НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ МОСКОВСКИЙ ЭНЕРГЕТИЧЕСКИЙ ИНСТИТУТ

КАФЕДРА ПРИКЛАДНОЙ МАТЕМАТИКИ

Многоагентные системы Отчет №3

«Реализация многоагентной системы

“Биржа”

с помощью фреймворка JADE»

Группа: А -13м -16

Студент: ПономаревС.А. Преподаватель: Фоминых И.Б.

Москва 2016

1. **Краткое описание фреймворка**

***Java Agent Development Framework*** (JADE) — программная среда разработки мультиагентных систем и приложений, поддерживающая FIPA-стандарты для интеллектуальных агентов.

Включает в себя:

* Среду выполнения агентов. Агенты регистрируются и работают под управлением среды;
* библиотеку классов, которые используются для разработки агентных систем;
* набор графических утилит для администрирования и наблюдения за жизнедеятельностью активных агентов.

Программная среда Jade подключается к любому проекту на языке Java.

***Агент*** – это программа обладающая независимым поведением и способная обмениваться сообщениями с другими агентами. Агент обладает набором поведений (функций, методов), для отработки заданной реакции на определенные события.

Основной целью данной платформы является обеспечение существования и взаимодействия агентов.

Задачи агентной платформы заключаются в следующем:

* управление жизненными циклами агентов.
* эффективная передача сообщений между агентами.

***Вычислительная модель*** агента является многозадачной. В агенте отдельные задачи (или *поведения*) могут выполняться одновременно. Каждые функциональная возможность и/или сервис, предоставляемый агентом, должны быть реализованы как одно поведение или их набор. Внутренний планировщик, скрытый от разработчика, автоматически управляет планированием поведений.

Такой подход позволяет гибко конфигурировать политики выделения потоков агентам:

* один поток на всех агентов;
* жесткое выделение одного потока на агента;
* фиксированный пул рабочих потоков, больший или меньший, чем количество агентов в системе.

По умолчанию в системе каждому агенту платформы соответствует один поток.

Для отработки реакции на события агент имеет ***поведения***. Агент может отрабатывать несколько поведений одновременно. Обработка поведений происходит не по приоритетам (как java потоки), а совместно. Очередность исполнения отдельных простых поведений не гарантируется, для этого нужно использовать поведения специальных типов или синхронизацию.

Поведениям не выделяются отдельные потоки, так как при большом количестве поведений происходит постоянное переключение между ними, а переключение между потоками происходит относительно долго (вызов поведения как метода происходит в среднем в 100 раз быстрее, чем переключение потоков).

***Сервис обмена сообщениями*** одна из основополагающих частей архитектуры платформы JADE. Сервис основан на асинхронной передаче сообщений. Каждый агент имеет свой «почтовый ящик» - очередь входящих сообщений, куда помещаются все направленные агенту сообщения. В тот момент, когда сообщение помещается в очередь входящих сообщений, агент оповещается об этом.

В связи с тем, что агенты, обменивающиеся сообщениями, могут находиться как в одном, так и в разных контейнерах JADE использует два типа протокола: MTP (message transport protocol) и IMTP (internal message transport protocol).

В том случае если агенты находятся в разных контейнерах, используется RMI(Java Remote Method Invocation).

1. **Листинг программы**

Данная многоагентная система реализована в интегрированной среде разработки Eclipse

**Код класса BuyerAgent.java**

package agent1t;

//package

import jade.core.Agent;

import jade.core.AID;

//import jade.core.Agent;

import jade.core.behaviours.\*;

import java.util.\*;

import java.util.List;

import jade.core.Agent;

import jade.core.AID;

import jade.core.behaviours.\*;

import jade.lang.acl.ACLMessage;

import jade.lang.acl.MessageTemplate;

import jade.domain.DFService;

import jade.domain.FIPAException;

import jade.domain.FIPAAgentManagement.DFAgentDescription;

import jade.domain.FIPAAgentManagement.ServiceDescription;

import jade.core.behaviours.SimpleBehaviour;

import jade.lang.acl.ACLMessage;

public class BuyerAgent extends Agent {

/\*\*

@author Ponomarev Sergey

\*/

private AID[] sellerAgents;

//Setup is analog to constuctor

protected void setup() {

// Printout a welcome message

System.out.println("Hello! Buyer-agent "+getAID().getName()+" is ready.");

// Get the title of the stuff to buy as a start-up argument

Object[] args = getArguments();

if (args != null && args.length > 0) {

targetStuffTitle = (String) args[0];

System.out.println("Trying to buy "+targetStuffTitle);

// Add a TickerBehaviour that schedules a request to seller agents every minute

addBehaviour(new TickerBehaviour(this, 60000) {

protected void onTick() {

System.out.println("Trying to buy "+targetStuffTitle);

// Update the list of seller agents

DFAgentDescription template = new DFAgentDescription();

ServiceDescription sd = new ServiceDescription();

sd.setType("stuff-selling");

template.addServices(sd);

try {

DFAgentDescription[] result = DFService.search(myAgent, template);

System.out.println("Found the following seller agents:");

sellerAgents = new AID[result.length];

for (int i = 0; i < result.length; ++i) {

sellerAgents[i] = result[i].getName();

System.out.println(sellerAgents[i].getName());

}

}

catch (FIPAException fe) {

fe.printStackTrace();

}

// Perform the request

myAgent.addBehaviour(new RequestPerformer());

}

} );

}

else {

// Make the agent terminate

System.out.println("No target stuff title specified");

doDelete();

}

}

//The title of the stuff to buy

private String targetStuffTitle;

// The list of known seller agents

//private AID[] sellerAgents = {new AID("seller1", AID.ISLOCALNAME),

//new AID("seller2", AID.ISLOCALNAME)};

// Put agent clean-up operations here

protected void takeDown() {

// Printout a dismissal message

System.out.println("Buyer-agent "+getAID().getName()+" terminating.");

}

/\*\*

Inner class RequestPerformer.

This is the behaviour used by buyer agents to request seller

agents the target stuff.

\*/

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(targetStuffTitle);

cfp.setConversationId("stuff-trade");

//Уникальное значение нужно, что бы различать сообщения от продавцов, т.к. они начинают диалоги с одинаковыми названиями

//Устаревшая хрень, но все равно занятно

cfp.setReplyWith("cfp"+System.currentTimeMillis()); // Unique value

myAgent.send(cfp);

// Prepare the template to get proposals

mt = MessageTemplate.and(MessageTemplate.MatchConversationId("stuff-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(targetStuffTitle);

order.setConversationId("stuff-trade");

order.setReplyWith("order"+System.currentTimeMillis());

myAgent.send(order);

// Prepare the template to get the purchase order reply

mt = MessageTemplate.and(MessageTemplate.MatchConversationId("stuff-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(targetStuffTitle+" successfully purchased from agent "+reply.getSender().getName());

System.out.println("Price = "+bestPrice);

myAgent.doDelete();

}

else {

System.out.println("Attempt failed: requested stuff already sold.");

}

step = 4;

}

else {

block();

}

break;

}

}

public boolean done() {

if (step == 2 && bestSeller == null) {

System.out.println("Attempt failed: "+targetStuffTitle+" not available for sale");

}

return ((step == 2 && bestSeller == null) || step == 4);

}

} // End of inner class RequestPerformer

}

**Код класса SellerAgent.java**

package agent1t;

/\*\*

@author Ponomarev Sergey

\*/

import jade.core.Agent;

import jade.core.behaviours.\*;

import java.util.\*;

import jade.lang.acl.ACLMessage;

import jade.lang.acl.MessageTemplate;

import jade.domain.DFService;

import jade.domain.FIPAException;

import jade.domain.FIPAAgentManagement.DFAgentDescription;

import jade.domain.FIPAAgentManagement.ServiceDescription;

public class SellerAgent extends Agent {

// The catalogue of stuffs for sale (maps the title of a stuff to its price)

private Hashtable catalogue;

// The GUI by means of which the user can add stuffs in the catalogue

private SellerGui myGui;

// Put agent initializations here

// setup = \_\_construct !

protected void setup() {

// Create the catalogue

catalogue = new Hashtable();

// Create and show the GUI

myGui = new SellerGui(this);

myGui.show();

// Register the stuff-selling service in the yellow pages

DFAgentDescription dfd = new DFAgentDescription();

dfd.setName(getAID());

ServiceDescription sd = new ServiceDescription();

sd.setType("stuff-selling");

sd.setName("JADE-stuff-trading");

dfd.addServices(sd);

try {

DFService.register(this, dfd);

}

catch (FIPAException fe) {

fe.printStackTrace();

}

// 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() {

// 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.");

}

/\*\*

This is invoked by the GUI when the user adds a new stuff for sale

\*/

public void updateCatalogue(final String title, final int price) {

addBehaviour(new OneShotBehaviour() {

public void action() {

catalogue.put(title, new Integer(price));

}

} );

}

/\*\*

Inner class OfferRequestsServer.

This is the behaviour used by seller agents to serve incoming requests

for offer from buyer agents.

If the requested stuff 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() {

MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.CFP);

ACLMessage msg = myAgent.receive(mt);

if (msg != null) {

// CFP Message received. Process it

String title = msg.getContent();

ACLMessage reply = msg.createReply();

Integer price = (Integer) catalogue.get(title);

if (price != null) {

// The requested stuff is available for sale. Reply with the price

reply.setPerformative(ACLMessage.PROPOSE);

reply.setContent(String.valueOf(price.intValue()));

}

else {

// The requested stuff is NOT available for sale.

reply.setPerformative(ACLMessage.REFUSE);

reply.setContent("not-available");

}

myAgent.send(reply);

}

else {

block();

}

}

} // End of inner class OfferRequestsServer

/\*\*

Inner class PurchaseOrdersServer.

This is the behaviour used by seller agents to serve incoming

offer acceptances (i.e. purchase orders) from buyer agents.

The seller agent removes the purchased stuff from its catalogue

and replies with an INFORM message to notify the buyer that the

purchase has been sucesfully completed.

\*/

private class PurchaseOrdersServer extends CyclicBehaviour {

public void action() {

MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.ACCEPT\_PROPOSAL);

ACLMessage msg = myAgent.receive(mt);

if (msg != null) {

// ACCEPT\_PROPOSAL Message received. Process it

String title = msg.getContent();

ACLMessage reply = msg.createReply();

Integer price = (Integer) catalogue.remove(title);

if (price != null) {

reply.setPerformative(ACLMessage.INFORM);

System.out.println(title+" sold to agent "+msg.getSender().getName());

}

else {

// The requested stuff has been sold to another buyer in the meanwhile .

reply.setPerformative(ACLMessage.FAILURE);

reply.setContent("not-available");

}

myAgent.send(reply);

}

else {

block();

}

} } // End of inner class PurshaseOffersServer}

1. **Пример работы системы**

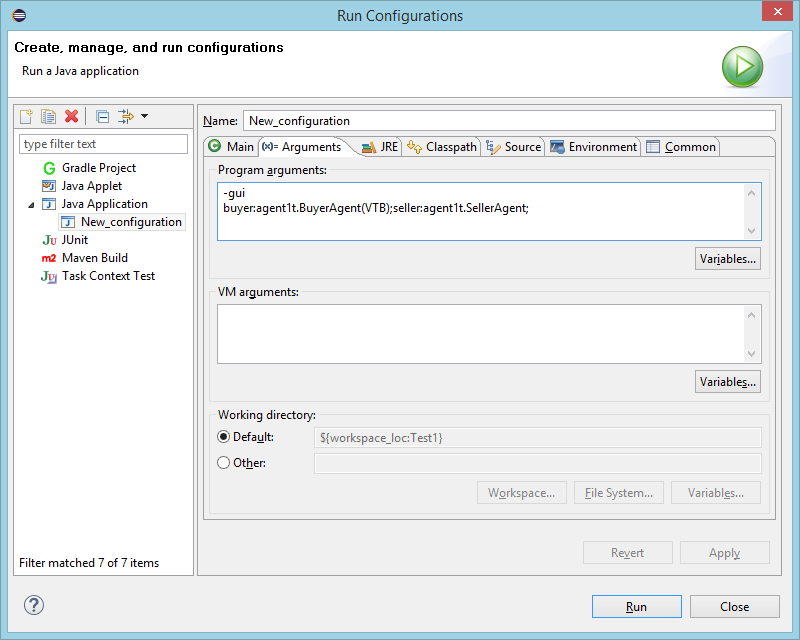


Рис1. Инициализация с двумя агентами

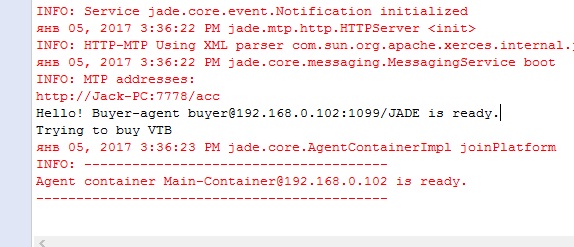


Рис.2. Агент пытается купить акции VTB

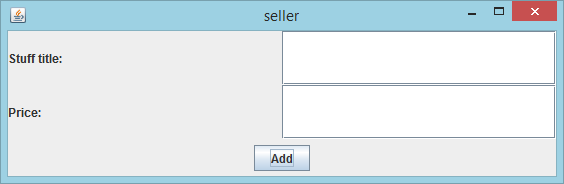


Рис3. Окно добавления товара

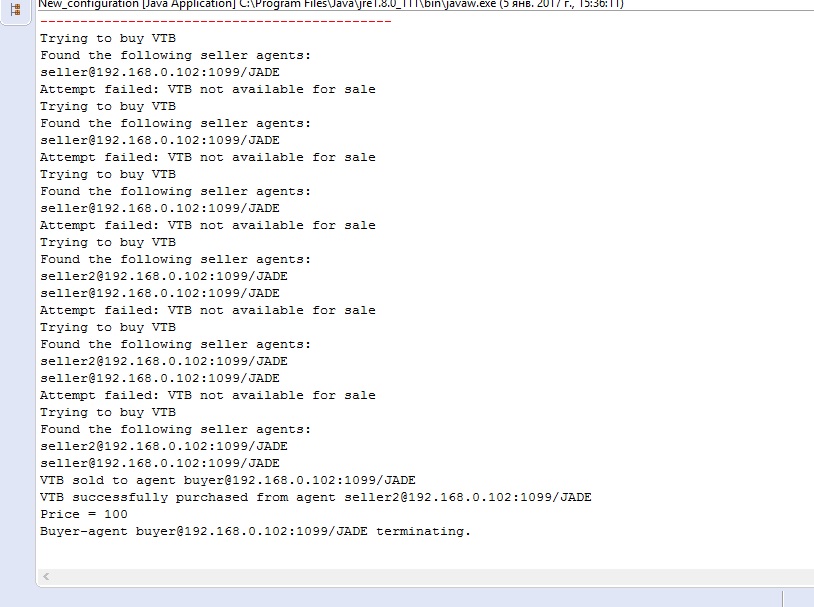


Рис5. Лог выполнения работы