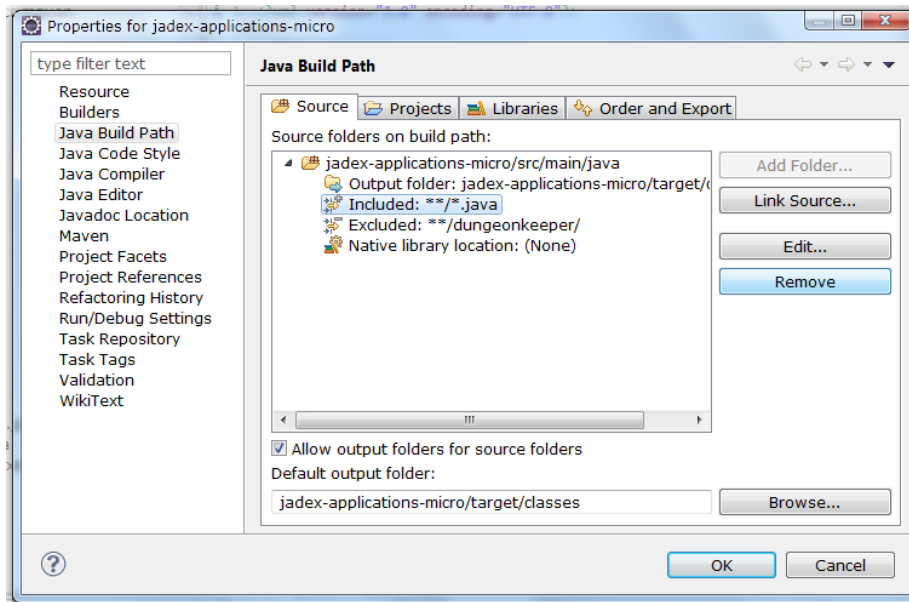


Figure 5: Fixing Maven Project Setup

- **Conversation id:** A conversation id should be set to a unique id and is used to find messages belonging to the same conversation instance.
- **Reply with:** Similar to a conversation id but used only in request-response scheme. This field can be set by the sender of a message.
- **In reply to:** In this field a receiver of a message with a non-empty reply with field is expected to copy the value of that field. The original sender can check if the value fits to its value in the reply with field.
- **Reply by:** The deadline until which a response is expected at latest.
- **Language:** The language that should be used to encode the message. In Jadex predefined languages are jadex-xml and jadex-binary.
- **Encoding:** The encoding type of the message.
- **Ontology:** The ontology name of the message.
- **Content:** The content of the message as string representation.

It has to be noted that most of the parameters are optional except the receiver(s) of a message. Using the Send button the message will be delivered to the actual receivers. Using the Clear button all fields will be emptied.

In the left area two lists of messages are presented. The first one called Sent Messages contains messages that have been sent from the conversation center to other parties. The second list called Received messages contains all messages that have been sent from other to the current platform. To inspect a received message further it can be double clicked in the list. This will open a new tab in the right area in which the parameter values of the received message are displayed. The tab will offer a Close and a Reply button in the tab. The reply button will automatically create an editable message as reply to the received one, i.e. the corresponding field values will be copied/exchanged.

After having altered the parameter values as desired the reply message can be sent back to the original sender using the Send button.

Chapter A2 - Communication Analyzer

Chapter A3 - Directory Facilitator

The directory facilitator tool can be used to inspect the component and service registrations done at the directory service (DF) of a platform. Please note that using Jadex V2 the directory facilitator is not used any more as default for service registrations, i.e. you won't find any service of an active component registered at the DF. But as there are still some (older) example applications such as Marsworld using the DF the corresponding GUI tool is still supported. It basically offers a tripartite view. In the topmost list called Registered Component Descriptions, all currently registered component descriptions are shown. Each component is allowed to only register exactly one component description that may contain an arbitrary number of service descriptions. These service descriptions

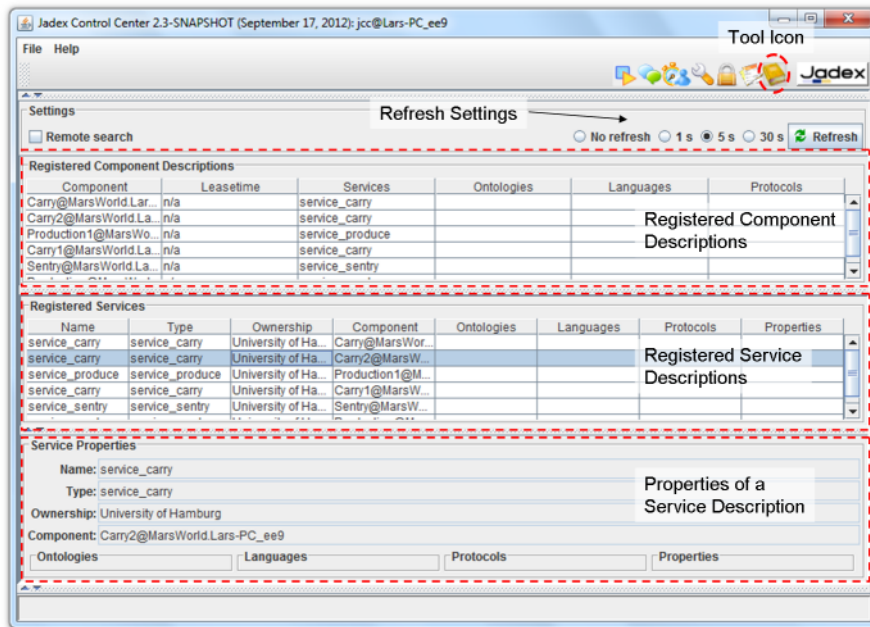


Figure 127: A3 Directory Facilitator@df_ov.png

are depicted in the list below called Registered Services. In the view at the bottom the details of one specific service description are displayed. The idea is that you can first select an interesting component description at the top and then select one of the contained service descriptions, which are subsequently shown with all aspects below. The tool also allows to remove a component registration by using the corresponding list, selecting an entry and choosing the *Remove component action* from the popup menu.