ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ имени академика С. П. КОРОЛЕВА» (САМАРСКИЙ УНИВЕРСИТЕТ)

ИСПОЛЬЗОВАНИЕ ПЛАТФОРМЫ JADE ДЛЯ РАЗРАБОТКИ МУЛЬТИАГЕНТНЫХ ПРИЛОЖЕНИЙ

Самара 2017

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ имени академика С. П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)

ИСПОЛЬЗОВАНИЕ ПЛАТФОРМЫ JADE ДЛЯ РАЗРАБОТКИ МУЛЬТИАГЕНТНЫХ ПРИЛОЖЕНИЙ

Составитель *Е.В. Симонова*

С А М А Р А
Издательство Самарского университета
2017

УДК 519.876.5
ББК 22.18я73

Составитель ***Е.В. Симонова***

Рецензент: канд. техн. наук, доц. Л.С. З е л е н к о

Использование платформы JADE для разработки мультиагентных приложений: [Электронный ресурс]: метод. указания / *Е.В. Симонова*. – Самара: Изд-во Самарского университета, 2017. – 62 с. : ил. Электрон. текстовые и граф. дан. (Кбайт).- 1 эл. опт. диск (CD-ROM)

Методические указания содержат достаточно подробное описание функциональности агентной платформы JADE, примеры разработки агентных приложений на основе платформы JADE. Даны рекомендации по разработке классов программных агентов. Примеры доведены до практической реализации, приводятся листинги кода классов программных агентов.

Методические указания предназначены для студентов направления 09.04.01 – «Информатика и вычислительная техника» в качестве методических указаний по курсу «Моделирование информационных систем» .

Подготовлены на кафедре информационных систем и технологий.

УДК 519.876.5
ББК 22.18я73

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

В методических указаниях описана функциональность агентной платформы JADE, а также примеры разработки агентных приложений на основе платформы JADE. Приводятся контрольные вопросы, а также индивидуальные задания для выполнения лабораторной работы.

Методические указания предназначены для студентов, обучающихся по направлению 09.04.01 – Информатика и вычислительная техника.

Содержание методических указаний соответствует разделам рабочей программы по дисциплине «Моделирование информационных систем» федерального компонента ГОС подготовки магистров по направлению 09.04.01 – Информатика и вычислительная техника.

ВВЕДЕНИЕ

В настоящее время наблюдается развитие агентных систем, возникших в результате технической эволюции информационных и программно-аппаратных средств современной инфосферы. Для эффективного использования агентных приложений необходимо освоить методологию и инструменты их проектирования и эксплуатации. К числу наиболее широко используемых относится агентная платформа JADE и соответствующая среда разработки агентных приложений, изучению и освоению которых посвящено данное пособие и лабораторный практикум.

Основные цели лабораторного практикума:

- Изучение архитектуры и назначения агентной платформы JADE.
- Практическое освоение среды JADE для создания агентных приложений и управления ими.
- Изучение примеров разработки простых агентных приложений.
- Практическое освоение базовых классов, поддерживающих платформу JADE: классов агента, поведения агента, взаимодействия между агентами, сервиса «желтых страниц».
- Приобретение навыков программирования агентов на основе базовых классов JADE.
- Приобретение навыков построения агентных приложений, агенты которых выполняют сложные переговоры, публикацию и поиск сервисов.

В процессе выполнения лабораторного практикума предлагается разрабатывать агентные приложения в порядке возрастания сложности:

1. Простой агент HelloWorldAgent.
2. Матчинг «Заказ – Ресурс» как пример отношения, наиболее часто встречающегося в сфере производства и логистики, для реализации которого требуется взаимодействие между агентами.
3. «Торговля книгами» – приложение, в котором агенты выполняют переговоры, публикацию и поиск сервисов.
4. «Проекты-программисты» – приложение, в котором агенты ведут переговоры между собой с использованием сообщений различного типа. При этом агенты имеют собственные модели поведения и преследуют различные цели.

1 ОБЩАЯ ХАРАКТЕРИСТИКА АГЕНТНОЙ ПЛАТФОРМЫ JADE

1.1 Назначение агентной платформы JADE и средства, предоставляемые разработчику агентных систем

Агент – это программный объект, способный воспринимать ситуацию, принимать решения и коммуницировать с подобными себе объектами, динамически устанавливая с ними связи. Под *мультиагентной системой* (МАС) будем понимать множество программных агентов, организованных в одно или несколько сообществ, и предназначенных для решения определенной задачи [1-4].

Средой разработки и существования МАС являются *агентные платформы*. Было разработано множество программных реализаций агентных платформ, каждая из которых имеет свои особенности, достоинства и недостатки. Вот лишь небольшой список из более чем ста доступных платформ, публикуемых на сайте организации *AgentLink* (*European Coordination Action for Agent-based Computing* [5]): JADE, FIPA-OS, AOS, ZEUS, KADOMA, NOMADS, ARA, AGLETS, GRASSHOPPER, TRACY, AJANTA, LEAP, JACK, SEMOA. Многие из них успешно существуют в виде коммерческих проектов (таких как JACK) или проектов, позиционируемых как проекты с открытым исходным кодом (JADE, ZEUS и др.).

В 90-х годах возникла необходимость создания единых стандартов на разработку агентных систем. В этот период были основаны две организации *MASIF* (*Mobile Agent System Interoperability Facility*) и *FIPA* (*Foundation of Physical Intelligent Agents*). В результате их работы появились стандарт MASIF и стандарт FIPA [6], дающие рекомендации по созданию систем мобильных агентов и систем интеллектуальных агентов.

Одной из наиболее популярных агентных платформ в настоящее время является платформа *JADE (Java Agent DEvelopment Framework)*. Проект JADE разрабатывается компанией Telecom Italia Lab с 2000 г. [7].

Агентная платформа JADE является типичным *Middleware*, т.е. программным обеспечением (ПО) среднего уровня (Рисунок 1), преставляющим собой набор средств для создания и управления системой с множеством агентов [8].

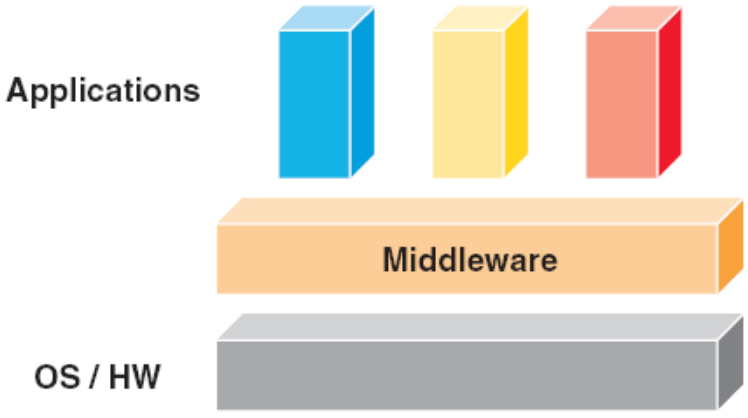

Рисунок 1 – Место Middleware в структуре ПО

Платформа разработки мультиагентных систем JADE включает в себя динамическую среду, где могут «жить» JADE агенты; библиотеку классов, которую программисты могут использовать для разработки собственных агентов; набор графических инструментов, позволяющих управлять активностью запущенных агентов.

JADE предоставляет программисту – разработчику агентных систем следующий набор средств [9]:

- *FIPA-compliant Agent Platform* – агентную платформу, основанную на стандарте FIPA и включающую обязательные типы системных агентов, которые автоматически активируются при запуске платформы.
- *Distributed Agent Platform* – распределенную агентную платформу, которая может использовать несколько компьютеров (узлов), причем на каждом узле запускается только одна Java Virtual Machine. Агенты исполняются как Java-потоки. Для доставки сообщений между агентами, в зависимости от их местонахождения, используется соответствующий транспортный механизм – *Multiple Domains support* – множество основанных на FIPA-спецификациях специализированных агентов, которые могут объединяться в федерацию, реализуя таким образом мультидоменную агентную среду.
- *Multithreaded execution environment with two-level scheduling*. Каждый JADE-агент имеет собственный поток управления, но он также способен работать в многопотоковом режиме. *Java Virtual Machine* проводит планирование задач, исполняемых агентами или одним из них.
- *Object-oriented programming environment*. Большинство концепций, свойственных FIPA-спецификации, представляются Java-классами, формирующими интерфейс пользователя.
- *Library of interaction protocols*. Использование стандартных интерактивных протоколов *fipa-request* и *fipa-contract-net*. Для того чтобы создать агента, который мог бы действовать согласно таким протоколам, разработчикам прикладных программ нужно только имплементировать специфические доменные действия, в то время как вся независимая от прикладной программы протокольная логика будет осуществляться системой JADE.
- *Administration GUI*. Простые операции управления платформой могут исполняться через графический интерфейс, отображающий активных агентов и контейнеры агентов. Используя GUI, администраторы платформы могут создавать, уничтожать, прерывать и возобновлять действия агентов, создавать иерархии доменов и мультиагентные федерации.

Платформа JADE написана на языке программирования Java с использованием Java RMI, Java CORBA IDL, Java Serialization и Java Reflection API. Она упрощает разработку мультиагентных систем благодаря использованию FIPA-спецификаций и инструментов (tools), которые поддерживают фазы исправления ошибок (debugging) и развертывания (deployment) системы. Эта агентная платформа может распространяться среди

компьютеров с разными операционными системами, и ее можно конфигурировать через удаленный GUI-интерфейс. Процесс конфигурирования этой платформы достаточно гибкий. Единственным требованием такой системы является установка на компьютере Java Run Time требуемой версии.

1.2 Архитектура агентной платформы JADE

1.2.1 Контейнеры и платформы

Платформа JADE является распределенной и представляет собой набор *контейнеров* (Рисунок 2).

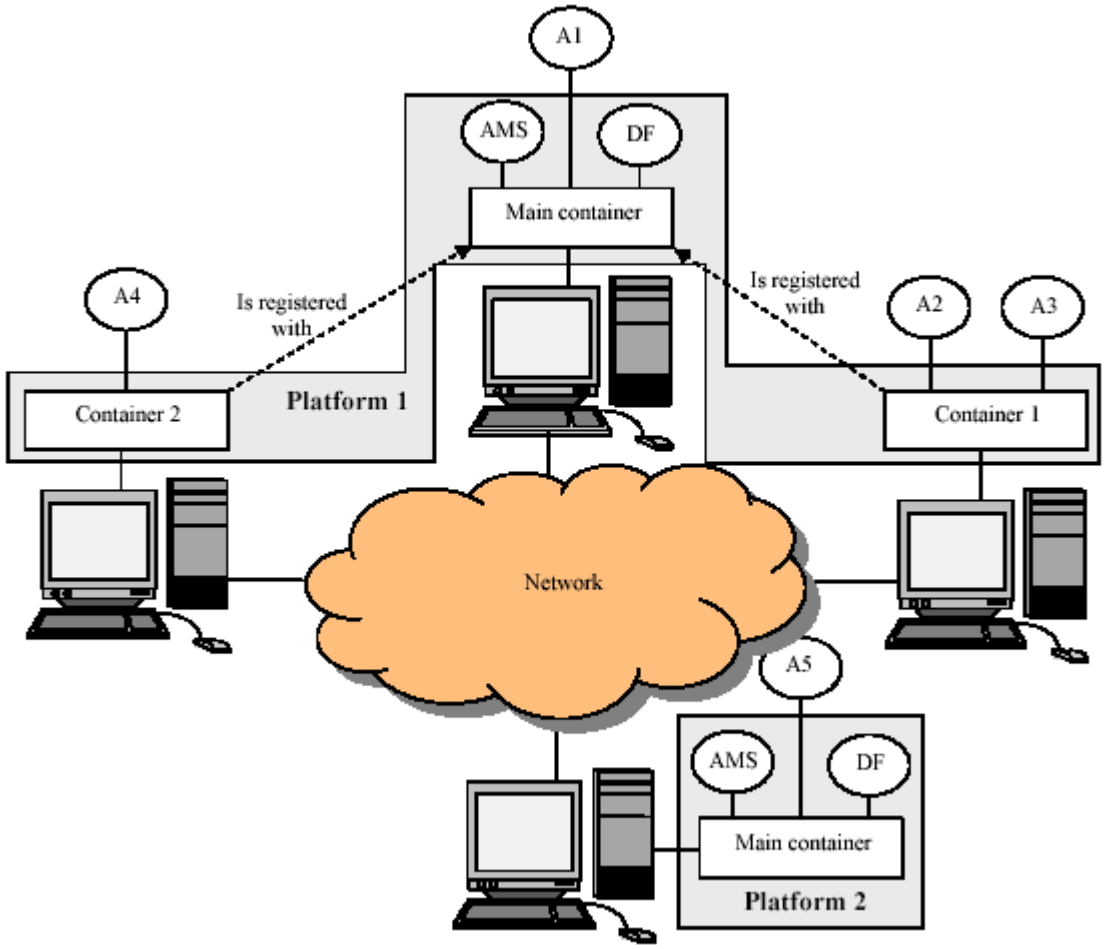

Рисунок 2 – Платформа JADE

Контейнером называется динамическая среда исполнения мультиагентных приложений, в которой находятся агенты. Каждый контейнер может содержать несколько агентов. Набор активных контейнеров называется *платформой*. Один из контейнеров всегда является главным (*Main container*), все остальные контейнеры связываются с ним и регистрируются в момент запуска. Поэтому первым контейнером при старте платформы должен быть главный, а все остальные контейнеры должны быть «обыкновенными» (т.е.

неглавными) контейнерами и должны заранее «знать», как найти главный контейнер, на котором они будут регистрироваться, т.е. должны иметь данные о хосте и порте.

Другой главный контейнер, запущенный где-либо в сети, представляет собой другую платформу, на которой могут зарегистрироваться новые обычные контейнеры. Рисунок 1 иллюстрирует эту концепцию на основе примера, показывающего две JADE-платформы, состоящие из трех и одного контейнера соответственно. JADE-агенты определяются с уникальными именами. При условии, что они знают имена других агентов, они могут общаться, независимо от их фактического местонахождения: в общем контейнере (агенты A2 и A3), в разных контейнерах на одной платформе (агенты A1 и A2), или вообще на разных платформах (A4 и A5). Пользователю не обязательно знать, как работает динамическая среда JADE, но необходимо запускать её перед началом выполнения своих агентов.

1.2.2 Агенты AMS и DF

Кроме возможности приёма регистраций от других контейнеров, главный контейнер отличается от обычного контейнера тем, что содержит два специальных агента, автоматически запускаемых одновременно с контейнером:

- *AMS* (*Agent Management System* – система управления агентами) обеспечивает службу управления агентами, которая позволяет создавать и удалять агентов, а также содержит в себе пространство имен агентов. Имя агента является уникальным и имеет следующий формат: <nickname>@<platform-name>. Зная имена друг друга, агенты могут обмениваться сообщениями как внутри контейнера и платформы, так и между различными платформами.
- *DF* (*Directory Facilitator* – менеджер директорий) представляет собой службу «желтых страниц» (*yellow pages*), где агенты могут публиковать информацию о предоставляемых ими сервисах. С помощью DF агент может находить агентов, предоставляющих необходимые ему сервисы, и вступать с ними в переговоры. Внутри одной платформы может существовать несколько DF, предоставляющих информацию о различных группах сервисов или о сервисах различных групп агентов. Использование DF описано в 4.4 Сервис «желтых страниц».

2 СРЕДА JADE ДЛЯ УПРАВЛЕНИЯ АГЕНТНЫМ ПРИЛОЖЕНИЕМ

JADE предоставляет набор инструментов, выполняющих управление агентным приложением и его отладку.

2.1 Запуск среды JADE

Для того чтобы запустить среду JADE, необходимо:

- установить на свой компьютер J2EE не ниже версии 1.5;
- установить на свой компьютер среду разработки Java-приложений IntelliJ IDEA;
- установить на свой компьютер среду разработки JADE доступной версии, например, версии 3.6.1. Для этого необходимо скопировать\загрузить файлы среды JADE на свой компьютер, например, в папку C:\Jade;
- Изменить конфигурационный параметр CLASSPATH следующим образом: в режиме командной строки ввести

```
prompt> set JADE_HOME=c:\jade
prompt> set CLASSPATH=%JADE_HOME%\lib\jade.jar;
%JADE_HOME%\lib\jadeTools.jar; %JADE_HOME%\lib\http.jar;
%JADE_HOME%\lib\iiop.jar;
%JADE_HOME%\lib\commons-codec\commons-codec-1.3.jar; %JADE_HOME%\classes
```

- Для запуска *Main Container* (с графическим интерфейсом) необходимо ввести

```
prompt> java jade.Boot –gui
```

Т.к. переменную CLASSPATH необходимо устанавливать каждый раз при перезапуске окна с командной строкой, целесообразно создать файл RunJade.bat, содержащий приведенные выше команды. Достаточно запустить файл RunJade.bat, чтоб открыть графический интерфейс RMA.

```
prompt> set JADE_HOME=c:\jade
prompt> set CLASSPATH=%JADE_HOME%\lib\jade.jar;
%JADE_HOME%\lib\jadeTools.jar; %JADE_HOME%\lib\http.jar;
%JADE_HOME%\lib\iiop.jar;
%JADE_HOME%\lib\commons-codec\commons-codec-1.3.jar; %JADE_HOME%\classes
java jade.Boot –gui
```

- скопировать на свой компьютер папку \demo. Эта папка содержит приложения *HelloWorldAgent, Order&Resource*.

2.2 Remote management agent

Remote management agent (Рисунок 3) представляет собой графическую консоль для управления мультиагентным приложением. Позволяет создавать новые контейнеры, управлять агентами, создавать сообщения и запускать средства отладки.

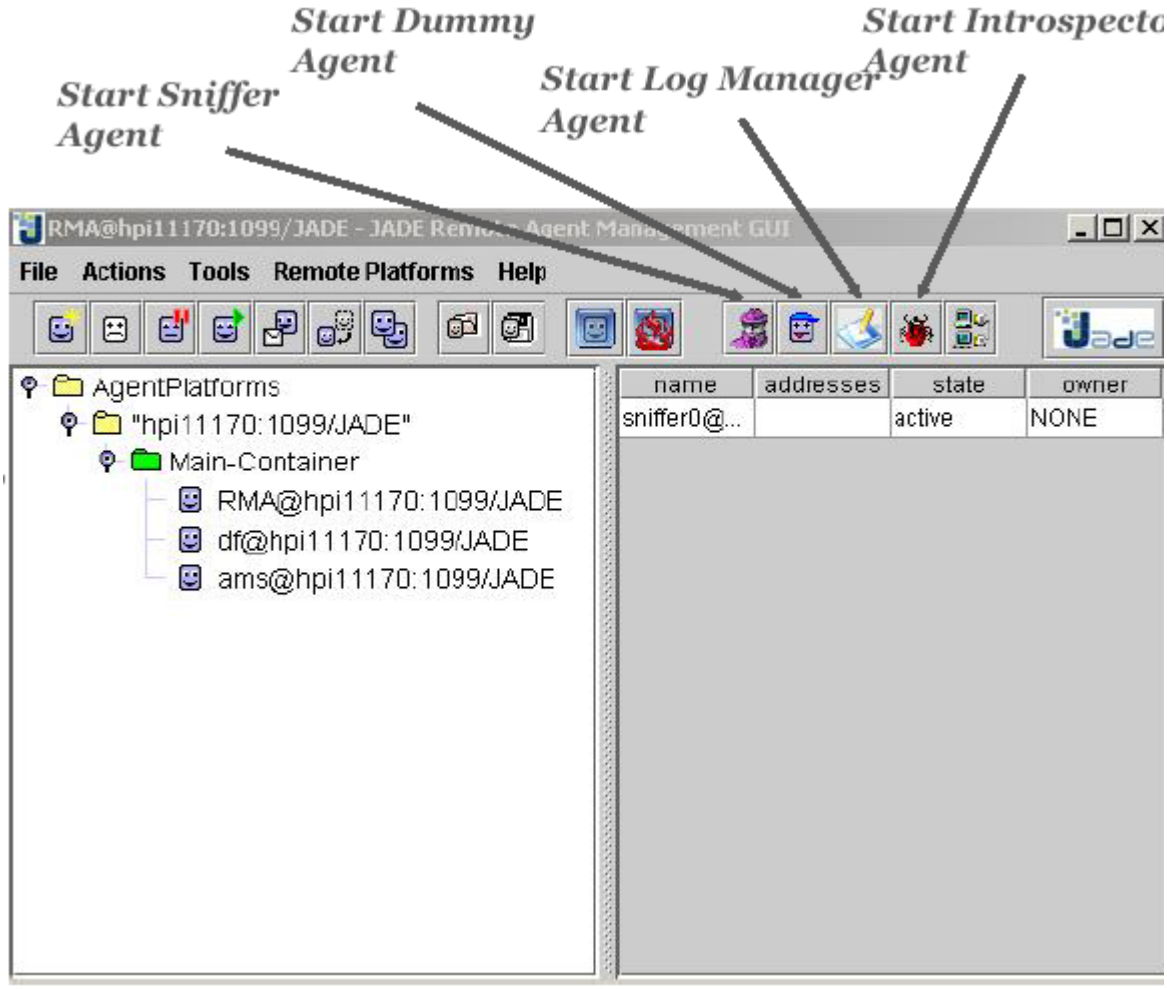

Рисунок 3 – Remote management agent

2.3 Dummy agent

Dummy agent (Рисунок 4) является графической утилитой, которая позволяет посылать и получать сообщения от имени определенного агента, а также сохранять и загружать очередь его сообщений (отправленных и полученных).

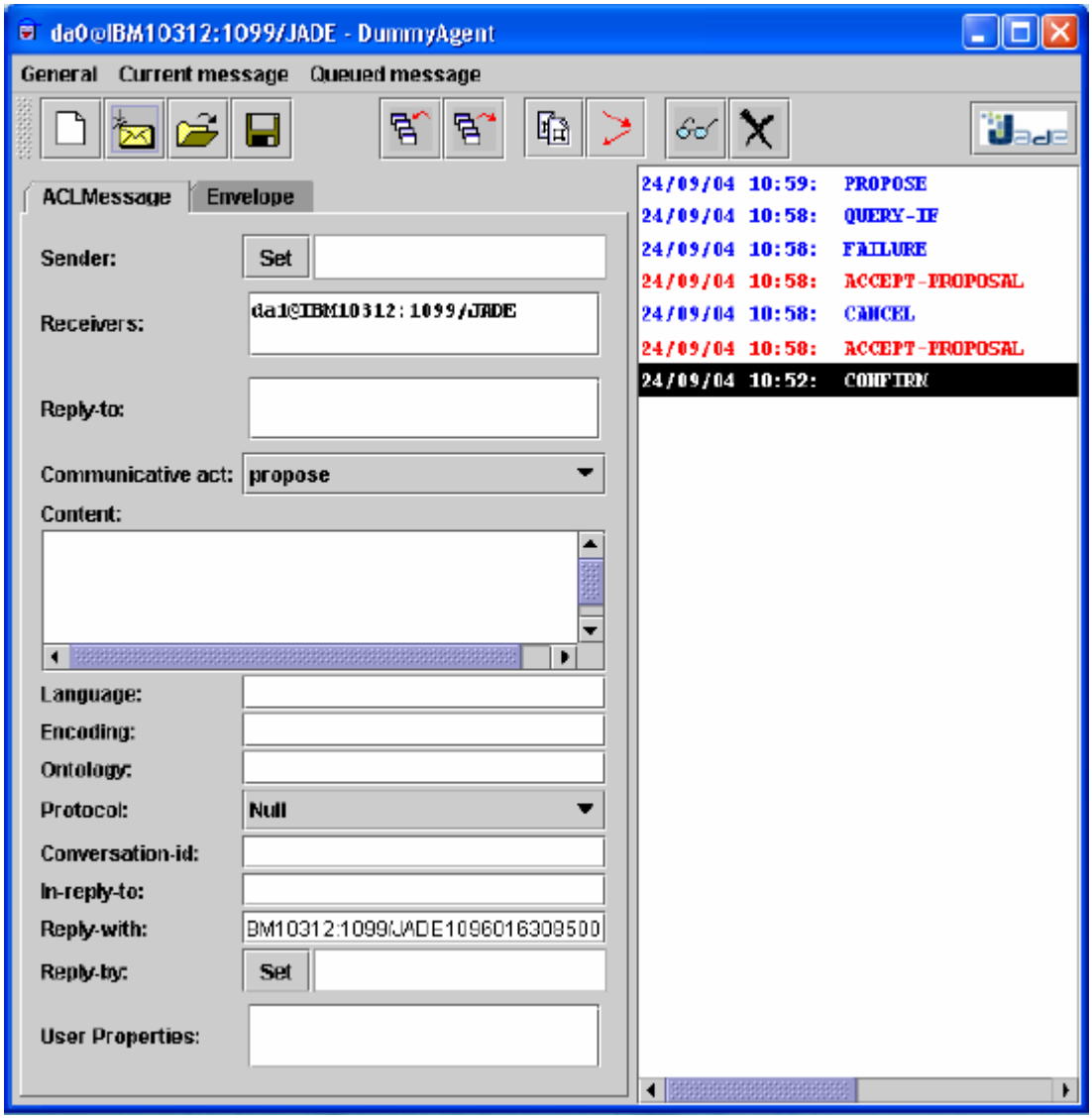

Рисунок 4 – Dummy agent

2.4 Sniffer agent

Sniffer agent (Рисунок 5) – это графическая утилита для просмотра потока сообщений между избранными агентами. Представляет обмен сообщениями в виде диаграмм последовательностей. Позволяет сохранять/загружать поток сообщений между агентами.

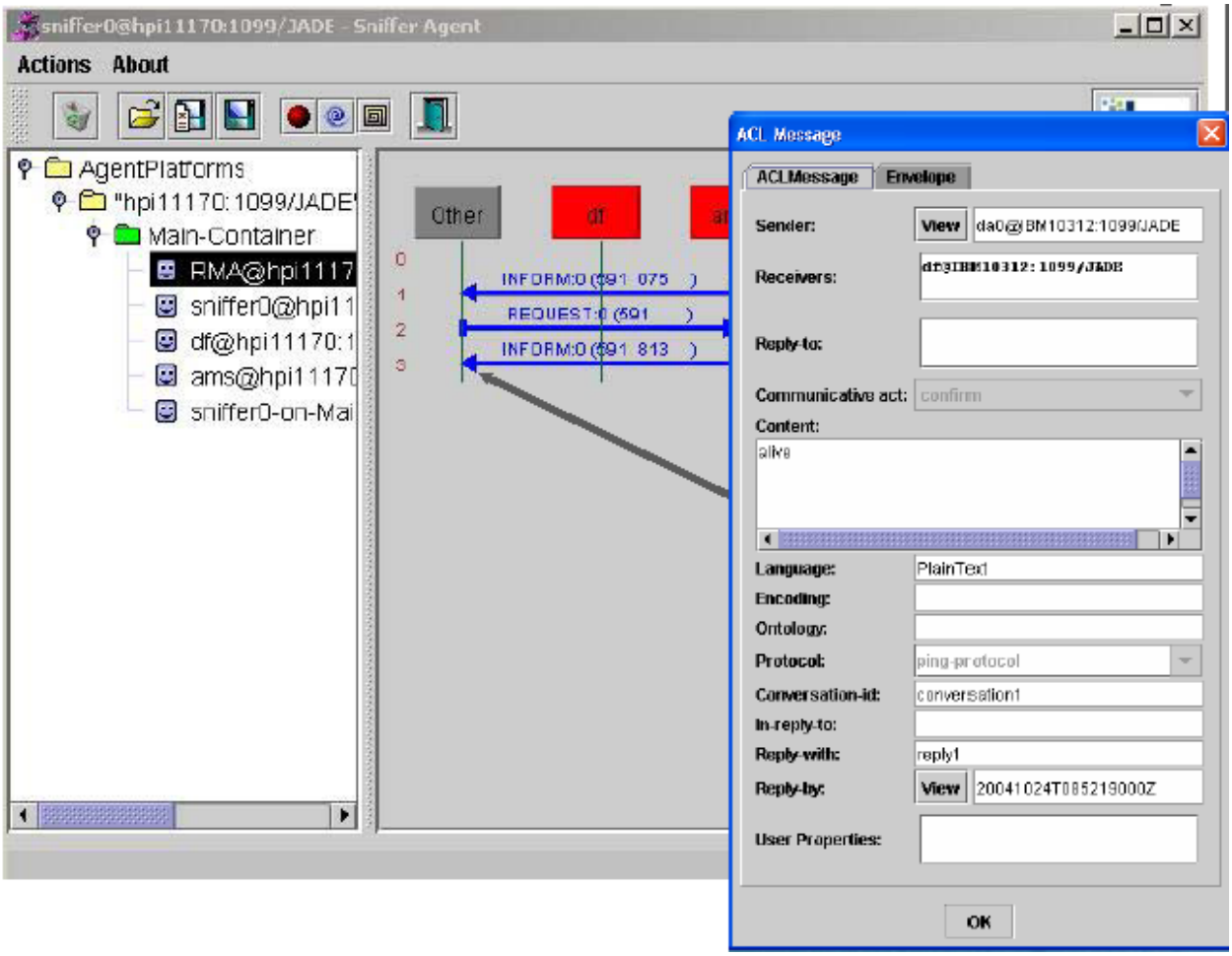

Рисунок 5 – Sniffer agent

2.5 Introspector agent

Introspector agent (Рисунок 6) – графическая утилита для просмотра внутреннего состояния агента. Позволяет контролировать жизненный цикл агента, просматривать очередь его сообщений, активные и выполненные поведения, а также запускать исполнение агента с задержками между операциями или по шагам. При этом «шагом» поведения агента считается исполнение метода *action()*, а не команда в коде языка Java.

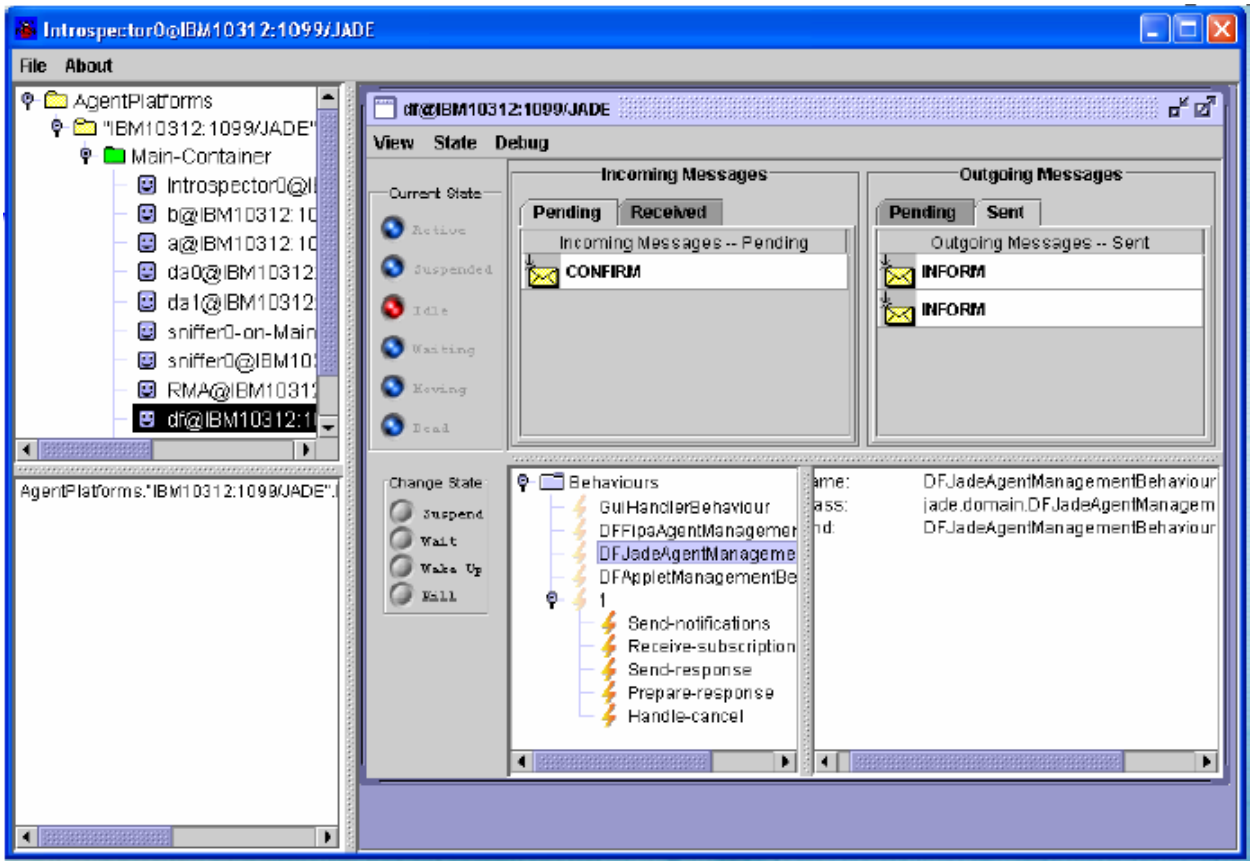

Рисунок 6 – Introspector agent

2.6 Log manager agent

Log Manager agent (Рисунок 7) – графическая утилита для отображения лога сообщений в процеесе работы агентного приложения.

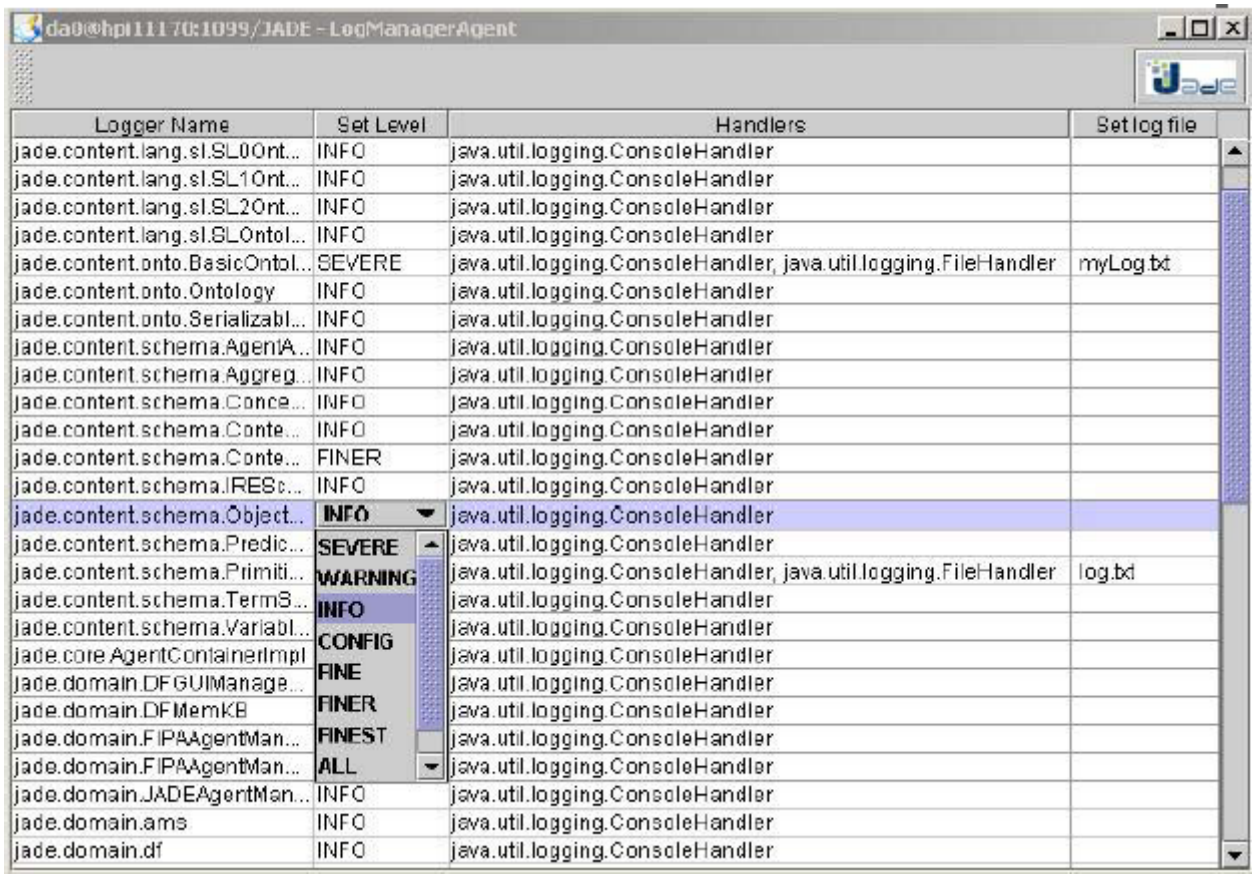

Рисунок 7 – Message log

2.7 DF GUI

DF GUI (Рисунок 8) – графическая утилита для визуализации желтых страниц. Позволяет регистрировать и удалять сервисы агентов, а также осуществлять поиск сервисов.

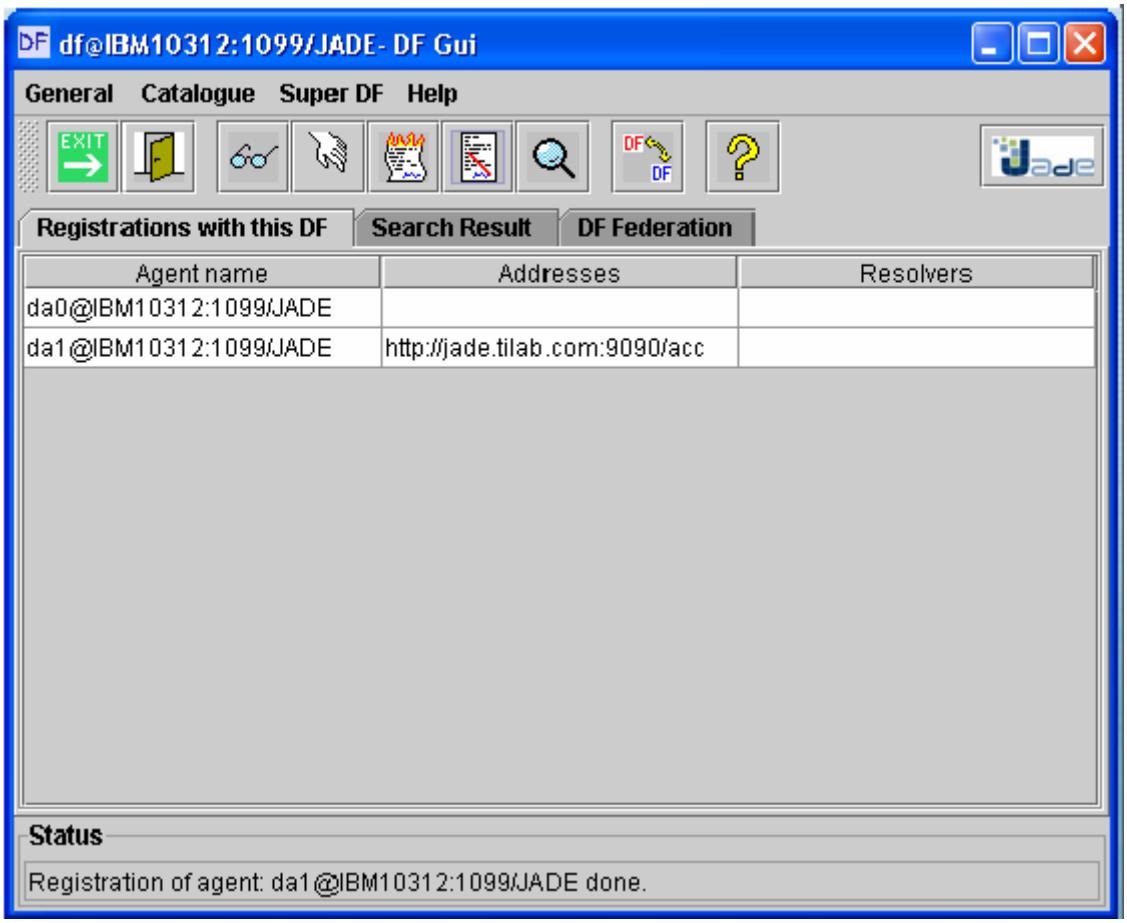

Рисунок 8 – DF GUI

3 ПРИМЕРЫ РАЗРАБОТКИ ПРОСТЫХ АГЕНТНЫХ ПРИЛОЖЕНИЙ В СРЕДЕ JADE

Рассмотрим примеры простых агентных приложений в среде JADE.

3.1 Агент HelloWorldAgent

Создадим и запустим агента HelloWorldAgent. Он является расширением типа Agent в JADE. Ниже приведен код класса агента HelloWorldAgent.java. Этот класс необходимо скомпилировать в jar-файл с таким же названием. Соответствующий файл находится в папке \demo\ HelloWorldAgent.

```
/*  Код класса HelloWorldAgent.java  */
import jade.core.Agent;
```

```
public class HelloWorldAgent extends Agent
{
        protected void setup()
{
        System.out.println("Hello World! My name is "+getLocalName());
 }
}
```

Созданый агент необходимо откомпилировать следующим образом:

```
javac -classpath <JADE-classes> HelloWorldAgent.java
```

Для того чтобы запустить откомпилированого агента на выполнение, необходимо запустить среду JADE и указать имя агента:

```
java -classpath <JADE-classes>;. jade.Boot Peter:HelloWorldAgent
```

Для создания агентного приложения используется стандартный инструмент, входящий в состав JADE, – *Remote Agent Management GUI (RAM)* (Рисунок 9). Прежде всего, необходимо выбрать контейнер с агентами (на рисунке по умолчанию выбран главный контейнер *Main-Container*).

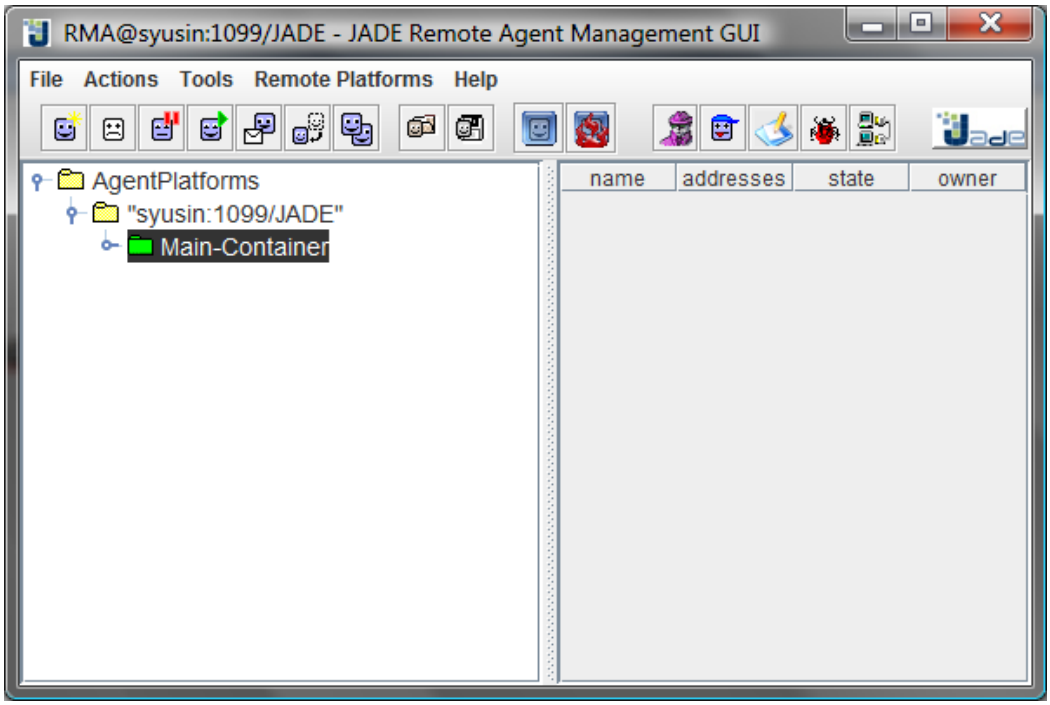

Рисунок 9 – Окно Remote Agent Management GUI (RAM)

Перед созданием агентов обязательно надо указать контейнер, иначе агенты не будут созданы (Рисунок 10).

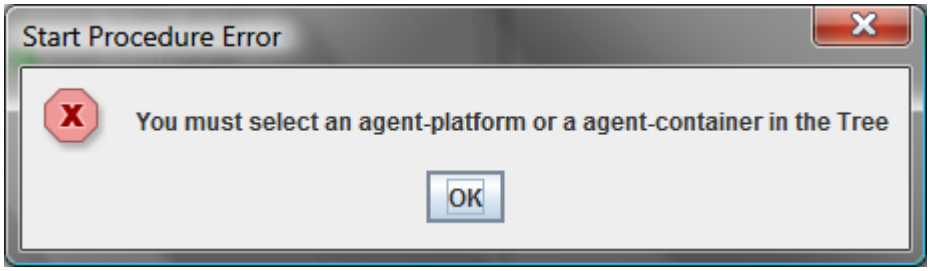

Рисунок 10 – Ошибка создания агента: не выбран контейнер

Создадим экземпляр агента типа *HelloWorldAgent* с именем *Peter*. Для этого выполним команды *Actions -> Start New Agent*. Необходимо указать имя экземпляра агента (*Peter*), а также тип создаваемого агента, который должен совпадать с именем класса агента (*HelloWorldAgent*) (Рисунок 11). После нажатия кнопки <OK> агент создается и регистрируется в системе, затем выполняются действия, описанные в методе *Setup()* агента. В данном случае агент выводит сообщение со своим именем на экран и остается в системе.

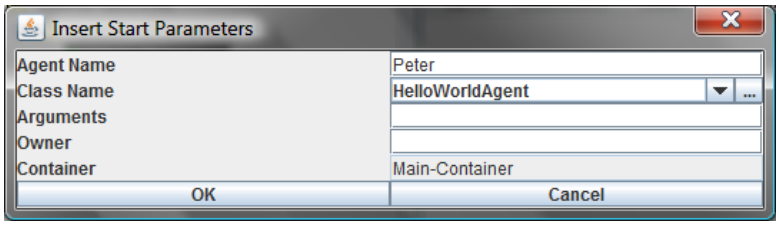

Рисунок 11 – Инициализация агента типа HelloWorldAgent

Результат действий агента отображается в окне командной строки (Рисунок 12).

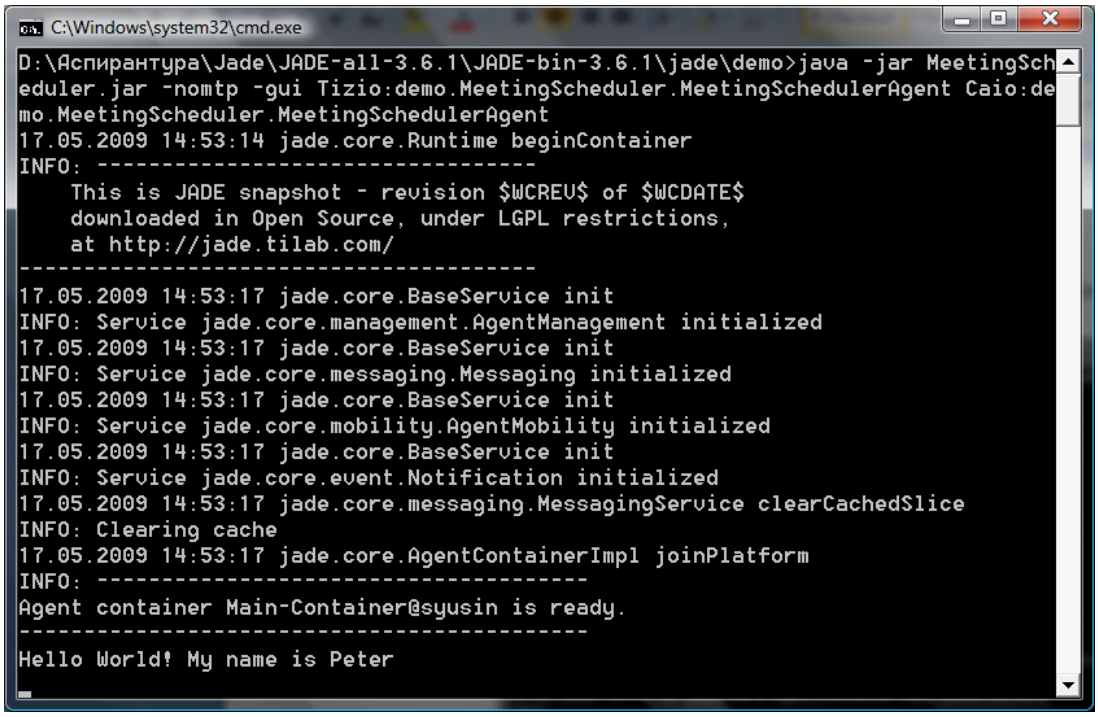

Рисунок 12 – Результат действий агента: вывод строки «Hello World! My name is Peter»

После выполнения действий агент остается в системе в активном состоянии: он присутствует в списке агентов контейнера *Main-Container* (Рисунок 13).

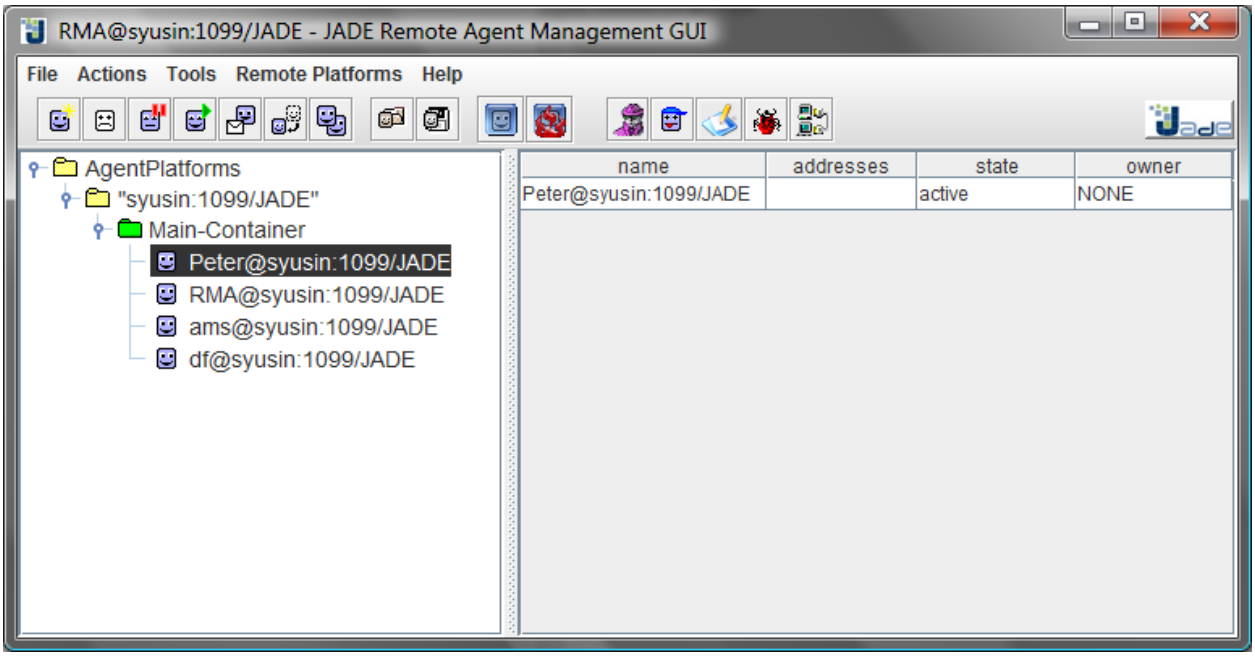

Рисунок 13 – Списокок агентов контейнера Main-Container

Для того чтобы удалить агента из системы (а также из списка агентов контейнера), в код класса агента необходимо добавить вызов метода *doDelete()*:

```
public class HelloWorldAgent extends Agent
{
 protected void setup()
{
        System.out.println("Hello World! My name is "+ getLocalName());
        // Make this agent terminate
        doDelete()
}
}
```

3.2 Приложение «Матчинг заказов и ресурсов»

Данный пример демонстрирует следующую функциональность:

- создание и регистрация в системе агентов различных типов;
- рассылка сообщений между агентами;
- осуществление матчинга по заданным условиям;
- вывод результата матчинга.

В качестве модели данных используется отношение Заказ – Ресурс, наиболее часто встречающееся в сфере производства и логистики. Ресурс и заказ имеют активных агентов, которые осуществляют матчинг (поиск

соответствия) для создания связей между заказом и подходящим ресурсом.

Агент заказа рассылает сообщения всем агентам ресурсов, зарегистрированным в системе. Сообщение содержит тип (информационное), ссылку на онтологию (в виде строки-названия), а также контент – название города, куда необходимо доставить заказ. Каждый ресурс может доставить заказ только в определенный город (пункт доставки), название которого задается пользователем при создании агента ресурса. Далее происходит матчинг – выполнить заказ могут только те ресурсы, пункт доставки которых совпадает с пунктом, указанным в сообщении, полученном от агента заказа.

3.2.1 Описание классов агентов заказа и ресурсов

Для реализации этой модели были определены два типа агентов – агент заказа (*OrderAgent)* и агент ресурса (*ResourceAgent*). Они являются расширением типа *Agent* в JADE. Ниже приведен код класса агента заказа *OrderAgent.java* и код класса агента ресурса *ResourceAgent.java*. Эти классы необходимо скомпилировать в отдельные jar-файлы с такими же названиями. Соответствующие файлы находятся в папке *\demo\Order&Resource*. Более подробно процесс разработки классов агентов будет показан в главе **4 РАЗРАБОТКА АГЕНТНОГО ПРИЛОЖЕНИЯ «ТОРГОВЛЯ КНИГАМИ» НА ОСНОВЕ ПЛАТФОРМЫ JADE**.

/* **Код класса OrderAgent.java** */

```
import jade.core.Agent;
import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.*;

public class OrderAgent extends Agent {

  protected void setup() {
          addBehaviour(new SimpleBehaviour(this)
     {
      private boolean finished = false;
      public String DestinationPoint = "Moscow";

      AID[] resources = {new AID("Resource1@syusin:1099/JADE"),
                         new AID("Resource2@syusin:1099/JADE"),
                         new AID("Resource3@syusin:1099/JADE"),
                         new AID("Resource4@syusin:1099/JADE"),
                         new AID("Resource5@syusin:1099/JADE")};

      public void action()
      {
         System.out.println(getLocalName() +  " is active");
         ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
         msg.setOntology("TestOntology");
         msg.setContent(DestinationPoint);
         for (int i=0; i < resources.length; i++)
            msg.addReceiver(resources[i]);
         send(msg);
         finished = true;
```

```
      }

      public boolean done()
      {
         return finished;
      }
     };
    }
}
```

Примечание: у агента заказа определен пункт доставки – Moscow.

/* Код класса ResourceAgent.java */

```
import jade.core.Agent;
import jade.core.behaviours.*;
import jade.lang.acl.*;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class ResourceAgent extends Agent {

  protected void setup() {
           addBehaviour(new SimpleBehaviour(this)
     {
      private boolean finished = false;
      public String DestinationPoint;

      public void action()
      {
         System.out.println(getLocalName() +  " is active");
         System.out.println(getLocalName() +  " input destination point:");
         BufferedReader buff = new BufferedReader(new InputStreamReader(System.in));
         try
         {
            DestinationPoint =  buff.readLine().toUpperCase();
         }
         catch (IOException E)  {}

         MessageTemplate m1 = MessageTemplate.MatchPerformative(ACLMessage.INFORM);
         MessageTemplate m2 = MessageTemplate.MatchOntology("TestOntology");
         MessageTemplate m3 = MessageTemplate.and(m1, m2);
         ACLMessage msg = blockingReceive(m3, 120000);
         if (msg != null)
         {
           System.out.println(getLocalName() +  ": message from " + msg.getSender().getLocalName() + " was
           received");
            if (DestinationPoint.equals(msg.getContent().toUpperCase()))
               System.out.println(getLocalName() +  ": order was accepted");
            else
               System.out.println(getLocalName() +  ": order was rejected");
         }
         else
            System.out.println(getLocalName() +  ": empty message was received");
         finished = true;
      }

      public boolean done()
      {
         return finished;
      }
     };
```

```
    }
}
```

Агент ресурса при создании и инициализации запрашивает у пользователя пункт доставки, а затем в течение определенного времени ждет поступления сообщения с определенными параметрами (сообщения фильтруются по типу и по ссылке на онтологию).

Примечание: метод *blockingReceive(m3, 120000)* использован для указания интервала времени (в миллисекундах), в течение которого агент должен ожидать входящее сообщение. После получения сообщения агент ресурса проверяет, совпадает ли его пункт доставки с пунктом, указанным в сообщении. Если совпадает, то агент принимает заказ, в противном случае, заказ отклоняется.

3.2.2 Создание агентного приложения «Матчинг заказов и ресурсов»

После запуска *Remote Agent Management GUI* открывается соответствующее окно (Рисунок 14).

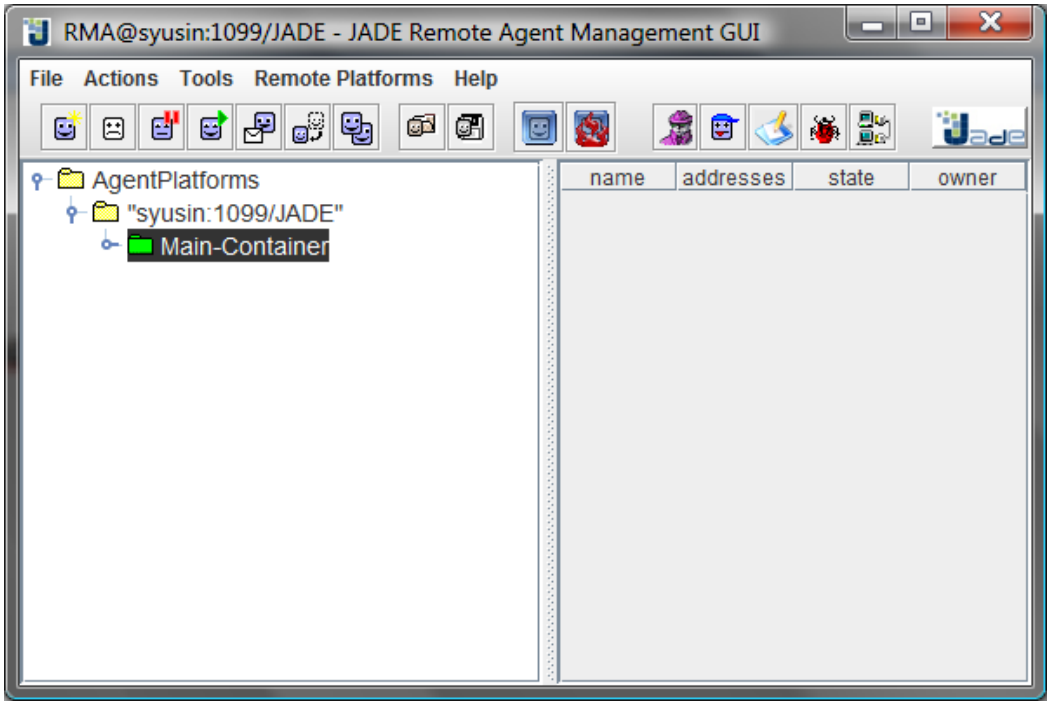

Рисунок 14 – Окно Remote Agent Management GUI (RAM)

В окне RAM следует создать одного агента заказа и необходимое количество агентов ресурсов.

Чтобы создать экземпляр агента типа *ResourceAgent* с именем *Resource1*, необходимо выполнить команды *Actions -> Start New Agent*. Далее следует указать имя экземпляра (*Resource1*), а также тип создаваемого агента, который должен совпадать с именем jar-файла (Рисунок 15). После нажатия кнопки <OK> агент создается и регистрируется в системе, при этом его статус

устанавливается в *Active*. Характеристики агента, выбранного в списке агентов данного контейнера, отображаются в правом окне RAM (Рисунок 16).

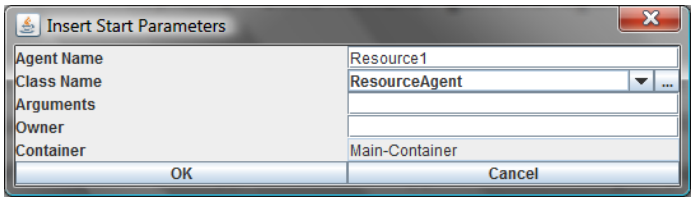

Рисунок 15 – Инициализация агента

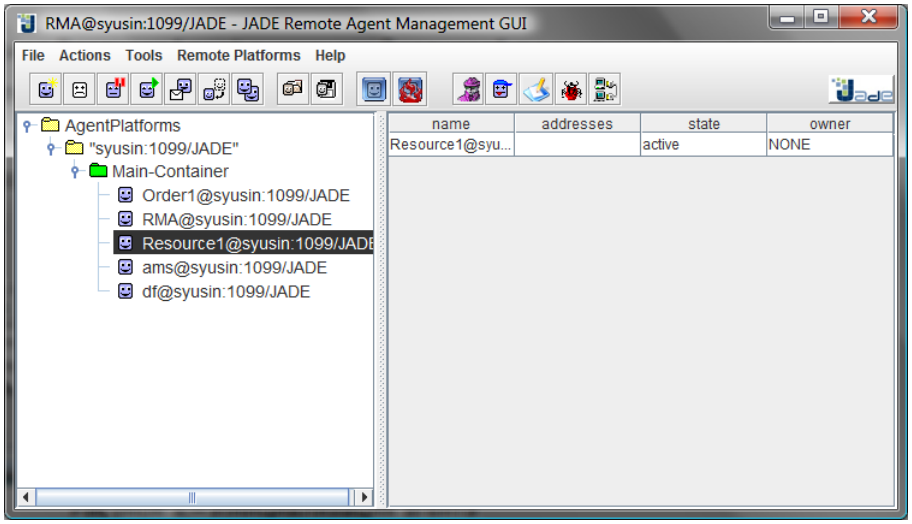

Рисунок 16 – Просмотр состояния агента в RAM

При создании агента ресурса в открывшемся окне командной строки пользователю будет предложено ввести название пункта доставки для данного ресурса (Рисунок 17). Можно вводить любое значение, но следует помнить, что заказ должен быть доставлен в пункт Moscow. Поэтому для успешного осуществления матчинга необходимо, чтобы хотя бы для одного агента ресурса был определен пункт доставки Moscow.

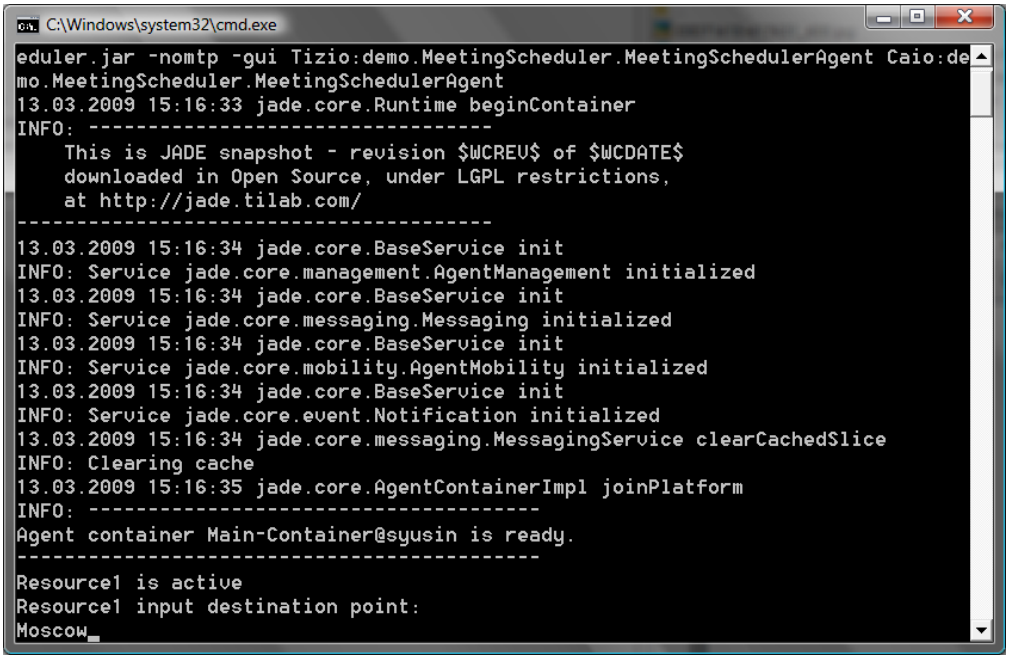

Рисунок 17 – Ввод наименования пункта доставки для агента ресурса

Создать еще четырех агентов ресурсов с именами *Resource2*, *Resource3*, *Resource4*, *Resource5*. Для каждого из них задать названия пунктов доставки (Рисунок 18). Таким образом, создается ресурсная база, на основании которой можно выбирать наиболее подходящий вариант для выполнения заказа. Для агентов ресурсов программно установлено время ожидания сообщения – 2 минуты. По истечении 2-х минут агент просматривает очередь сообщений и, если она пуста, выводится сообщение *ResourceX: empty message was received*. Поэтому необходимо инициализировать всех агентов ресурсов и агента заказа в течение 2-х минут.

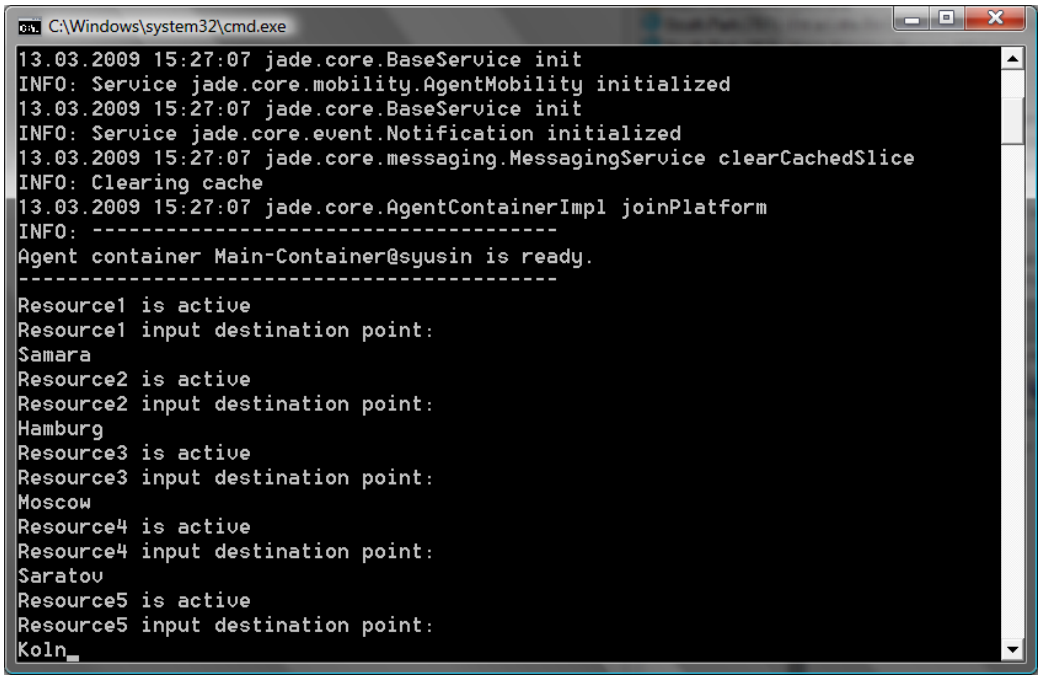

Рисунок 18 – Инициализация пяти агентов ресурсов

Создание агента заказа типа *OrderAgent* показано на Рисунок 19.

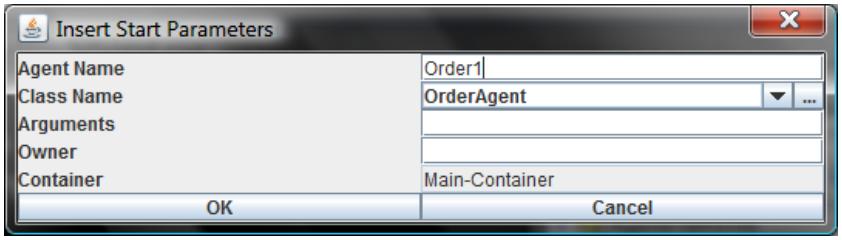

Рисунок 19 – Инициализация агента заказа

Т.к. в классе агента *OrderAgent.java* указан пункт доставки Moscow, агент заказа будет выбирать ресурс с таким же пунктом доставки.

После создания агента заказа он сразу же рассылает сообщения агентам ресурсов. Каждый агент реагирует на это сообщение, при этом возможны два варианта ответа: подтверждение возможности выполнения заказа или отклонение его. Если несколько агентов смогут принять заказ, то все они ответят положительно. Ответы агентов ресурсов приведены на Рисунок 20. Сообщение типа *ResourceX: message from Order1 was received* означает, что агент ресурса получил сообщение от агента заказа, а сообщения типа *ResourceX: order was accepted* или *ResourceX: order was rejected* означают соответственно прием или отклонение заказа. Из полученного списка сообщений следует, что только *Resource3* подтвердил возможность выполнения заказа, т.к. только у этого ресурса пункт доставки совпадает с пунктом доставки, указанным в заказе (Рисунок 20). Остальные агенты ресурсов отклонили заказ. Следует отметить, что ответы ресурсов поступали не в том порядке, в котором они были инициализированы в системе. Следовательно, пересылка сообщений и ответов между агентами выполнялась в случайном порядке.

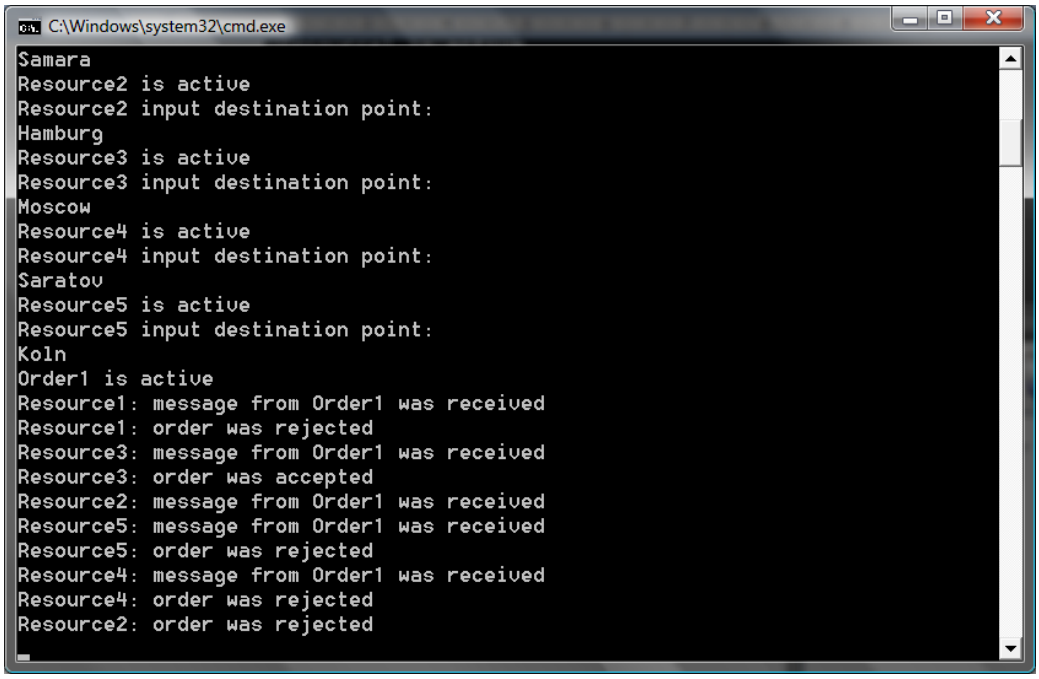

Рисунок 20 – Ответы агентов ресурсов на сообщение агента заказа

4 РАЗРАБОТКА АГЕНТНОГО ПРИЛОЖЕНИЯ «ТОРГОВЛЯ КНИГАМИ» НА ОСНОВЕ ПЛАТФОРМЫ JADE

Основные этапы разработки мультиагентного приложения с использованием платформы JADE рассмотрим на примере приложения «Торговля книгами» [10]. В приложении действуют агенты-книготорговцы и агенты-покупатели, приобретающие книги по поручению своих владельцев.

Каждый агент-покупатель получает название книги, которую необходимо купить («требуемая книга»), в качестве аргумента командной строки и периодически запрашивает предложения у всех известных ему агентов-продавцов. Как только предложение получено, агент-покупатель принимает его и выдает заказ на покупку. Если несколько агентов-продавцов предоставляют предложение агенту-покупателю, он выбирает из них лучшее (с самой низкой ценой). После покупки агент-покупатель завершает свою работу.

Каждый агент-продавец имеет минимальный интерфейс, с помощью которого пользователь может добавить названия новых книг и цены на них в свой локальный каталог книг для продажи. Агент-продавец непрерывно проверяет поступление запросов от агентов-покупателей. При получении запроса на книгу каждый агент-продавец проверяет свой локальный каталог и, если запрашиваемая книга находится в его каталоге, отвечает на запрос, сообщая агенту-покупателю цену. После выполнения заказа купленная книга автоматически удаляется из каталога.

Коды классов, реализующих данное приложение, находятся в папке *\jade\src\examples\bookTraiding*.

4.1 Класс «Агент»

4.1.1 Создание агента

Агент JADE создается с помощью определения класса, наследуемого от класса *jade.core.Agent*, и реализации метода *setup()*:

```
import jade.core.Agent;
public class BookBuyerAgent extends Agent {
protected void setup() {
// Printout a welcome message
System.out.println("Hello! Buyer-agent "+getAID().getName()+" is ready.");
}
}
```

Метод *setup()* выполняет инициализацию агента. Далее агент функционирует в рамках своего поведения (4.2 Класс «Поведение агента»).

4.1.2 Идентификация агента

Каждый агент имеет определенный «идентификатор агента», представляющий собой экземпляр класса *jade.core.AID*. Метод *getAID()* класса *Agent* позволяет получить идентификатор агента. Объект *AID* содержит глобальное уникальное имя, а также адрес. Идентификатор агента JADE имеет структуру: *<имя_агента>@<имя_платформы>*, поэтому агент с именем *Peter*, находящийся на платформе *P1*, будет иметь глобальное уникальное имя *Peter@P1*. Адрес, содержащийся в *AID*, – это адрес платформы, где находится агент. Этот адрес используется только в тех случаях, когда агент должен общаться с другими агентами, которые находятся на другой платформе. *AID* агента может быть получен по его идентификатору следующим образом:

```
String nickname = "Peter";
AID id = new AID(nickname, AID.ISLOCALNAME);
```

Константа *ISLOCALNAME* указывает, что первый параметр представляет собой локальный идентификатор на платформе, а не глобальный уникальный идентификатор этого агента.

4.1.3 Запуск агента

Созданый агент необходимо откомпилировать следующим образом:

```
javac -classpath <JADE-classes> BookBuyerAgent.java
```

Для того чтобы запустить откомпилированого агента на выполнение, необходимо запустить среду JADE и указать имя агента:

```
java -classpath <JADE-classes>;. jade.Boot buyer:BookBuyerAgent
```

Будет получен следующий результат:

```
C:\jade>java -classpath <JADE-classes> jade.Boot buyer:BookBuyerAgent
5-mag-2008 11.06.45 jade.core.Runtime beginContainer
INFO: ---------------------------------
This is JADE snapshot - revision 5995 of 2007/09/03 09:45:22
downloaded in Open Source, under LGPL restrictions,
at http://jade.tilab.com/
----------------------------------------
5-mag-2008 11.06.51 jade.core.BaseService init
INFO: Service jade.core.management.AgentManagement initialized
5-mag-2008 11.06.51 jade.core.BaseService init
INFO: Service jade.core.messaging.Messaging initialized
5-mag-2008 11.06.52 jade.core.BaseService init
INFO: Service jade.core.mobility.AgentMobility initialized
5-mag-2008 11.06.52 jade.core.BaseService init
INFO: Service jade.core.event.Notification initialized
5-mag-2008 11.06.52 jade.core.messaging.MessagingService clearCachedSlice
INFO: Clearing cache
```

```
5-mag-2008 11.06.53 jade.mtp.http.HTTPServer <init>
INFO: HTTP-MTP Using XML parser
com.sun.org.apache.xerces.internal.parsers.SAXParser
5-mag-2008 11.06.54 jade.core.messaging.MessagingService boot
INFO: MTP addresses:
http://NBNT2004130496.telecomitalia.local:7778/acc
5-mag-2008 11.06.54 jade.core.AgentContainerImpl joinPlatform
INFO: ----------------------------------------
Agent container Main-Container@NBNT2004130496 is ready.
------------------------------------------
      Hello! Buyer-agent buyer@NBNT2004130496:1099/JADE is ready.
```

При запуске среды JADE выполняется инициализация ядра сервисов JADE. Процесс завершается сообщением о том, что контейнер с именем *«Main-Container»* готов к работе. Когда среда JADE запускает пользовательского агента, он выводит своё приветствие с именем *«Buyer»*, указанным при создании агента. Имя платформы *«NBNT2004130496: 1099/JADE»* назначается автоматически в соответствии с адресом хоста и порта, на котором работает среда JADE.

4.1.4 Завершение работы агента

Даже если пользовательский агент ничего не делает после вывода приветствия, он продолжает работать. Чтобы завершить его работу, необходимо вызвать метод *doDelete ()*. Непосредственно перед завершением работы агента вызывается метод *takeDown ()*, который выполняет операции по очистке агента.

4.1.5 Передача аргументов агенту

Агенты могут получать при запуске аргументы, заданные в командной строке. Эти аргументы могут быть получены как массив *Object* с помощью метода *getArguments ()* класса *Agent*. Агент *BookBuyerAgent* должен получить в качестве аргумента командной строки название книги, которую необходимо купить. Для достижения этой цели изменим агента (заметим, что список известных агентов-продавцов для отправки запросов фиксирован. В 4.4.2.2 Поиск сервисов будет показано, как находить агентов-продавцов динамически).

```
import jade.core.Agent;
import jade.core.AID;
public class BookBuyerAgent extends Agent {
// The title of the book to buy
private String targetBookTitle;
// The list of known seller agents
private AID[] sellerAgents = {new AID("seller1", AID.ISLOCALNAME),
new AID("seller2", AID.ISLOCALNAME)};
// Put agent initializations here
protected void setup() {
// Printout a welcome message
System.out.println("Hello! Buyer-agent "+getAID().getName()+" is ready.");
// Get the title of the book to buy as a start-up argument
```

```
Object[] args = getArguments();
if (args != null && args.length > 0) {
targetBookTitle = (String) args[0];
System.out.println("Trying to buy "+targetBookTitle);
}
else {
// Make the agent terminate immediately
System.out.println("No book title specified");
doDelete();
}
}
// Put agent clean-up operations here
protected void takeDown() {
// Printout a dismissal message
System.out.println("Buyer-agent "+getAID().getName()+" terminating.");
}
}
```

Аргументы командной строки задаются внутри скобок и разделяются пробелами.

```
C:\jade>java jade.Boot buyer:BookBuyerAgent(The-Lord-of-the-rings)
...
...
5-mag-2008 11.11.00 jade.core.AgentContainerImpl joinPlatform
INFO: ----------------------------------------
Agent container Main-Container@NBNT2004130496 is ready.
--------------------------------------------
Hello! Buyer-agent buyer@NBNT2004130496:1099/JADE is ready.
      Trying to buy The-Lord-of-the-Rings
```

4.2 Класс «Поведение агента»

Активность агента выполняется в рамках *«поведения»* (*режима работы*). Каждый агент исполняется в отдельном потоке Java. Логика агента строится из комбинации режимов.

Поведение представляет собой последовательность действий, которые должен выполнить агент (аналогично методу Java-класса), и реализуется как объект класса, наследуемого от *jade.core.behaviours.Behaviour*. Поведение агента определяется с помощью метода *addBehaviour ()* класса *Agent*. Поведение может быть добавлено в любой момент: при запуске агента (в методе *Setup ()*) или внутри другого режима работы.

Каждый класс, наследуемый от класса *Behaviour*, должен переопределить метод *action()*, который фактически определяет операции, задающие поведение агента, а также метод *done()*, который возвращает булево значение, указывающее на завершение того или иного поведения, которое должно быть удалено из пула режимов работы агента.

4.2.1 Планирование и исполнение режимов работы агента

Агент может выполнять одновременно несколько моделей поведения. Однако важно заметить, что планирование режимов агента является не упреждающим (как в потоках Java), а кооперативным. Это означает, что, когда поведение планируется к исполнению, метод *action()* вызван и работает до тех пор, пока режим завершится. Поэтому программист должен определять, когда агент переключается от выполнения одного режима к выполнению следующего. Такой подход имеет ряд преимуществ:

- поддерживает работу с одним потоком Java для агента (что является весьма важным в условиях ограниченных ресурсов, какими обладают, например, сотовые телефоны);
- обеспечивает более быстрое переключение между режимами работы, чем переключение между потоками Java;
- устраняет проблемы синхронизации параллельных режимов работы для доступа к одним и тем же ресурсам (что ускоряет производительность), т.к. все режимы выполняются одним и тем же потоком Java;
- позволяет сохранять состояние агента в постоянном хранилище для последующего возобновления (сохранямость агента) или передачи его другому контейнеру для удаленного исполнения (мобильность агента).

Жизненный цикл агента представлен на Рисунок 21. При создании агента вызывается метод *setup(),* в котором происходит инициализация агента. Далее в бесконечном цикле выбирается следующий режим из пула активных и исполняется. Затем, в зависимости от типа режима, он либо удаляется из пула активных, либо остается в нем и выполняется снова, когда до него дойдет очередь. При отсутствии активных режимов (например, если агент ждет сообщение) поток агента «засыпает» (что позволяет экономить ресурсы процессора). При удалении агента вызывается метод *takeDown()*.

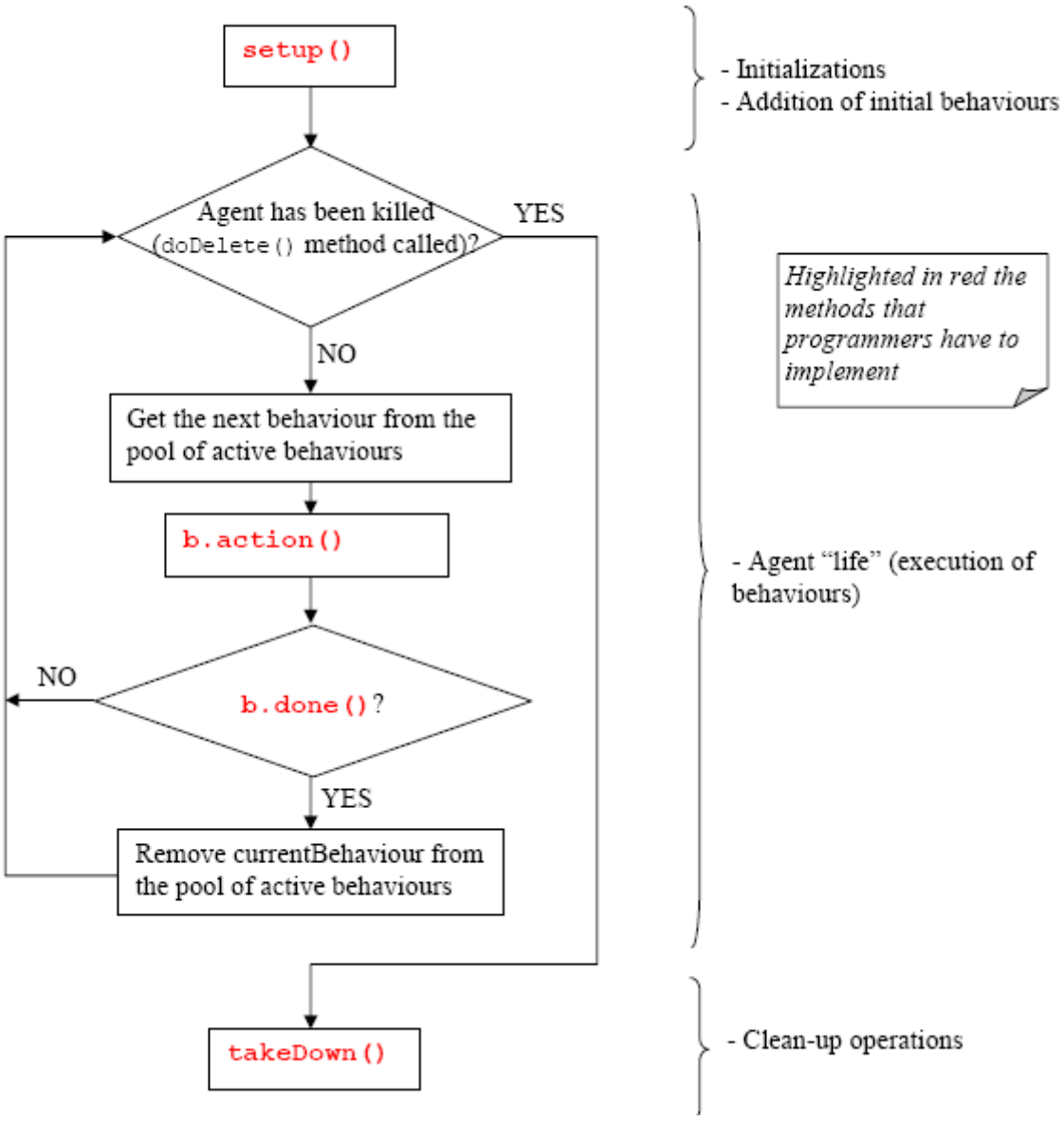

Рисунок 21 - Жизненный цикл агента

В соответствии с описанным механизмом планирования, режим, представленный ниже, не позволяет запустить другой режим, т.к. его метод *action()* никогда не завершается.

```
public class OverbearingBehaviour extends Behaviour {
public void action() {
while (true) {
// do something
}
}
public boolean done() {
return true;
}
}
```

4.2.2 Типы режимов агента

Различают три типа поведения агентов.

1) Одноразовый (*one-shot*) режим, у которого метод *action()* выполняется только один раз, после чего режим становится неактивным. *jade.core.behaviours.OneShotBehaviour* уже реализует метод *done()*, возвращающий значение *true*, и может быть легко модифицирован для реализации одноразового поведения.

```
public class MyOneShotBehaviour extends OneShotBehaviour {
public void action() {
// perform operation X
}
}
```

Операция X будет выполнена только один раз.

2) Циклический (*cyclic*) режим, который никогда не завершается и всегда продолжает оставаться активным. Метод *action()* выполняет одинаковые операции каждый раз, когда он вызывается. *jade.core.behaviours.CyclicBehaviour* уже реализует метод *done()*, возвращающий значение *false*, и может быть легко модифицирован для реализации циклического поведения.

```
public class MyCyclicBehaviour extends CyclicBehaviour {
public void action() {
// perform operation Y
}
}
```

Операция Y выполняется многократно, до завершения работы агента, выполняющего циклическое поведение.

3) Общий (*generic*) режим, который выполняет различные операции в зависимости от значения некоторой переменной. Режим остается активным до тех пор, пока выполняется условие, заданное в методе *done()*.

```
public class MyThreeStepBehaviour extends Behaviour {
private int step = 0;
public void action() {
switch (step) {
case 0:
// perform operation X
step++;
break;
case 1:
// perform operation Y
step++;
break;
case 2:
// perform operation Z
step++;
break;
}
```

```
}
public boolean done() {
return step == 3;
}
}
```

Операции X, Y и Z выполняются одна за другой, а затем поведение завершается.

JADE предусматривает возможность комбинации простых режимов работы агента для создания более сложных режимов.

4.2.3 Планирование операций в заданных временных точках

JADE предлагает два готовых класса (в пакете *jade.core.behaviours*), с помощью которых можно легко реализовывать режимы работы, выполняющие определенные операции в заданные моменты времени.

1) Режим *WakerBehaviour*, у которого методы *action()* и *done()* реализованы таким образом, что абстрактный метод *handleElapsedTimeout ()* выполняется по истечении заданного интервала времени (указанного в конструкторе). После исполнения метода *handleElapsedTimeout ()* поведение завершается.

```
public class MyAgent extends Agent {
protected void setup() {
System.out.println("Adding waker behaviour");
addBehaviour(new WakerBehaviour(this, 10000) {
protected void handleElapsedTimeout() {
// perform operation X
}
} );
}
}
```

Операция X выполняется через 10 секунд после вывода на экран сообщения «Adding waker behaviour».

2) Режим *TickerBehaviour*, у которого методы *action()* и *done()* реализованы таким образом, что абстрактный метод *onTick()* выполняется многократно, через определенный интервал времени (указанный в конструкторе). Режим *TickerBehaviour* никогда не завершается.

```
public class MyAgent extends Agent {
protected void setup() {
addBehaviour(new TickerBehaviour(this, 10000) {
protected void onTick() {
// perform operation Y
}
} );
}
}
```

Операция Y выполняется периодически каждые 10 секунд.

4.2.4 Режимы работы агентов в приложении «Торговля книгами»

В соответствии с описанием основных типов поведения агентов, проанализируем поведение агентов *Book-buyer* и *Book-seller* в приложении «Торговля книгами».

4.2.4.1 Поведение агента Book-buyer

Агент *Book-buyer* должен периодически запрашивать у агента продавца книгу, которую ему было поручено купить. Этого можно достичь, если модифицировать режим *TickerBehaviour* так, что каждый «тик» добавляет другой режим, который фактически выполняет запрос к агенту-продавцу. Метод *setup()* класса *BookBuyerAgent* необходимо изменить следующим образом.

```
protected void setup() {
// Printout a welcome message
System.out.println("Hello! Buyer-agent "+getAID().getName()+" is ready.");
// Get the title of the book to buy as a start-up argument
Object[] args = getArguments();
if (args != null && args.length > 0) {
targetBookTitle = (String) args[0];
System.out.println("Trying to buy "+targetBookTitle);
// Add a TickerBehaviour that schedules a request to seller agents every minute
addBehaviour(new TickerBehaviour(this, 60000) {
protected void onTick() {
myAgent.addBehaviour(new RequestPerformer());
}
} );
}
else {
// Make the agent terminate
System.out.println("No target book title specified");
doDelete();
}
}
```

Обратите внимание на использование защищённой переменной *myAgent*: каждый режим содержит указатель на агента, который его исполняет.

Режим *RequestPerformer*, фактически выполняющий запрос к агенту-продавцу, будет описан в 4.3 Класс, где будет обсуждаться взаимодействие между агентами.

4.2.4.2 Поведение агента Book-seller

Каждый агент *Book-seller* ожидает прихода запросов от агентов-покупателей и обслуживает их. Запросы могут быть различных типов: запрос о представлении предложения на книгу и запрос на выполнение заказа. Для реализации запросов различных типов агенту *Book-seller* необходимо задать два циклических поведения: одно для обслуживания запросов на предложение, а

другое для обслуживания заказов. Принципы обнаружения и обслуживания входящих запросов от агентов-покупателей, описано в 4.3 Класс «взаимодействие между агентами», где будет обсуждаться взаимодействие между агентами. Кроме того, необходимо задать агенту *Book-seller* одноразовое поведение, реализующее обновление каталога книг, доступных для продажи в тех случаях, когда пользователь добавляет новую книгу из GUI. Класс *BookSellerAgent* может быть реализован следующим образом (классы *OfferRequestsServer* и *PurchaseOrdersServer* будут описаны в 4.3 Класс «взаимодействие между агентами»).

```
import jade.core.Agent;
import jade.core.behaviours.*;
import java.util.*;
public class BookSellerAgent extends Agent {
// The catalogue of books for sale (maps the title of a book to its price)
private Hashtable catalogue;
// The GUI by means of which the user can add books in the catalogue
private BookSellerGui myGui;
// Put agent initializations here
protected void setup() {
// Create the catalogue
catalogue = new Hashtable();
// Create and show the GUI
myGui = new BookSellerGui(this);
myGui.show();
// Add the behaviour serving requests for offer from buyer agents
addBehaviour(new OfferRequestsServer());
// Add the behaviour serving purchase orders from buyer agents
addBehaviour(new PurchaseOrdersServer());
}
// Put agent clean-up operations here
protected void takeDown() {
// Close the GUI
myGui.dispose();
// Printout a dismissal message
System.out.println("Seller-agent "+getAID().getName()+" terminating.");
}
/**
This is invoked by the GUI when the user adds a new book for sale
*/
public void updateCatalogue(final String title, final int price) {
addBehaviour(new OneShotBehaviour() {
public void action() {
catalogue.put(title, new Integer(price));
}
} );
}
}
```

4.3 Класс «взаимодействие между агентами»

Одной из наиболее важных функциональных возможностей, которую предоставляют JADE агенты, является возможность коммуникации между ними. Обмен сообщениями в JADE является асинхронным и осуществляется с помощью Message Transport System. Формат сообщения соответствует стандарту FIPA. При этом внутри платформы для ускорения работы сообщения

пересылаются в виде Java-объектов, а при обмене между платформами (или с другими MAC) – в виде XML-строки. При отправке сообщения оно попадает в «почтовый ящик» (очередь сообщений) агента-адресата, о чем агент уведомляется. Сообщение может быть получено из «почтового ящика» агента и обработано любым из его режимов. Для выбора нужного типа сообщения из очереди программист может использовать фильтры. Пример обмена сообщениями показан на Рисунок 22.

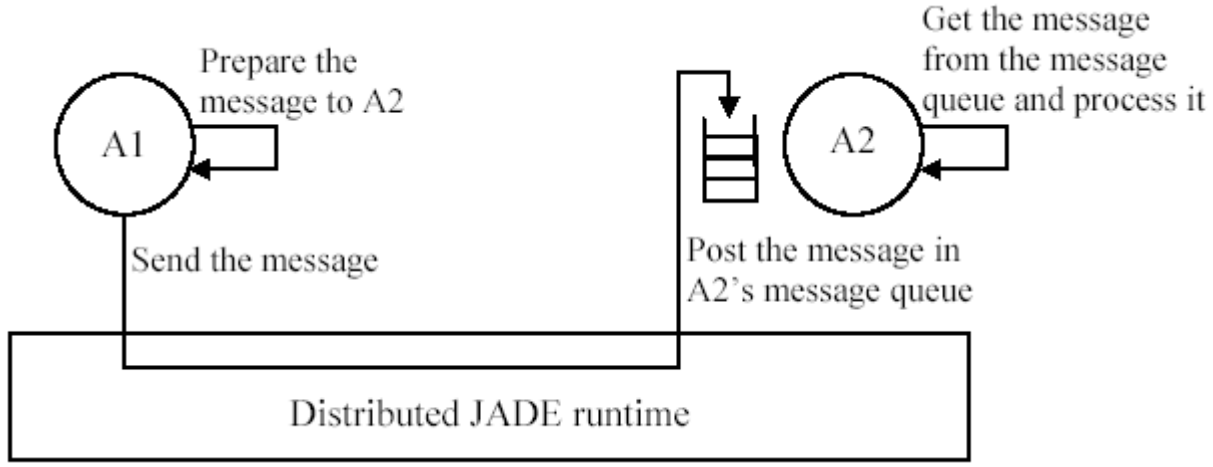

Рисунок 22 – Обмен сообщениями

Для ускорения разработки мультиагентных приложений в JADE существует библиотека шаблонов переговоров (*FIPA protocols*), таких как аукцион, contract net, подписка и т.д. Пример шаблона *Contract net* приведен на Рисунок 23.

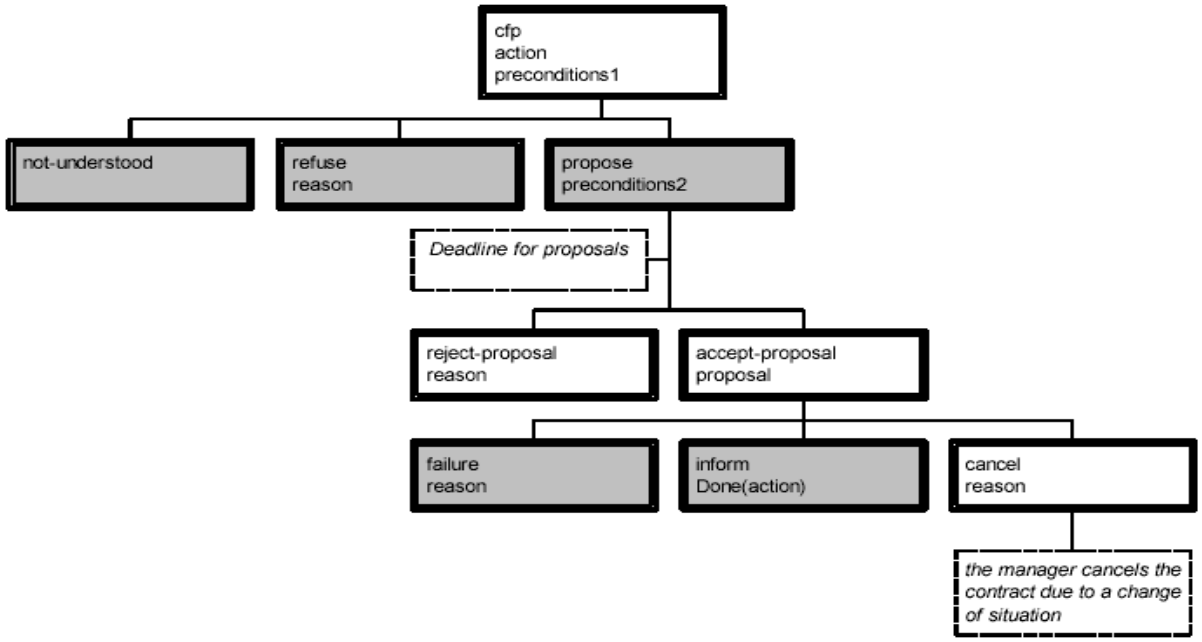

Рисунок 23 – Протокол Contract net

4.3.1 Язык ACL

Сообщения, которыми обмениваются между собой агенты JADE, имеют

формат языка ACL, определенного FIPA [4] как международный стандарт взаимодействия агентов. Этот формат включает следующие поля:

- отправитель сообщения;
- список получателей;
- коммуникативные действия («пожелания»), указывающие цель отправки сообщения. Типы сообщений:
  - REQUEST (запрос), если отправитель желает, чтобы получатель произвёл некоторое действие,
  - INFORM (информирование), если отправитель желает, чтобы получатель был извещён о некотором факте,
  - QUERY_IF, если отправитель желает знать, достигнуто или нет заданное условие,
  - CFP (извещение о предложении), PROPOSE (предложение), ACCEPT_PROPOSAL (принятие предложения), REJECT_PROPOSAL (отклонение предложения), если отправитель и получатель ведут переговоры,
  - другие типы коммуникативных действий;
- содержание, т.е. фактическая информация, содержащаяся в сообщении (например, действия, которые будут выполняться по запросу REQUEST; факт, о котором отправитель желает информировать в сообщении INFORM и т.п.);
- язык, на котором представлено содержание (отправитель и получатель должны иметь возможность кодировать/декодировать выражения согласно синтаксису языка);
- онтология, т.е. словарь символов, используемых в содержании, и их значения (отправитель и получатель должны понимать смысл символов одинаково);
- поля, используемые для управления несколькими параллельными переговорами, определяющие интервал времени для получения ответа, такие как *conversation-id*, *reply-with*, *in-reply-to*, *reply-by*.

Сообщение в JADE реализуется как объект класса *jade.lang.acl.ACLMessage*, предоставляющего методы *get()* и *set()* для установки значений всех полей при получении и посылке сообщений.

4.3.2 Посылка сообщений

Чтобы отправить сообщение другому агенту, необходимо заполнить поля объекта *ACLMessage* и вызвать метод *send()* класса *Agent*. Приведенный ниже код демонстрирует, как создать и отправить сообщение, информирующее агента по имени *Peter* о том, что сегодня идет дождь.

```
ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
msg.addReceiver(new AID("Peter", AID.ISLOCALNAME));
msg.setLanguage("English");
msg.setOntology("Weather-forecast-ontology");
msg.setContent("Today it's raining");
```

```
send(msg);
```

4.3.3 Посылка сообщений в приложении «Торговля книгами»

В приложении «Торговля книгами» используется сообщение CFP для извещения о том, что агент продавца *Buyer-agent* отправил агенту покупателя *Seller-agent* предложение на приобретение книг. Сообщение PROPOSE содержит предложение продавца, а сообщение ACCEPT_PROPOSAL содержит заказ покупателя. Сообщение REFUSE агента продавца сообщает о том, что книги, запрошенной агентом покупателя, нет в каталоге. В обоих типах сообщений, отправляемых агенту покупателя, содержится название книги. Содержанием сообщения PROPOSE должна быть цена на книгу. Ниже приведен пример, демонстрирующий, каким образом создается и оправляется сообщение CFP.

```
// Message carrying a request for offer
ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
for (int i = 0; i < sellerAgents.lenght; ++i) {
cfp.addReceiver(sellerAgents[i]);
}
cfp.setContent(targetBookTitle);
myAgent.send(cfp);
```

4.3.4 Получение сообщений

Среда исполнения JADE автоматически помещает сообщения, как только они доставлены, в очередь сообщений получателя. Агент может получать сообщения из своей очереди сообщений с помощью метода *receive()*. Этот метод возвращает первое сообщение, находящееся в очереди (и удаляет его оттуда), либо ничего не возвращает, если очередь сообщений пуста, после чего сразу же завершается.

```
ACLMessage msg = receive();
if (msg != null) {
// Process the message
}
```

4.3.5 Блокирование режима работы агента для ожидания сообщения

Очень часто в поведении агента необходимо реализовать получение сообщений от других агентов, например, в режимах *OfferRequestsServer* и *PurchaseOrdersServer*, которые были описаны в 4.2.4.2 Поведение агента Book-seller, требуется обслуживать сообщения от агента покупателя, пересылающего заявки на предложение и заказ. Такое поведение должно работать постоянно (циклическое поведение), причем при каждом выполнении метода *action()* требуется проверка, что сообщение было получено и обработано. Ниже

представлен режим *OfferRequestsServer* (режим *PurchaseOrdersServer* реализуется аналогично).

```
/**
Inner class OfferRequestsServer.
This is the behaviour used by Book-seller agents to serve incoming requests
for offer from buyer agents.
If the requested book is in the local catalogue the seller agent replies
with a PROPOSE message specifying the price. Otherwise a REFUSE message is
sent back.
*/
private class OfferRequestsServer extends CyclicBehaviour {
public void action() {
ACLMessage msg = myAgent.receive();
if (msg != null) {
// Message received. Process it
String title = msg.getContent();
ACLMessage reply = msg.createReply();
Integer price = (Integer) catalogue.get(title);
if (price != null) {
// The requested book is available for sale. Reply with the price
reply.setPerformative(ACLMessage.PROPOSE);
reply.setContent(String.valueOf(price.intValue()));
}
else {
// The requested book is NOT available for sale.
reply.setPerformative(ACLMessage.REFUSE);
reply.setContent("not-available");
}
myAgent.send(reply);
}
}
} // End of inner class OfferRequestsServer
```

В данном коде поведение *OfferRequestsServer* реализовано как внутренний класс класса *BookSellerAgent*. Это упрощает реализацию, т.к. обеспечивает прямой доступ к каталогу книг для продажи, но не является обязательным.

Метод *createReply ()* класса *ACLMessage* автоматически создает новую настройку *ACLMessageproperly* получателя и все поля, используемые для управления сеансом (*conversation-id*, *reply-with*, *in-reply-to*).

В соответствии с Рисунок 21, при добавлении поведения, описанного выше, поток агента входит в непрерывный цикл, что слишком нагружает процессор. Чтобы избежать этого, *метод action()* поведения *OfferRequestsServer* следует выполнять лишь тогда, когда появится новое сообщение. Для этого можно использовать метод *block()* класса поведений. Этот метод помечает поведение как «заблокированное», так что агент не планирует следующих выполнений. Когда в очередь сообщений агента добавится новое сообщение, все заблокированные режимы снова становятся доступными для исполнения и получают возможность обрабатывать полученные сообщения. Метод *action()* должен быть изменен следующим образом.

```
public void action() {
ACLMessage msg = myAgent.receive();
if (msg != null) {
```

```
// Message received. Process it
...
}
else {
block();
}
}
```

Приведенный выше код является типичным (и рекомендуемым) шаблоном для реализации приема сообщений внутри поведения агента.

4.3.6 Выбор сообщений с указанными характеристиками из очереди сообщений

Т.к. оба режима *OfferRequestsServer* и *PurchaseOrdersServer* являются циклическими, их метод *action()* которых начинается с вызова метода *myAgent.receive()*, что порождает следующую проблему: как определить, что поведение *OfferRequestsServer* выбирает из очереди сообщений только сообщения, в которых содержится запрос о наличии книги, а поведение *PurchaseOrdersServer* – только сообщения, содержащие заказы на поставку? Для разрешения этой проблемы нужно изменить код, указав соответствующие «шаблоны» при вызыве метода *receive()*. Когда шаблон указан, метод *receive()* возвращает первое сообщение (если оно есть), соответствующее данному шаблону, игнорируя все неподходящие сообщения. Подобные шаблоны реализованы как экземпляры класса *jade.lang.acl.MessageTemplate*, который предоставляет набор методов, позволяющих просто и гибко создавать шаблоны.

Как было указано в 4.3.3 *Посылка сообщений в приложении «Торговля книгами»*, для запроса о наличии книги используется сообщение CFP, и сообщение ACCEPT_PROPOSAL – для передачи предложения о заказе. Поэтому необходимо изменить метод *action()* класса *OfferRequestServer* так, чтобы вызов *myAgent.receive()* игнорировал все сообщения, кроме CFP.

```
public void action() {
MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.CFP);
ACLMessage msg = myAgent.receive(mt);
if (msg != null) {
// CFP Message received. Process it
...
}
else {
block();
}
}
```

4.3.7 Сложные переговоры

Режим *RequestPerformer*, описанный в 4.2.4.1 Поведение агента Book-buyer, представляет собой пример поведения, проводящего «сложные»

переговоры. В данном случае, «переговоры» – это последовательность сообщений, которыми обмениваются два или более агентов с чётко определёнными причинными и временными отношениями. Поведение *RequestProposal* должно послать CFP-сообщение нескольким агентам, известным агентам-продавцам, получить обратно все ответы и, в случае, если получен хотя бы один ответ типа PROPOSE, позже послать сообщение ACCEPT_PROPOSAL агенту-продавцу, который сделал предложение, и получить обратно ответ. Всякий раз, когда происходит обмен сообщениями, необходимо определить управляющие поля в сообщениях, участвующих в обмене. Это позволит легко и однозначно создавать шаблоны, соответствующие возможным ответам.

```
/**
Inner class RequestPerformer.
This is the behaviour used by Book-buyer agents to request seller
agents the target book.
*/
private class RequestPerformer extends Behaviour {
private AID bestSeller; // The agent who provides the best offer
private int bestPrice; // The best offered price
private int repliesCnt = 0; // The counter of replies from seller agents
private MessageTemplate mt; // The template to receive replies
private int step = 0;
public void action() {
switch (step) {
case 0:
// Send the cfp to all sellers
ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
for (int i = 0; i < sellerAgents.length; ++i) {
cfp.addReceiver(sellerAgents[i]);
}
cfp.setContent(targetBookTitle);
cfp.setConversationId("book-trade");
cfp.setReplyWith("cfp"+System.currentTimeMillis()); // Unique value
myAgent.send(cfp);
// Prepare the template to get proposals
mt = MessageTemplate.and(MessageTemplate.MatchConversationId("book-trade"),
MessageTemplate.MatchInReplyTo(cfp.getReplyWith()));
step = 1;
break;
case 1:
// Receive all proposals/refusals from seller agents
ACLMessage reply = myAgent.receive(mt);
if (reply != null) {
// Reply received
if (reply.getPerformative() == ACLMessage.PROPOSE) {
// This is an offer
int price = Integer.parseInt(reply.getContent());
if (bestSeller == null || price < bestPrice) {
// This is the best offer at present
bestPrice = price;
bestSeller = reply.getSender();
}
}
repliesCnt++;
if (repliesCnt >= sellerAgents.length) {
// We received all replies
step = 2;
}
```

```
}
else {
block();
}
break;
case 2:
// Send the purchase order to the seller that provided the best offer
ACLMessage order = new ACLMessage(ACLMessage.ACCEPT_PROPOSAL);
order.addReceiver(bestSeller);
order.setContent(targetBookTitle);
order.setConversationId("book-trade");
order.setReplyWith("order"+System.currentTimeMillis());
myAgent.send(order);
// Prepare the template to get the purchase order reply
mt = MessageTemplate.and(MessageTemplate.MatchConversationId("book-trade"),
MessageTemplate.MatchInReplyTo(order.getReplyWith()));
step = 3;
break;
case 3:
// Receive the purchase order reply
reply = myAgent.receive(mt);
if (reply != null) {
// Purchase order reply received
if (reply.getPerformative() == ACLMessage.INFORM) {
// Purchase successful. We can terminate
System.out.println(targetBookTitle+" successfully purchased.");
System.out.println("Price = "+bestPrice);
myAgent.doDelete();
}
step = 4;
}
else {
block();
}
break;
}
}
public boolean done() {
return ((step == 2 && bestSeller == null) || step == 4);
}
} // End of inner class RequestPerformer
```

JADE предоставляет основу для реализаций обмена сообщениями по определенным протоколам в пакете *jade.protopackage*. В частности, обмен сообщениями, который был реализован в примере, поддерживается протоколом *Contract net* и может быть реализован с помощью класса *jade.proto.ContractNetInitiator*.

4.3.8 Получение сообщений в блокирующем режиме

Кроме метода *receive()*, класс *Agent* предоставляет также метод *blockingReceive()*, который является блокирующим вызовом, т.к. он не возвращает управление до тех пор, пока в очереди сообщений агента не появится сообщение. Перегруженная версия этого метода получает в качестве аргумента *MessageTemplate* (метод не возвращает управление до тех пор, пока в очереди сообщений агента не появится сообщение, удовлетворяющее шаблону).

Подчеркнем, что метод *blockingReceive()* фактически блокирует поток агента. Поэтому, если из некоторого режима вызывается метод *blockingReceive()*, выполнение других режимов приостанавливается до тех пор, пока метод *blockingReceive()* не вернёт управление. Хорошим стилем программирования переговоров между агентами является использование *blockingReceive()* в методах *setup()* и *takeDown()*; использование внутри поведения метода *receive()* в сочетании с *Behaviour.block()* (как показано в 4.3.5 *Блокирование режима работы агента для ожидания сообщения*).

4.4 Сервис «желтых страниц»

В приведенном примере было сделано допущение, что существует некое фиксированное множество агентов-продавцов (*seller1* и *seller2*) и каждый покупатель заранее знает о них (поле *AID[] sellerAgents* класса *BookBuyerAgent* в коде, приведенном в *4.1.5 Передача аргументов агенту*). В данной главе описано, как использовать сервис «жёлтых страниц», предоставляемый платформой JADE, чтобы позволить агентам-покупателям динамически узнавать об агентах-продавцах, доступных в данный момент времени.

4.4.1 Агент DF

Агент *Directory Facilitator* (DF) – менеджер директорий (согласно спецификации FIPA) представляет службу желтых страниц (*yellow pages*), где агенты могут публиковать информацию о предоставляемых ими сервисах. С помощью DF агент может находить агентов, предоставляющих необходимые ему сервисы, и вступать с ними в переговоры. Внутри одной платформы может существовать несколько DF, например, предоставляющих информацию о различных группах сервисов или о сервисах различных групп агентов. Пример использования DF показан на Рисунок 24.

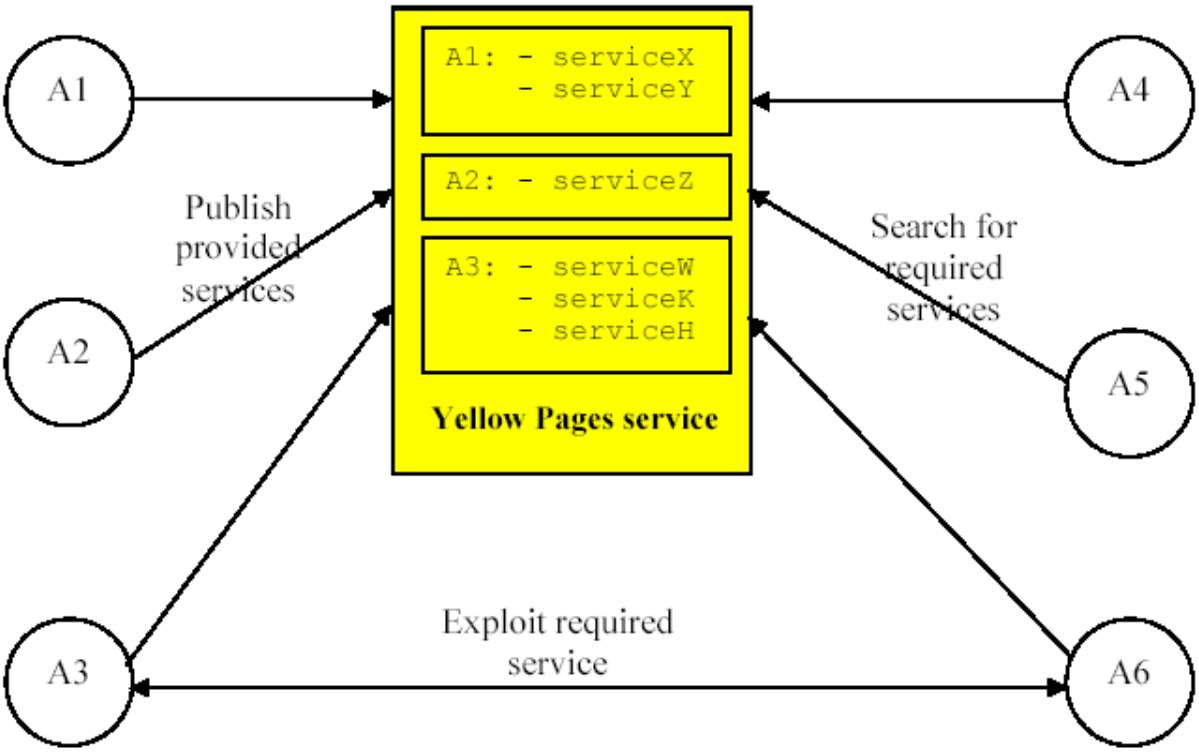

Рисунок 24 – Сервис «желтых страниц»

Каждая платформа JADE по умолчанию поддерживает агента DF (его локальное имя «df»). Могут быть запущены и объединены несколько DF-агентов (включая те, что находятся в платформе по умолчанию), для обеспечения единого распределённого каталога «жёлтых страниц».

4.4.2 Взаимодействие с DF

Поскольку DF является агентом, имеется возможность взаимодействовать с ним стандартным образом, обмениваясь ACL-сообщениями, используя, согласно спецификациям FIPA, соответствующий язык содержания (язык *SLO*) и соответствующую онтологию (онтологию *FIPA-agent-management*). Чтобы упростить эти взаимодействия, JADE предоставляет класс *jade.domain.DFService*, с помощью которого можно публиковать и выполнять сервисы через вызовы методов.

4.4.2.1 Публикация сервисов

Агент, желающий опубликовать (сделать доступными) один или более сервисов, должен предоставить DF, включающий AID этого агента, список возможных языков и онтологий, которые должны быть известны другим агентам для взаимодействия с ним, и список сервисов для публикации. Для каждого публикуемого сервиса предоставляется описание, включающее тип сервиса, его имя, языки и онтологии, необходимые для его использования, и ещё некоторые свойства, специфичные для данного сервиса. Классы *DFAgentDescription*, *ServiceDescription* и *Property*, входящие в состав пакета

jade.domain.FIPAAgentManagement, соответственно представляют собой три упомянутые абстракции.

Чтобы опубликовать сервис, агент должен создать описание, представленное экземпляром класса *DFAgentDescription*, и вызвать статический метод *register()* класса *DFService*. Обычно регистрация сервиса выполняется в методе *setup()*, как показано ниже в примере с агентом-продавцом книг.

```
protected void setup() {
...
// Register the book-selling service in the yellow pages
DFAgentDescription dfd = new DFAgentDescription();
dfd.setName(getAID());
ServiceDescription sd = new ServiceDescription();
sd.setType("book-selling");
sd.setName("JADE-book-trading");
dfd.addServices(sd);
try {
DFService.register(this, dfd);
}
catch (FIPAException fe) {
fe.printStackTrace();
}
...
}
```

Заметим, что в этом простом примере не были описаны ни языки, ни онтологии, ни свойства, специфичные для данного сервиса.

Когда агент завершает работу, следует отменить регистрацию опубликованных сервисов.

```
protected void takeDown() {
// Deregister from the yellow pages
try {
DFService.deregister(this);
}
catch (FIPAException fe) {
fe.printStackTrace();
}
// Close the GUI
myGui.dispose();
// Printout a dismissal message
System.out.println("Seller-agent "+getAID().getName()+" terminating.");
}
```

4.4.2.2 Поиск сервисов

Агент, которому требуется найти некоторые сервисы, должен предоставить агенту DF описание необходимого шаблона. Результатом поиска является список всех описаний, которые удовлетворяют указанному шаблону. Описание соответствует шаблону, если все поля, указанные в шаблоне, присутствуют с теми же значениями и в описании.

Использование статического метод *search()* класса *DFService* можно проиллюстрировать примером, в котором агент-покупатель книг динамически находит всех агентов, предоставлямых сервисом *«book-selling»* (продажа книг).

```
public class BookBuyerAgent extends Agent {
```

```
// The title of the book to buy
private String targetBookTitle;
// The list of known seller agents
private AID[] sellerAgents;
// Put agent initializations here
protected void setup() {
// Printout a welcome message
System.out.println("Hello! Buyer-agent "+getAID().getName()+" is ready.");
// Get the title of the book to buy as a start-up argument
Object[] args = getArguments();
if (args != null && args.length > 0) {
targetBookTitle = (String) args[0];
System.out.println("Trying to buy "+targetBookTitle);
// Add a TickerBehaviour that schedules a request to seller agents every minute
addBehaviour(new TickerBehaviour(this, 60000) {
protected void onTick() {
// Update the list of seller agents
DFAgentDescription template = new DFAgentDescription();
ServiceDescription sd = new ServiceDescription();
sd.setType("book-selling");
template.addServices(sd);
try {
DFAgentDescription[] result = DFService.search(myAgent, template);
sellerAgents = new AID[result.length];
for (int i = 0; i < result.length; ++i) {
sellerAgents[i] = result[i].getName();
}
}
catch (FIPAException fe) {
fe.printStackTrace();
}
// Perform the request
myAgent.addBehaviour(new RequestPerformer());
}
} );
...
```

Заметим, что обновление списка известных агентов-продавцов производится каждый раз перед очередной попыткой купить указанную книгу, т.к. агенты-продавцы в системе могут динамически появляться и исчезать. Класс *DFService* также предоставляет поддержку подписки на уведомления от DF о появлении агента, предоставляющего указанный сервис (методы *searchUntilFound()* и *createSubscriptionMessage()*).

4.5 Создание приложения «Торговля книгами»

Продемонстрируем работу приложения «Торговля книгами». Для этого создадим сцену с несколькими действующими агентами продавцов и покупателей, определим свойства каждого агента, запустим сцену на выполнение и будем в динамике наблюдать матчинг между агентами продавцов и покупателей.

4.5.1 Описание сценария «Торговля книгами»

Пусть в сценарии будут действовать 4 продавца книг и 4 покупателя. Продавцы могут предлагать по различным ценам как одну, так и несколько

книг трех наименований: «Hamlet», «Romeo and Juliet», «King Lear». Каждый покупатель стремится купить одну из 3-х книг, причем двое из них хотят купить книгу одного наименования. Каждому продавцу и каждому покупателю сопоставляется собственный агент. Индивидуальные значения свойств агентов определены в Таблица 1.

Таблица 1 – Значения свойств агентов

Агент	Свойства
BookSeller1	Название книги = Hamlet, Цена = 100 Название книги = King Lear, Цена = 320
BookSeller2	Название книги = Hamlet, Цена = 150
BookSeller3	Название книги = Romeo and Juliet, Цена = 200 Название книги = Hamlet, Цена = 120
BookSeller4	Название книги = Romeo and Juliet, Цена = 250 Название книги = King Lear, Цена = 300
BookBuyerHamlet	Название книги = Hamlet
BookBuyerRomeoAndJuliet	Название книги = Romeo and Juliet
BookBuyerKingLear	Название книги = King Lear
BookBuyerHamlet2	Название книги = Hamlet

Агенты продавцов и покупателей имеют определенную модель поведения и критерии принятия решения по выбору приемлемого варианта решения (Таблица 2).

Таблица 2 – Описание модели поведения агентов

Наименование агента	Цель	Необходимые параметры	Критерии выбора	Особенности
Агент продавца	Продать весь имеющийся в наличии товар (книги)	Список продаваемых книг с указанием наименования и цены за один экземпляр	Продажа книги первому обратившемуся покупателю по первоначально заявленной цене	Не выполняется просмотр списка покупателей, продажа книги по одной первому обратившемуся покупателю
Агент покупателя	Приобрести определенный вид товара (книгу с заданным наименованием)	Наименование книги, которую необходимо приобрести	Выбор среди всех продавцов того, который предлагает книгу с подходящим наименованием по минимальной цене	Принятие решения о покупке после первого просмотра списка всех продавцов, предлагающих книги. Удаление агента из системы после удачной покупки

4.5.2 Последовательность выполнения сценария «Торговля книгами»

1. Для запуска агентов можно компилировать их непосредственно из .java файлов с необходимыми параметрами, передаваемыми в командной строке. Однако удобнее воспользоваться механизмом регистрации, предоставляемым системным агентом RMA. Для этого файлы классов агентов необходимо сначала скомпилировать в .jar файлы, при этом название файла должно совпадать с названием главного класса в файле (т.к. при регистрации агента с помощью графического интерфейса не предусмотрено поле для ввода имени jar-файла, только для ввода имени класса).
2. Создадим агентов продавцов книг. Для этого выберем пункт главного меню *Actions -> Start New Agent*. Назовем агента *BookSeller1* (
3. Рисунок 25). Нажмем кнопку <*OK*>.

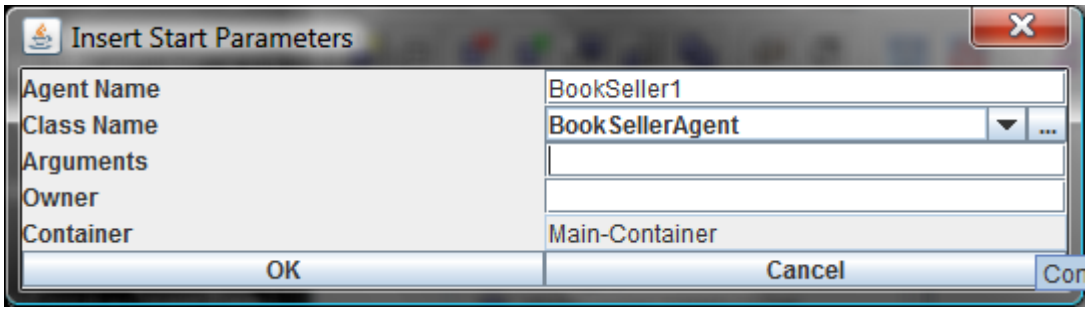

Рисунок 25 – Форма создания агента продавца

На экране появится форма для ввода наименования книги и ее цены (Рисунок 26). После каждого нажатия на кнопку <*OK*> будет создано новое наименование книги с заданными в форме параметрами для агента продавца, имя которого указано в заголовке формы.

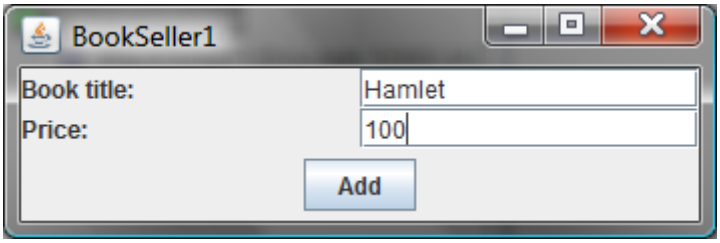

Рисунок 26 – Форма ввода параметров агента продавца

Примечание: при закрытии формы на Рисунок 26 агент прекращает работу! Поэтому формы ввода параметров для всех агентов продавцов должны все время быть открытыми!

Выполним аналогичные действия для создания всех 4-х агентов продавцов с набором книг для каждого из них (на Рисунок 27 показана форма создания

агента второго продавца).

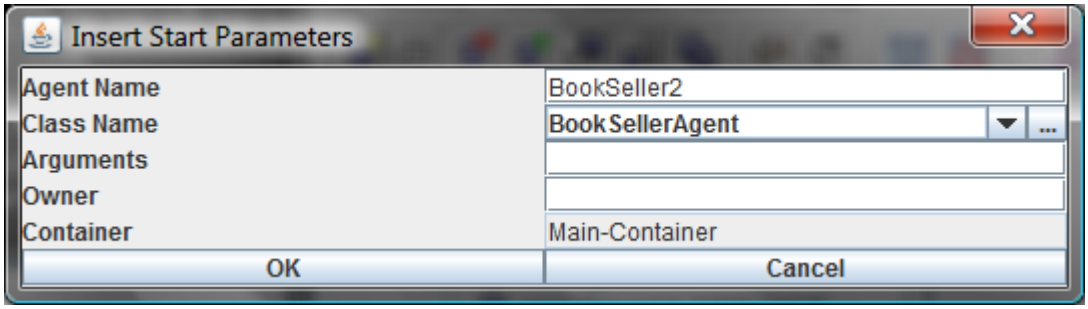

Рисунок 27 – Форма создания второго продавца

После создания и ввода параметров всех агентов продавцов контейнер по умолчанию в RMA выглядит, как показано на Рисунок 28.

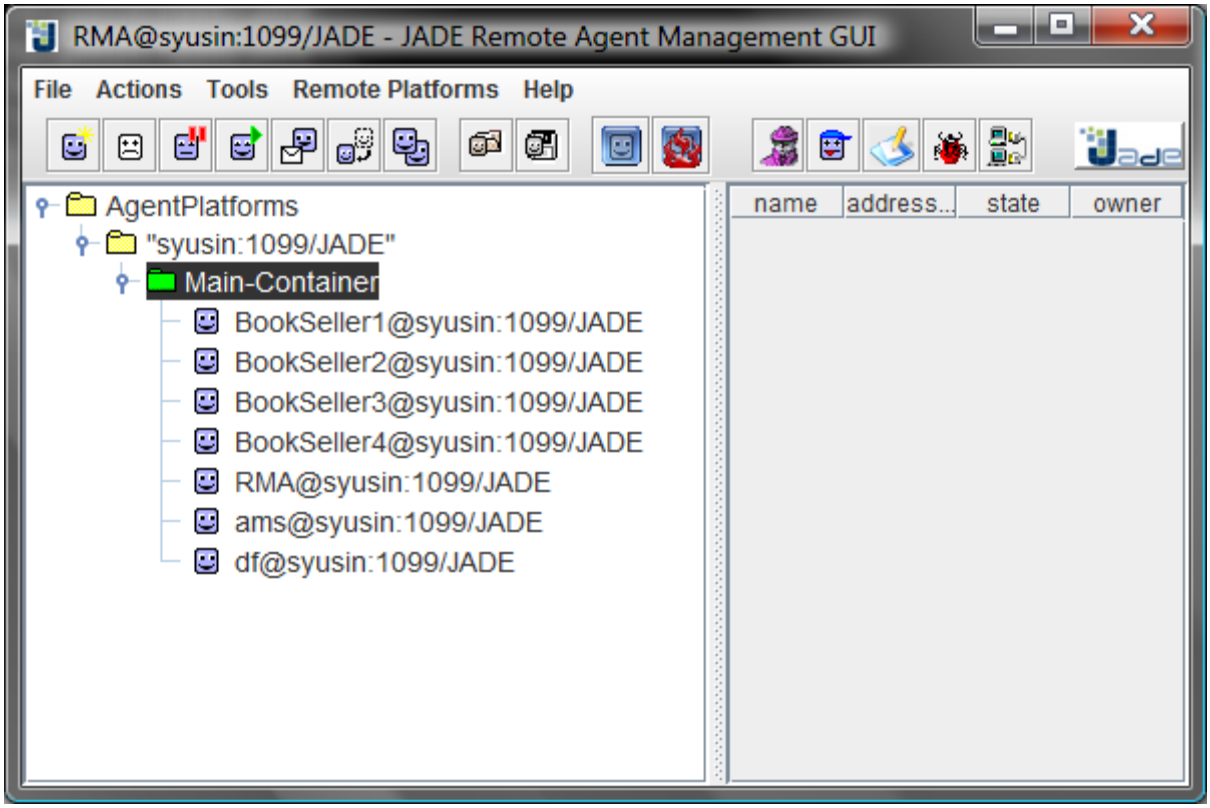

Рисунок 28 – Контейнер по умолчанию с зарегистрированными агентами продавцов в RMA

Окно командной строки показано на Рисунок 29.

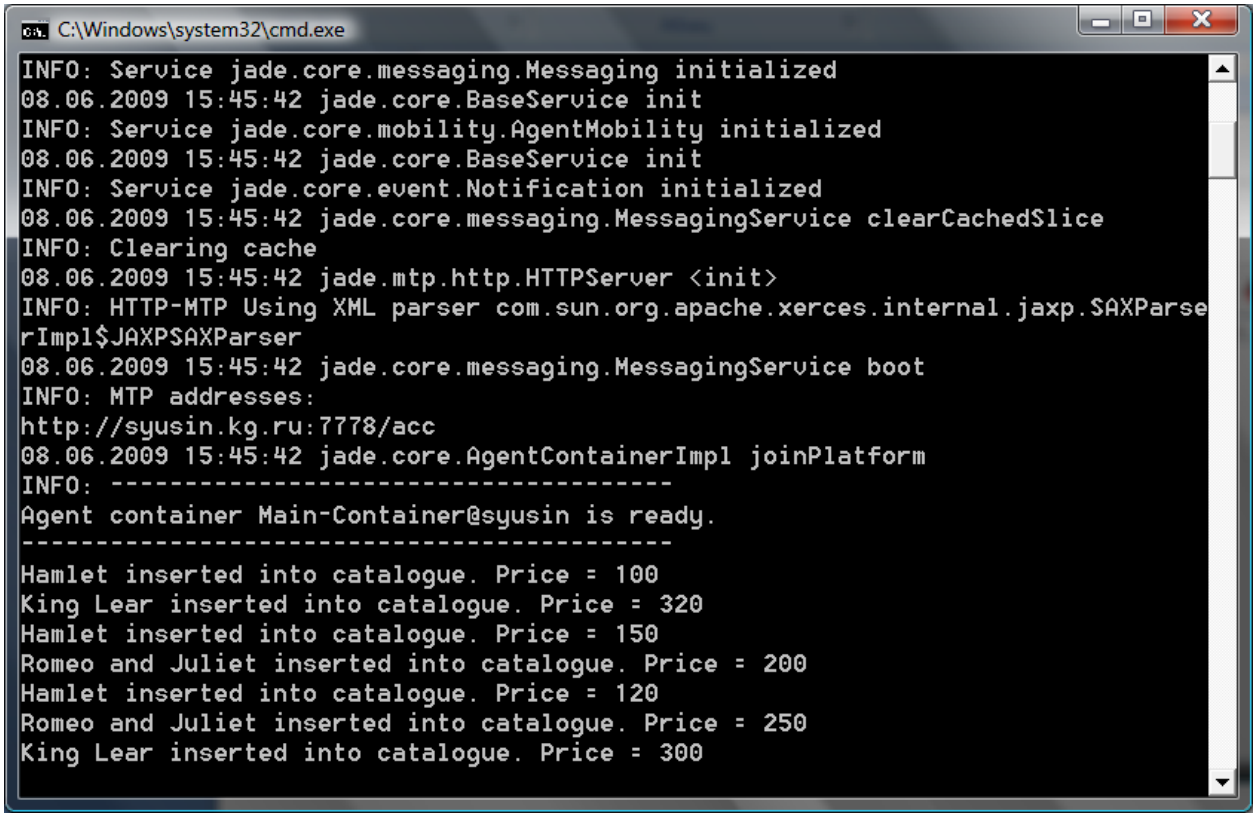

Рисунок 29 – Окно командной строки после ввода всех агентов продавцов

4. Создадим агента покупателя книг. Для этого выбирем пункт главного меню *Actions -> Start New Agent*. Назовем агента *BookBuyerHamlet* (он хочет купить книгу «Hamlet»). Наименование книги, которую хочет купить агент, указывается в поле *Arguments*. На Рисунок 30 показана форма создания агента покупателя книги «Hamlet».

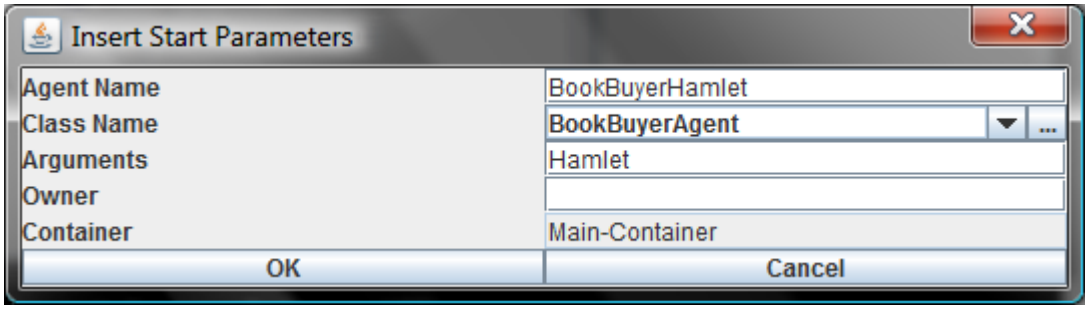

Рисунок 30 – Форма создания агента покупателя книги «Hamlet»

После создания агент регистрируется в системе и сразу же начинает выполнять матчинг для поиска соответствия наименования требуемой книги и наименований предлагаемых книг. Агент покупателя посылает запросы агентам продавцов периодически через определенный интервал времени.
Создадим еще двух агентов покупателей книг, как показано на Рисунок 31, Рисунок 32.

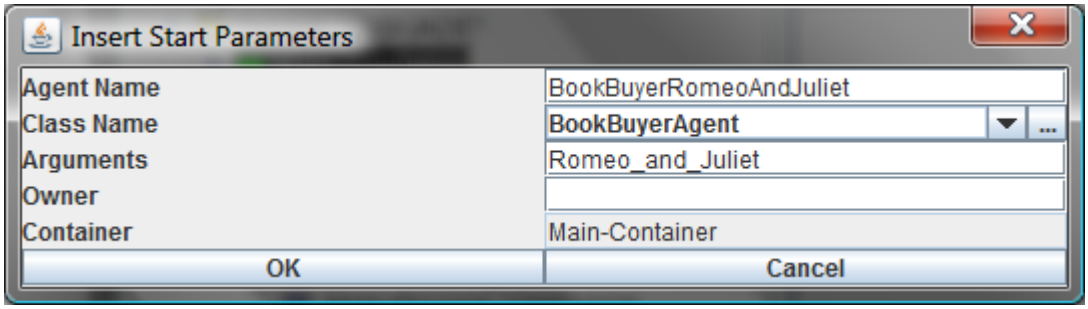

Рисунок 31 – Форма создания агента покупателя книги «Romeo and Juliet»

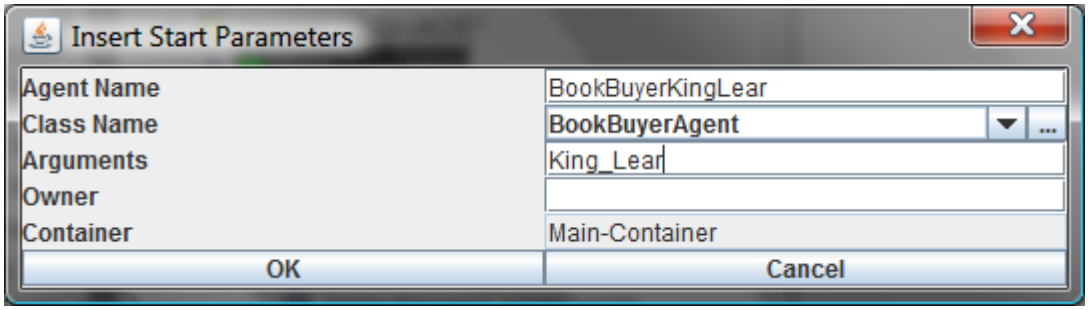

Рисунок 32 – Форма создания агента покупателя книги «King Lear»

Примечание: наименования книг в форме создания не должны содержать пробелов, т.к. система обрезает наименования до первого пробела и, соответственно, искомая книга найдена не будет (хотя агент продавца и позволяет вводить наименования книг с пробелами). Поэтому пробелы должны заменяться в наименовании, например, нижним подчеркиванием. Пример неудачного ввода наименования книги с пробелом представлен на Рисунок 33. Наличие пробела привело к сокращению наименования «Romeo and Juliet» до «Romeo» и матчинг был осуществлен неуспешно.

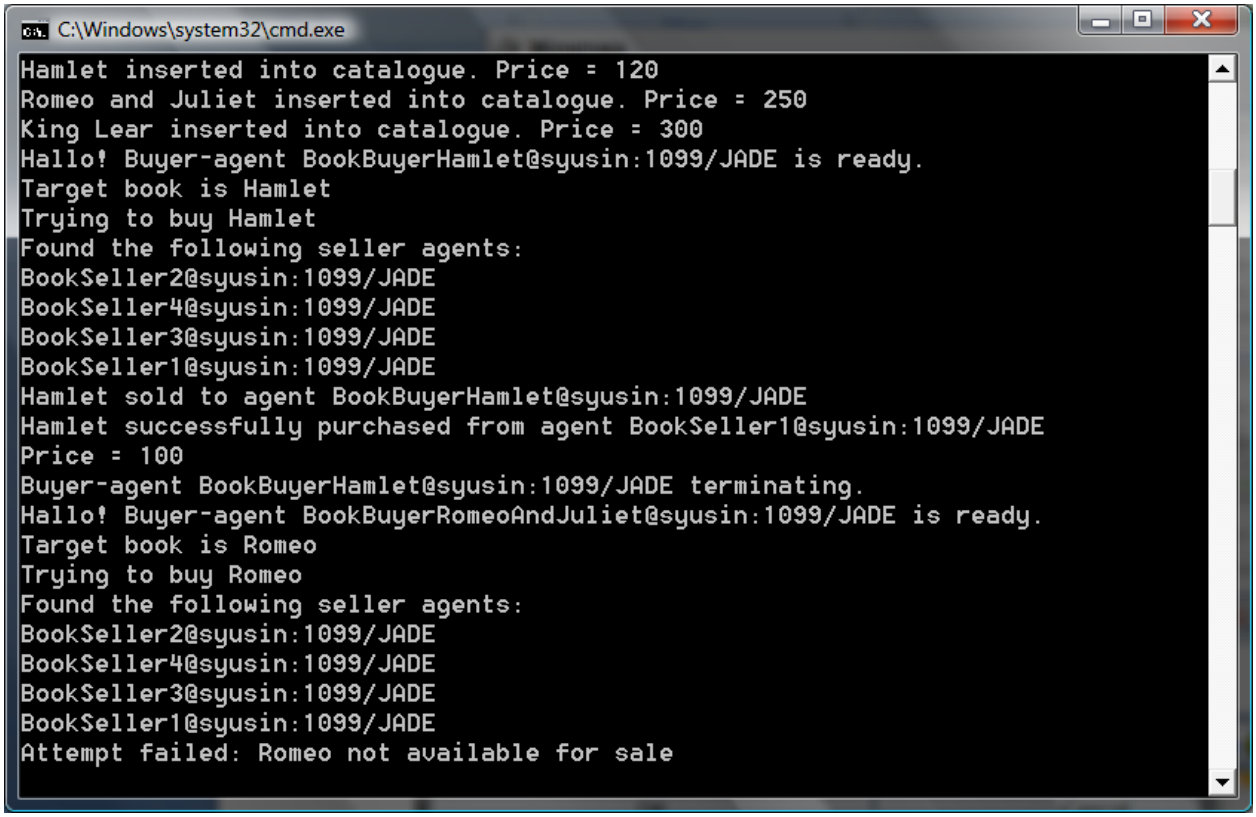

Рисунок 33 – Ошибочный ввод наименования книги

После регистрации всех агентов покупателей они сразу вступают в процесс матчинга. Каждый агент покупателя находит наиболее подходящего продавца (предлагающего наименьшую цену за искомую книгу) и создает с ним связь. Результат матчинга представлен на Рисунок 34.

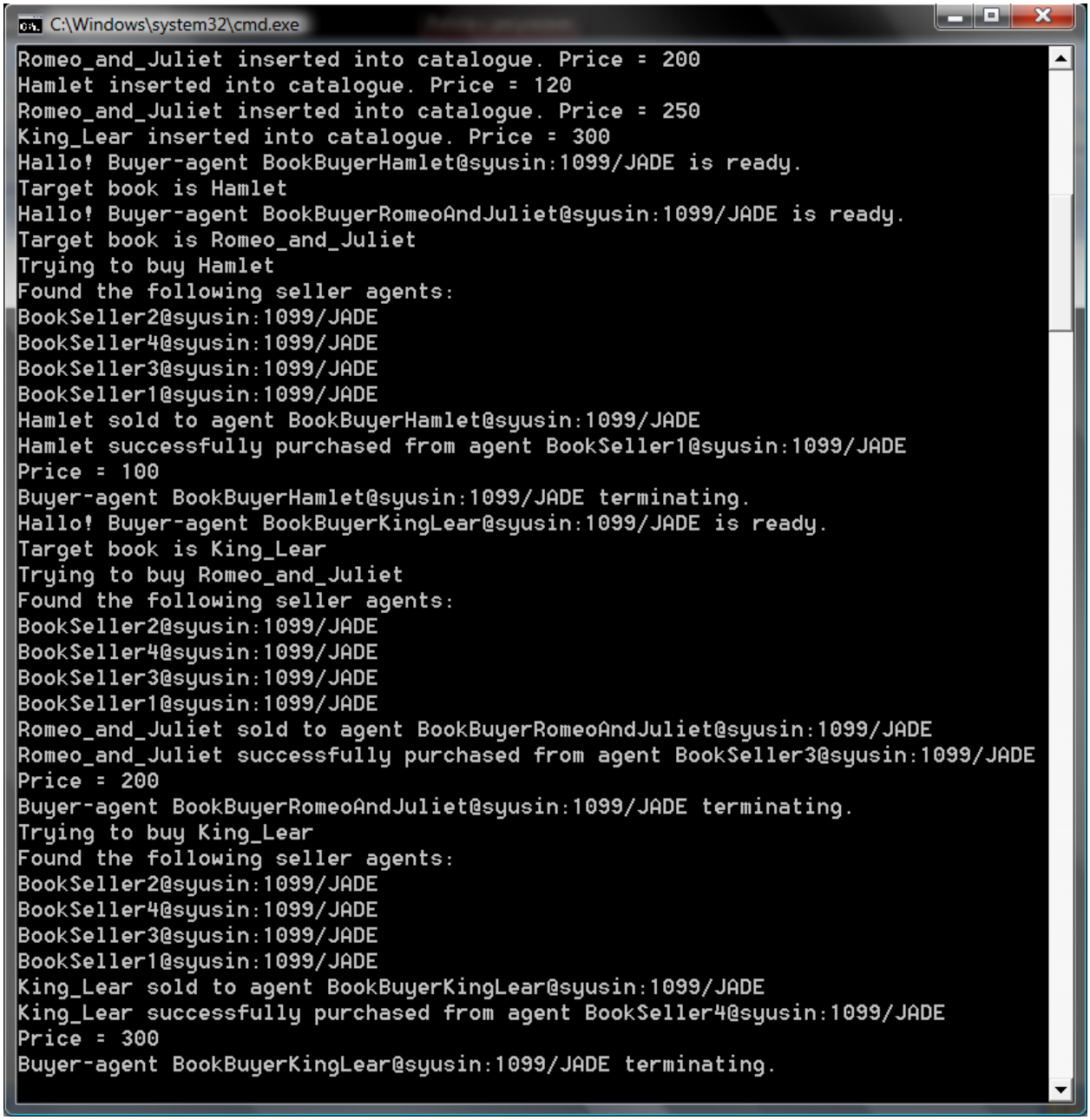

Рисунок 34 – Результат матчинга агентов

5. Зарегистрируем еще одного покупателя книг *BookBuyerHamlet2*, который также стремится купить книгу с наименованием «Hamlet» (Рисунок 35).

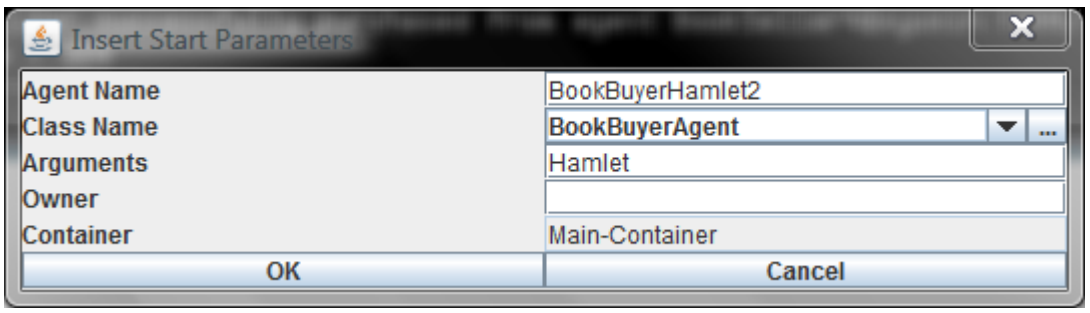

Рисунок 35 – Форма создания агента BookBuyerHamlet2

В результате матчинга новый агент успешно покупает книгу, установив связь с продавцом *BookSeller3*, выбрав минимальную цену из оставшихся вариантов. Таким образом, новые агенты будут и далее устанавливать связи в порядке возрастания цены на книги (поскольку в данном примере это единственный количественный критерий отбора) пока не будут задействованы все варианты. Результат матчинга для агента *BookBuyerHamlet2* приведены на Рисунок 36.

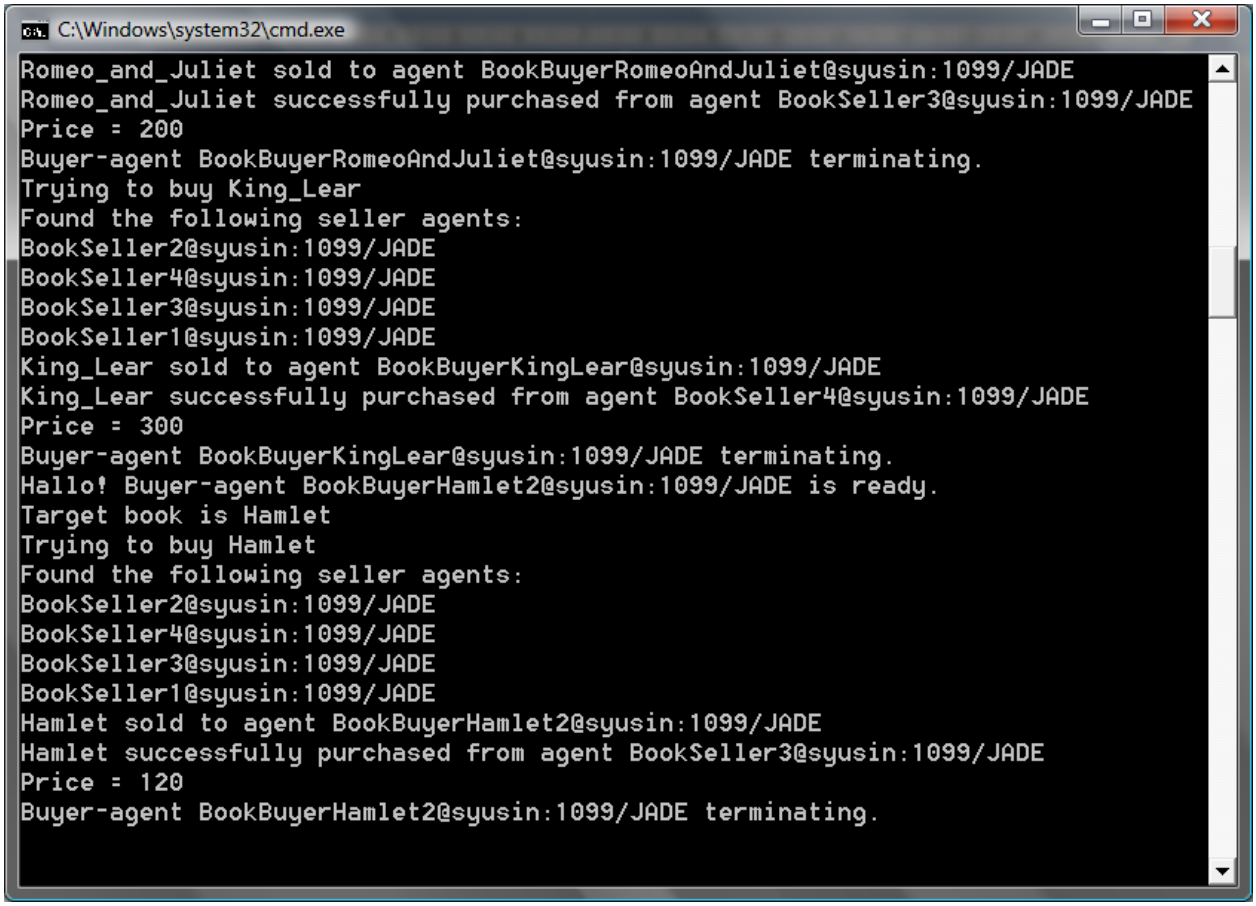

Рисунок 36 – Результат матчинга для агента BookBuyerHamlet2

5 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие функции выполняет агентная платформа JADE?
2. Что такое платформа и контейнер JADE?
3. Для чего предназначены агенты AMS и DF?
4. Какие инструменты предоставляет JADE для управления агентными приложениями? Каковы их функции?
5. Какова последовательность работы с агентом в среде JADE?
6. Какие функции выполняет класс «Агент» JADE?
7. Какие функции выполняет класс «Поведение агента» JADE?
8. Какие типы поведений агента Вам известны?
9. Из каких этапов состоит жизненный цикл агента JADE?
10. Как запланировать операцию агента JADE на заданный момент времени?

11. Какие функции выполняет класс «Взаимодействие между агентами» JADE?
12. Какие поля включает сообщение агенту согласно международному стандарту FIPA взаимодействия агентов?
13. Какие типы сообщений, пересылаемых между агентами, Вам известны?
14. Как работают методы посылки/получения сообщений?
15. Как реализовать ожидание сообщения агентом?
16. Как реализовать выбор сообщения с указанными характеристиками из очереди сообщений агента?
17. Как реализовать сложные переговоры между агентами?
18. Какие функции выполняет сервис «желтых страниц»?
19. Как опубликовать сервис?
20. Как найти необходимый для агента сервис?

6 ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

1. Разработайте мультиагентное приложение «Заказы-перевозчики» в предметной области «Логистика». В фирму, выполняющую перевозки крупногабаритных грузов, поступают заказы на транспортировку разнообразных грузов на различные расстояния. Фирма располагает некоторым множеством средств перевозки (перевозчиками). Заказы характеризуются весом груза и средствами, которыми заказ располагает для оплаты перевозки. Характеристиками перевозчика являются его максимальная грузоподъемность и продолжительность перевозки. Для выполнения каждого заказа на перевозку требуется подобрать одного перевозчика, имеющего максимальную грузоподъемность и/или обеспечивающего минимальную длительность перевозки. В приложении должны функционировать 2 агента заказов и 3 агента перевозчиков со следующими значениями атрибутов.

Agent	***Values***
Project Demand_1	CargoWeight = 1000; Account = 1000
Project Demand_2	CargoWeight = 2000; Account = 1500
Vessel Resource_1	MaxWeight = 1000; CruiseTime = 5
Vessl Resource_2	MaxWeight = 2000; CruiseTime = 6
Vessel Resource_3	MaxWeight = 2500; CruiseTime = 10

2. Разработайте мультиагентное приложение для предметной области «Размещение заказов на оборудование». Двум фирмам требуется закупить оборудование. С помощью информации двух заводов-изготовителей были определены показатели функционирования необходимого оборудования. Подберите для каждой фирмы завод-изготовитель.

Фирма	Требования фирмы		
	Стоимость, д.е.	Производительность, у.е.	Надежность, у.е.
1	5	100	8
2	6	150	5

Варианты оборудования	Показатели эффективности оборудования завода-изготовителя	
	производитель-ность, д.е.	надежность, у.е.
1	120	9
2	200	6

3. Разработайте мультиагентное приложение для предметной области «Продажа автомобилей». Требования покупателей и предложения продавцов приведены в таблицах. Какие автомобили будут куплены? Кто из покупателей не купит автомобиль?

Требования покупателей

	Покупатели				
	1	2	3	4	5
Пробег автомобиля	800	900	1000	700	1500
Цена, которую согласен заплатить покупатель, д.е.	5000	4500	4200	4900	5200

Предложения продавцов

Автомобили	Действительный пробег	Цена продавца, д.е.
1	1000	4900
2	650	5000
3	1000	4750
4	900	4600
5	750	4000

6. Разработайте мультиагентное приложение для предметной области «Кадры предприятия». 6 претендентов на 5 должностей проходят собеседование. Требования для назначения на определенную должность, а также характеристики каждого из претендентов приведены ниже в двух отдельных таблицах. Определите, на какую должность будет назначен каждый из

претендентов. Кто из претендентов не получит должность?

Требования для назначения на должность

	Должности				
	1	**2**	**3**	**4**	**5**
Рейтинг	850	900	1000	700	770
Ограничения по возрасту	35	40	40	45	50
Оклад	10000	12000	15000	7500	8000

Характеристики претендента

Претенденты	**Действительный рейтинг**	**Возраст**
1	800	33
2	720	42
3	1100	38
4	850	46
5	920	35
6	800	44

5. Разработайте мультиагентное приложение для предметной области «Кредитование предприятий». Четыре предприятия намерены получить кредит на развитие производства. Показатели эффективности работы предприятий приведены в следующей таблице. Банк предоставит кредит наиболее эффективно работающему предприятию. Пять банков определили максимальный размер кредита и срок его погашения.

Номер предприятия	**Показатели эффективности работы предприятия**			**Размер запрашиваемого кредита, д.е.**
	прибыль, д.е.	**Себестоимость единицы продукции, д.е.**	**производи-тельность, у.е.**	
1	30	40	300	100000
2	25	20	200	200000
3	40	45	250	150000
4	50	30	120	180000

Максимальный размер кредита

	Номер банка				
	1	**2**	**3**	**4**	**5**
Размер кредита, д.е.	120000	100000	150000	200000	220000
Срок погашения, мес.	8	10	7	6	5

ЗАКЛЮЧЕНИЕ

Динамично развивающаяся открытая агентная платформа JADE, обеспечивающая широкий спектр функциональных возможностей при работе с агентами, является эффективной средой разработки и управления агентными приложениями в различных предметных областях.

Методические указания «Использование платформы JADE для разработки мультиагентных приложений» посвящены рассмотрению функциональных воможностей мультиагентной платформы JADE для разработки приложений, реализующих взаимодействие агентов.

Методические указания содержат рекомендации по разработке классов агнетов.

Логическим завершением методических указаний являются контрольные вопросы и индивидуальные задания для самостоятельной работы студентов.

Вопросы, имеющие практическое значение для студентов при выполнении лабораторной работы, освещены с необходимой для использования полнотой.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Городецкий В.И., Грушинский М.С., Хабалов А.В. Многоагентные системы (обзор)// Новости искусственного интеллекта, №2, 1998, с. 64-116.

2. Андреев В.В., Минаков И.А., Пшеничников В.В., Симонова Е.В., Скобелев П.О. Основы построения мультиагентных систем. – Самара: изд-во ПГУТИ, 2007. – 290 с.

3. Абрамов Д.В., Андреев В.В., Симонова Е.В., Скобелев П.О. Открытые мультиагентные системы для принятия решений в задачах динамического распределения ресурсов. – Самара: изд-во ПГУТИ, 2008. – 290 с.

4. George Rzevski. FAQ On Agents and Multi-Agent Systems // http://www.naun.org/journals/educationinformation/eit-11.pdf

5. European Coordination Action for Agent-based Computing // http://www.agentlink.org

6. The Foundation of Physical Intelligent Agents // http://www.fipa.org/.

7. Java Agent DEvelopment Framework // http://jade.tilab.com/.

8. JADE. A White Paper // http://jade.cselt.it/papers/2003/WhitePaperJADEEXP.pdf

9. Кузьмин Д.А., Селютин Д.Г., Подкидышева Л.И. Анализ средств разработки мультиагентных систем. – http://library.krasu.ru/ft/ft/_articles/0112838.pdf.

10. Bellifemine F., Caire G., Greenwood D. Developing Multi-Agent System with JADE. – WILEY, 2007.

Методические материалы

ИСПОЛЬЗОВАНИЕ ПЛАТФОРМЫ JADE
ДЛЯ РАЗРАБОТКИ МУЛЬТИАГЕНТНЫХ ПРИЛОЖЕНИЙ

Методические указания

Составитель ***Симонова Елена Витальевна***

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ имени академика С. П. КОРОЛЕВА»
(Самарский университет)
443086, САМАРА, МОСКОВСКОЕ ШОССЕ, 34.

Изд-во Самарского университета.
443086 Самара, Московское шоссе, 34.