

Praktikum: Intelligente Agenten in Jadex

In diesem Termin werden Sie die *Jadex* Agentenplattform kennen lernen. Sie stellt eine generische Middleware zur Entwicklung verteilter Agentensysteme bereit. Nachdem sie das Verhalten der sog. *CleanerWorld* bereits kennen gelernt haben, werden Sie nun die Arbeitsweise der Agenten im Detail untersuchen und sie erweitern.

Aufgabe 1 – Installation und Dokumentation

Distributionen des Jadex-Systems können von der Projektseite unter:

<http://vsis-www.informatik.uni-hamburg.de/projects/jadex/download.php>

bezogen werden. Bitte laden Sie sich die aktuelle Version (gegenwärtig Version 0.96 beta1) herunter und richten Sie sich Ihren Arbeitsplatz so ein, dass die in der Distribution enthaltenen .jar Dateien entweder im Classpath ihres Systems oder in den Projekteinstellungen ihrer bevorzugten IDE verfügbar sind. Achten Sie darauf, dass Ihre IDE auch XML unterstützt (z.B. durch Nachinstallation des Eclipse Webtools-Projekts WTP).

Das System kann mittels der Aufrufe:

```
java
jadex.adapter.standalone.Platform
[-conf filename] [-transport
classname:port] [-notransport] [-
nogui] [-noamsagent] [-nodfagent]
[-autoshtutdown]
oder
java -jar jadex_standalone.jar
[options]
```

gestartet werden.

Es erscheint die Oberfläche zur Administration des Systems (das sog. Jadex Control Center).

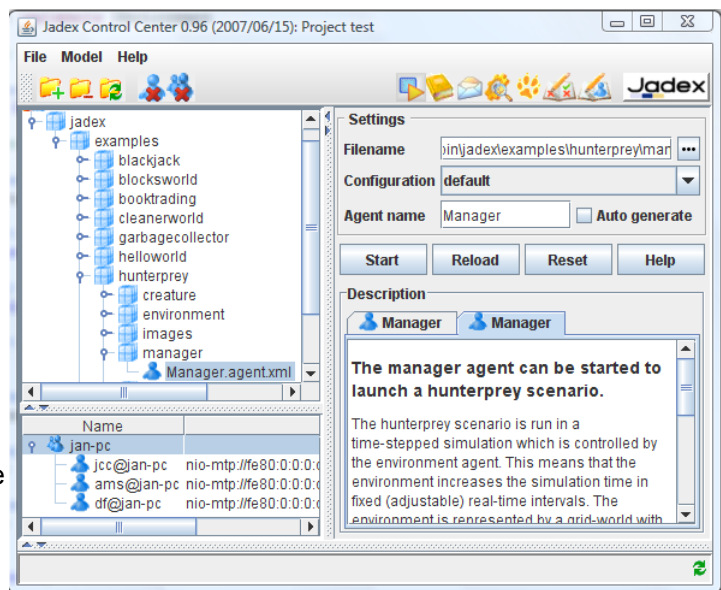
Außerdem sollte die Dokumentation (Userguide / Tutorial / Toolguide) der heruntergeladenen Version hilfreich sein. Dazu folgende allgemeinen Fragen:

Frage 1:

Aus welchen Elementen setzen sich Jadex Agenten zusammen?

Frage 2:

Welche Werkzeuge stellt das System bereit und wie können diese eingesetzt werden?



Die Cleanerworld

Das sog. Cleanerworld-Beispiel kennen Sie aus der Vorlesung. Sie finden es im Package `jadex.examples.cleanerworld.multi`. Die Anwendung wird mittels eines Manager-Agenten gestartet (`jadex.examples.cleanerworld.multi.manager.Manager.agent.xml`).

Frage 3:

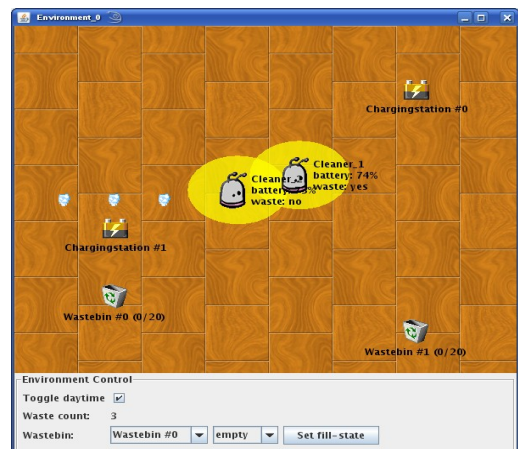
Vergewöhnen Sie sich die Arbeitsweise des Agenten. Er beinhaltet zwei Konfigurationen, die sich in der Anzahl der gestarteten Agenten unterscheiden. Erstellen Sie eine neue Konfiguration in der 4 Agenten gestartet werden.

Frage 4:

Die hauptsächliche Aufgabe der Agenten ist der Abtransport von Müll zu den Papierkörben. Nutzen sie die verfügbaren Werkzeuge, um die Abfolge von Zielen und Plänen zu identifizieren, die diese Funktionalität realisieren. Nennen Sie die Abfolge der Ziele und Pläne und beschreiben Sie wichtige Konzepte aus dem Body der Pläne.

Frage 5:

Eine wichtige Eigenschaft der Agenten ist die Erhaltung eines Mindestladestandes der Stromversorgung. Dies wird durch ein *maintaingoal* realisiert. Was passiert wenn sie die `<targetcondition>` auskommentieren? Warum?



Frage 6: Agenten-Koordination:

Ein wesentlicher Vorteil der zielorientierten Programmierung ist, dass Verhaltensweisen als unabhängige Komponenten (Ziel/Plan-Teilbäume) definiert werden. Da eine *Reasoning Engine* die Verfolgung von Zielen und Ausführung von Plänen zur Laufzeit bestimmt, können Agenten leicht erweitert werden indem neue Zeile / Pläne zum Agenten hinzugefügt werden. In der letzten Teilaufgabe werden die untersuchten *Cleaner-Agenten* um neue Verhaltensweisen, bzw. die Fähigkeit zur Koordination erweitert.

Bisher suchen *Cleaner-Agenten* unabhängig voneinander. Wenn Sie zufällig auf eine Menge von *Waste-Objekten* treffen, transportieren die diese nacheinander zum Papierkorb. Nun sollen die Agenten dahingehend erweitert werden, dass sie sich gegenseitig über Ihre *Waste-Funde* informieren und so gemeinsam größere Ansammlungen von *Wastes* abtransportieren (6.1). Außerdem sollen die Agenten ihre Suche in der Umgebung koordinieren (6.2). zu diesem Aufgabenblatt stellen wir Ihnen eine *Capability* bereit, die die benötigte Kommunikation zwischen den Agenten realisiert, so das sie sich auf die Anwendungslogik konzentrieren können.

Die einzelnen Arbeitsgruppen teilen sich bitte so auf, dass beide Teilaufgaben bearbeitet werden. Bitte erläutern Sie die Gesamtlösung der beiden Aufgaben 6.1 und 6.2., also den bereits vorhandenen Lösungscode zusammen mit Ihren Ergänzungen. Beschreiben Sie dabei, wozu die verschiedenen Pläne, Ziele, Events, usw. notwendig sind. (Es wird für 6.1 und 6.2 jeweils ca. eine DIN A 4 Seite Text erwartet.)

6.1: Koordinierter Waste-Abtransport:

Erlauben sie den *Cleaner-Agenten* sich gegenseitig zu informieren, wenn sie *Waste-Objekte* gefunden haben um so gemeinsam größere Haufen von *Wastes* abzarbeiten. *Waste-Objekte* die im Sichradius erscheinen werden automatisch im Beliefset "wastes" eingetragen. Es werden automatisch Ziele gestartet um diese *Waste-Objekte* abzutransportieren. Es genügt also die *Waste-Objekte* zwischen den Agenten auszutauschen.

Lösungsskizze: Hier stellen wir einen einfachen Lösungsansatz vor. Dies ist nur ein Vorschlag um den Umfang der Aufgabe zu verdeutlichen. Sie sind frei eigene Lösungen zu konzipieren.

1: deklarieren Sie einen Plan, der gestartet wird, wenn der Beliefset „wastes“ sich ändert (Hinweis: trigger des Plans anpassen. Schauen Sie sich in der Jadex-Dokumentation an, welche Unterelemente "trigger" aufweisen kann).

2: Implementieren Sie den deklarierten Plan um den Versand von *Waste-Objekten* zu veranlassen. Zum Nachrichten-Austausch siehe: „Weitere Allg. Lösungshinweise“

Zusatzaufgabe (optional): Nun werden Agenten auch zu *Waste-Objekten* fahren, die andere Agenten schon abtransportiert haben. Erweitern sie die Agenten so das ebenfalls ausgetauscht wird, wenn *Waste-Objekte* aufgenommen wurden.

6.2: Koordinierte Suche:

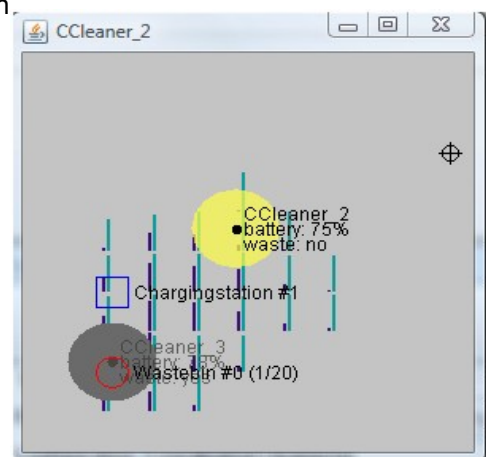
Die Agenten sollen ihre Such in der Umgebung koordinieren. Die *Cleaner-Agenten* merken sich die bereits besuchten Bereiche der Umgebung (Beliefset: „*visited_positions*“) und entscheiden anhand dieses Wissens welche Bereiche noch besucht werden sollen (Plan: *LeastSeenWalkPlan*). Bereiche die bereits von anderen Agenten besucht wurden brauchen nicht mehr von anderen Mitgliedern eines Teams besucht werden. Sorgen sie dafür das die Agenten sich gegenseitig über ihr Umgebungswissen informieren und so ihre Arbeitsweise koordinieren.

Lösungsskizze: Hier stellen wir einen einfachen Lösungsansatz vor. Dies ist nur ein Vorschlag um den Umfang der Aufgabe zu verdeutlichen. Sie sind frei eigene Lösungen zu konzipieren.

Ein einfacher Lösungsansatz ist es, periodisch das gewonnene Wissen über die Umgebung an die anderen Team-Mitglieder zu senden. Eine alternative wäre es, den Versand beim Auftreten bestimmter Ereignisse, beispielsweise Wissensänderungen zu triggern.

1: Deklarieren sie ein Ziel, zum periodischen Versand der Umgebungsinformationen (Beliefset: „*visited_positions*“).

2: Deklarieren und implementieren sie einen Plan der dieses Ziel erfüllt, indem er periodisch auf die



Wissensbasis zugreift und den Versand entsprechender Nachrichten veranlasst. Zum Nachrichtenaustausch siehe: „Weitere Allg. Lösungshinweise“

3: Sorgen sie Dafür, dass das Ziel aus 1 beim Start des Agenten initialisiert wird (Beispielsweise beim Start des Agenten: <configuration> → <initialgoal>).

Anhang:

Gemeinsame Umgebung:

Agenten können auch in einer gemeinsamen Umgebung gegeneinander antreten. Hierzu muss lediglich der Belief „df“ in den einzelnen Agenten überschrieben werden.

```
<!-- The df to search for the environment. -->
<belief name="df" class="AgentIdentifier">
  <assignto ref="actsensecap.df"/>
  <fact>
    new AgentIdentifier("df@alex",
      new String[]{"nio-mtp://vsiisstaff0:2102"})
  </fact>
</belief>
```

Um die auf einem entfernten Rechner laufende Umgebung zu beobachten, können Sie den Agenten `jadex.examples.cleanerworld.multi.environment.EnvironmentObserver.agent.xml` starten. Auch hier wird die Adresse des entfernten Rechners in der Wissensbasis unter dem Bezeichner `df` angegeben. Im Praktikum werden wir ggf. die Adresse einer gemeinsamen Umgebung bekannt geben.

Kommunikation:

Falls Ihre Agenten miteinander kommunizieren sollen müssen in den ADF alle Nachrichten deklariert werden die empfangen und versendet werden. Details finden sich im Jadex Userguide Abschnitt 9.2. Die hier aufgeführten Beispiele stammen aus dem Hunter-Prey Anwendungsbeispiel (`jadex.examples.hunterprey.*`):

```
<events>
  <!-- Receiving a Location object. -->
  <messageevent name="any_message" type="fipa" direction="receive">
    <parameter name="performative" class="String" direction="fixed">
      <value>SFipa.INFORM</value>
    </parameter>
    <parameter name="content-class" class="Class" direction="fixed">
      <value>Location.class</value>
    </parameter>
    ...
  </messageevent>
  ...
</events>
```

In den Deklarationen der Pläne eines Agenten kann auf diese Nachrichten Bezug genommen werden, um zu veranlassen das der Plan gestartet wird wenn eine entsprechende Nachricht eintrifft:

```
<plans>
  <!-- MyPlan. -->
  <plan name="do_something">
    <body>new MyPlan()</body>
    <waitqueue>
      <messageevent ref="any_message"/>
    </waitqueue>
  </plan>
</plans>
```

```

/**
 * handle Message.
 */
public class AddTargetPlan extends Plan
{
    //----- methods -----

    /**
     * The plan body.
     */
    public void body()
    {

        // retrieve the message:
        IMessageEvent inf = (IMessageEvent)getInitialEvent();

        // retrieve the message content:
        Location loc = (Location)inf.getContent();

        // store the location object:
        getBeliefbase().getBeliefSet("my_targets").addFact(loc);

    }
}
</plan>
...

```

Der Plan der die eingehenden Nachrichten verarbeitet wird zunächst ihren Inhalt extrahieren:
Der Versand von Nachrichten wird über die Java-API veranlasst. Hierzu sind Adressaten durch eindeutige Bezeichner anzugeben. Diese können von einem „Gelbe-Seiten“ Service erfragt werden.

Um sich bei diesem Service anzumelden müssen Agenten die benötigte Funktionalität importieren

```

<capabilities>
  <!-- Include the df capability as dfcap for finding other agents
        and registering the production service. -->
  <capability name="dfcap" file="jadex.planlib.DF"/>
  <!-- Include the move capability as move for basic movement. -->
  <capability name="move" file="Movement" />
</capabilities>

```

und die zu verwendenden Ziele deklarieren:

```

  <!-- Register the agent description at the df. -->
  <achievegoalref name="df_register">
    <concrete ref="dfcap.df_register"/>
  </achievegoalref>
  <!-- Deregister the agent description at the df. -->
  <achievegoalref name="df_deregister">
    <concrete ref="dfcap.df_deregister"/>
  </achievegoalref>
  <!-- Usable for searching other agents. -->
  <achievegoalref name="df_search">
    <concrete ref="dfcap.df_search"/>
  </achievegoalref>
</goals>

```

Agenten melden sog. Services an unter denen sie gefunden werden können. Beispielsweise das sie prinzipiell bei der Jagd helfen würden.

```

<properties>
  <!--<property name="logging.level">Level.WARNING</property-->
  <property name="fipa.servicedescription.hunt">
    SFipa.createServiceDescription(
      "service_hunt",
      "service_hunt",
      "HAW Hamburg"
    )
  </property>
  <property name="fipa.agentdescription.hunt">
    SFipa.createAgentDescription(
      null,
      (ServiceDescription) $propertybase.getProperty("fipa.servicedescription.hunt")
    )
  </property>
</properties>

```

Die Anmeldung soll typischer Weise bei der Initialisierung des Agenten stattfinden:

```
<configurations>
  <configuration name="default">
    <goals>
      <!-- Create initial goals. -->
      <initialgoal ref="df_register">
        <parameter ref="description">
          <value>
            $propertybase.getProperty("fipa.agentdescription.hunt")
          </value>
        </parameter>
      </initialgoal>
    </goals>
  </configuration>
</configurations>
```

Zuletzt ein Beispiel wie nach Agenten gesucht und Nachrichten versandt werden können.

```
// Search for Hunt_Service
// Create a service description to search for.
ServiceDescription sd = new ServiceDescription("service_hunt", null, null);
AgentDescription dfadesc = new AgentDescription();
dfadesc.addService(sd);

// A hack - default is 2! to reach more Agents, we have
// to increase the number of possible results.
SearchConstraints constraints = new SearchConstraints();
constraints.setMaxResults(-1);

// Use a subgoal to search
IGoal ft = createGoal("df_search");
ft.getParameter("description").setValue(dfadesc);

dispatchSubgoalAndWait(ft);
AgentDescription[] hunters = (AgentDescription[]) ft.getParameterSet("result").getValues();

if(hunters.length>0)
{
  IMessageEvent mevent = createMessageEvent("any_message");
  for(int i=0; i<hunters.length; i++)
    mevent.getParameterSet(SFipa.RECEIVERS).addValue(hunters[i].getName());
  mevent.setContent(my_location); // set a Location.class instance !!!
  sendMessage(mevent);
}
```