**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”**

**іНСТИТУТ КОМП’ютерних НАУК та ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ**

### *Кафедра “Системи автоматизованого проектування”*



***Ознайомлення з процесом розроблення мультиагентної системи в середовищі JADE***

Методичні вказівки до лабораторної роботи № 2

з дисципліни “Методи проектування мультиагентних систем”

для студентів спеціальності 7.030.505 “Системне проектування”

та магістрів за фахом 8.030.505 “Системне проектування”.

Затверджено

на засіданні кафедри

“Системи автоматизованого проектування”

Протокол № 8 від 21.XI.2005 р.

на засіданні методичної ради ІКНІ

Протокол № 4-05/06 від 1.XII.2005 р.

ВАК № 1769 від 12.XII.2005 р.

Львів-2011

***Ознайомлення з процесом розроблення мультиагентної системи в середовищі JADE.*** Методичні вказівки до лабораторної роботи № 2 з дисципліни “Методи проектування мультиагентних систем” для студентів спеціальності 7.030.505 “Системне проектування” та магістрів за фахом 8.030.505 “Системне проектування” для стаціонарної та заочної форм навчання/Укл. А.Б.Романюк. - Львів: Національний університет ”Львівська політехніка”, 2011. - 23с.

## Укладачі: Романюк А. Б., канд. техн. наук, доцент

## 

Відповідальний за випуск: Лобур М. В., доктор техн. наук, професор

### Рецензенти: Каркульовський В. І., канд. техн. наук, доцент

ТкаченкоС.П., канд. техн. наук, доцент**.**

**МЕТА РОБОТА**

* Ознайомитися з процесом розробки мультиагентної системи в середовиці JADE.

**КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ**

|  |
| --- |
| JADE (Java Agent Development Framework) — програмне середовище розробки мультиагентних систем та прикладних програм. В JADE вбудовані інструменти, що полегшують адміністрування платформи і розробку прикладних програм.  В цьому керівництві показано, як створювати прості агенти і як зробити так щоб вони виконували задачі і комунікували один з одним. JADA повністю написане на Java і JADE програмісти повністю працюють на Java при розробці агентів. |
| 1. ***Загальний огляд JADE***   JADE це проміжне або підпрограмне забезпечення (скор. ППЗ; [англ.](http://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) [middleware](http://en.wikipedia.org/wiki/Middleware)) , яке призначене для спрощення та полегшення розробки мультиагентних систем. Воно складається з:   * + Середовища реального часу в якому агенти JADA можуть існувати. Це середовище потрібно запустити на вказаному хості перед запуском першого агента.   + Бібліотеки класів, які програмісти повинні/можуть використовувати для розробки своїх агентів   + Набору графічних інструментів, які дозволяють адмініструвати і відслідковувати активність запущених агентів |
| * 1. ***Контейнери та платформи***   Кожен запущений екземпляр середовища JADE називається контейнером (***Container***), тому що він може містити декілька агентів. Група (набір) активних контейнерів називається платформою (***Platform***). Один спеціальний основний контейнер (Main Container) завжди повинен бути активним у платформі і всі інші контейнери ним реєструються як тільки вони створюються. З цього слідує, що перший контейнер, який запускається на платформі повинен бути основним контейнером, а всі інші контейнери повинні бути звичайними “normal” тобто не основними контейнерами і повинні отримати вказівки де шукати (хост і порт) їх основного контейнера (тобто того контейнера, де вони повинні зареєструватися). Якщо інший основний контейнер запущений де небудь в мережі то це буде інша платформа і нові звичайні контейнери можуть бути в ній зареєстровані. На рис. 1. показано ілюстрацію наведених вище концепцій, як сценарій з двома JADE платформами, які складаються з трьох і одного контейнерів, відповідно. JADE агенти ідентифікуються за унікальними іменами і при умові, що вони знають імена один одного, вони можуть спілкуватися безпосередньо незалежно від їх фактичного місцезнаходження: всередині одного контейнера (наприклад агенти А2 і А3), в різних контейнерах (наприклад агенти А1 і А2) або в різних платформах (наприклад агенти А4 і А5).  Розробнику не обов’язково знати, як працює середовище JADA. Потрібно тільки запустити його перед запуском своїх агентів. Запуск JADE як основного або звичайного контейнера і виконання агентів в ньому описано в **JADE Administrative Tutorial** або ще детальніше в **Administrator’s Guide**, які доступні на сайті JADE. Нижче наведено декілька прикладів.  Наступна командна стрічка запускає основний контейнер і активує графічний інтерфейс керування JADE. Опція <classpath> повинна містити всі jade класи та всі необхідні спеціалізовані класи.  java -cp <classpath> jade.Boot -gui  Наступна командна стрічка запускає звичайний контейнер (опція –container), який реєструється основним контейнером, що запущений на хості avalon.tilab.com (опція –host) і активує агента, який називається john класс myPackage.MyClass (опція -agents)  java -cp <classpath> jade.Boot -container -host avalon.tilab.com -agents  john:myPackage.myClass    Рисунок 1. Контейнери та платформи. |
| * 1. ***AMS та DF***   Крім можливості приймання реєстрацій від інших контейнерів, основний контейнер відрізняється від звичайних контейнерів тим, що він містить два спеціальні агенти (автоматично запускаються коли запускається основний контейнер).  AMS (Agent Management System) – система керування агентами, яка підтримує (забезпечує роботу) службу іменування (тобто, гарантує, що кожен агент всередині платформи має унікальне ім’я) і наділений повноваженнями у платформі (наприклад, може створювати або знищувати агентів у віддалених контейнерах, через запити до AMS). В цьому керівництві не показано, як взаємодіяти з AMS, оскільки це стосується вже поглибленого JADE програмування.  DF (Directory facilitator) – маршрутизатор каталогів, який підтримує (забезпечує роботу ) службу «Жовті сторінки», за допомогою якої агент може знайти інших агентів, які надають послуги необхідні йому для досягнення своїх цілей. Використання служби «Жовті сторінки» агента DF показано в розділі 6. |
| 1. ***Аналіз предметної області. Приклад предметної області «Торгівля книжками».***   Цей розділ починається простим прикладом, що буде використовуватися і далі в керівництві для опису кроків, які необхідні для розробки агентно- орієнтованих прикладних програм разом з JADE. Сценарієм, який розглядається в цьому прикладі передбачаються агенти, які продають книжки та інші агенти, які купують книжки від імені користувачів.  Кожен агент – покупець, отримує назву книги, яку він повинен купити, як аргумент командної стрічки і періодично запитує всіх йому відомих агентів-продавців для отримання пропозиції на купівлю. Як тільки така пропозиція отримана агент покупець її підтверджує і надсилає замовлення. Якщо більш ніж один агент-продавець зробили пропозицію агенту-покупцю, агент-покупець вибирає одну кращу (найнижча ціна) з них. Здійснивши покупку, агент-покупець завершує свою роботу.  Кожен агент-продавець має мінімалістичний графічний інтерфейс, за допомогою, якого користувач може додавати нові назви (та їх ціну) в локальний каталог книжок на продаж. Агенти-продавці знаходяться в стані постійного очікування запитів від агентів-покупців. Коли вони отримують запит на книжку, вони перевіряють чи є дана книжка в їхньому каталозі. Якщо книжка є цьому каталозі то відповідають на запит пропозицією з ціною, інакше – відмовляють. Коли вони отримують замовлення на купівлю, вони обробляють його і видаляють книжку з свого каталогу.  Всі питання, пов’язані з електронною оплатою виходять за межі цього прикладу і не розглядаються у цьому керівництві.  Тексти програм цього прикладу доступні серед інших прикладів, які надає JADE - пакет examples.bookTrading |
| 1. ***Розроблення мультиагентної системи. Створення JADE агентів. Клас Agent.***   Створення агента в JADE це просто визначення класу, який успадковує клас jade.core.Agent та реалізація методу setup(), як показано нижче:  import jade.core.Agent;  public class BookBuyerAgent extends Agent {  protected void setup() {  // Виведення привітального повідомлення  System.out.println(“Hello! Buyer-agent “+getAID().getName()+” is ready.”);  }  }  Метод setup() призначений для ініціалізації агента. Фактичне завдання, яке повинен виконати агент зазвичай реалізовують в межах поведінок “behaviours” , як буде описано в розділі4. |
| * 1. ***Ідентифікатори агента***   Кожен агент ідентифікується «ідентифікатором агента», який представляється екземпляром класа jade.core.AID. Метод getAID() класу Agent дозволяє отримати ідентифікатор агента. Об’єкт AID містить глобальне унікальне ім’я та адреси. Ім’я в JADE має форму *<nickname>@<platform-name>* , якщо агент названий *Peter* існує на платформі *P1* то глобальне унікальне ім’я буде наступне *Peter@P1* . Адреси, які входять в AID – адреси платформи де існує агент. Ці адреси використовуються тільки тоді, коли агент має потребу в спілкуванні (комунікації) з іншим агентом, який існує на іншій платформі.  Якщо ім’я агента відоме то його AID можна отримати наступним чином:  String nickname = “Peter”;  AID id = new AID(nickname, AID.ISLOCALNAME);  Прапорець (константа) ISLOCALNAME показує, що перший параметр відповідає імені (локальному для платформи) і не є глобальним унікальним іменем агента. |
| * 1. ***Запуск агента***   Компіляцію агента можна здійснити наступним чином:  javac –classpath <JADE-classes> BookBuyerAgent.java  Для виконання скомпільованого агента середовище JADA повинно бути запущеним і потрібно вибрати ім’я агента для запуску:  java –classpath <JADE-classes>;. jade.Boot buyer:BookBuyerAgent  Більше деталей про компіляцію та запуск агентів можна знайти в **JADE Administrative Tutorial** або в **JADE Administrator’s Guide** , які доступні на сайті JADE  Результат виконання останньої команди наведений на рис 2.  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.  Рисунок 2. Результат виконання команди.  Перша частина виведеного на екран тексту – заголовок JADE, який виводиться кожного разу коли запускається середовище JADE. Далі йде ініціалізація сервісів ядра (сервіси ядра тут не описуються), які активовуються в запущеній платформи. На завершення, вказується що контейнер з ім’ям “Main-Container” готовий і це є завершенням запуску середовища JADE. Коли середовище JADE запущене, запускається агент і виводить привітання. Ім’я агента “buyer”, як було вказано в командній стрічці. Ім’я платформи “NBNT2004130496:1099/JADE” присвоюється автоматично на основі хоста і порта де запущено JADE. (для присвоювання імені платформи див. JADE Administrator’s Guide)   * 1. ***Завершення роботи агента***   Навіть якщо не передбачені жодні дії після виведення привітання, агент залишається запущеним. Для того щоб завершити його роботу потрібно викликати метод doDelete(). Подібно до методу setup(), який викликається середовищем JADE відразу після запуску агента і містить його ініціалізацію, метод takeDown() викликається безпосередньо перед завершенням роботи агента і містить операції очищення. |
| * 1. ***Передача аргументів агенту.***   Агенти можуть отримувати аргументи при запуску з командної стрічки. Ці аргументи можуть бути отримані як масив Object, за допомогою метода getArguments() класу Agent. Нам потрібно, щоб агент BookBuyerAgent отримував назву книжки, яку необхідно купити, через аргумент командної стрічки.  Для досягнення цього, агент модифікується наступним чином:  import jade.core.Agent;  import jade.core.AID;  public class BookBuyerAgent extends Agent {  // Назва книжки, яку необхідно купити  private String targetBookTitle;  // Список відомих агентів-продавців  private AID[] sellerAgents = {new AID(“seller1”, AID.ISLOCALNAME),  new AID(“seller2”, AID.ISLOCALNAME)};  // Ініціалізація агента  protected void setup() {  // Виведення привітання  System.out.println(“Hello! Buyer-agent “+getAID().getName()+” is ready.”);  // Отримання назви книжки, яку необхідно купити, через аргумент при старті  Object[] args = getArguments();  if (args != null && args.length > 0) {  targetBookTitle = (String) args[0];  System.out.println(“Trying to buy “+targetBookTitle);  }  else {  // Негайно завершити роботу агента  System.out.println(“No book title specified“);  doDelete();  }  }  // Операції очищення  protected void takeDown() {  // Виведення повідомлення про вивільнення  System.out.println(“Buyer-agent “+getAID().getName()+” terminating.”);  }  }  Потрібно зауважити, що список відомих продавців, для надсилання запитів фіксований. В подальшому буде показано, як їх отримувати динамічно.  Аргументи командної стрічки вказані у дужках і розділені пробілами.  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 |
| 1. ***Завдання агента – клас BEHAVIOUR***   Як вже зазначалося, фактична робота, яку повинен виконувати агент, зазвичай, реалізовується через поведінки “behaviours” агента.  Поведінка представляє задачу, яку агент може виконати і реалізовується як об’єкт класу, який успадковується від класу jade.core.behaviours.Behaviour. Для того щоб агент міг виконати задачу, описану об’єктом поведінки, достатньо додати поведінку агента за допомогою метода addBehaviour() класу Agent. Поведінка може бути додана у будь який момент: коли агент запускається (в методі setup()) або з інших поведінок. Кожен клас, який успадковує Behaviour повинен реалізовувати метод action(), який фактично визначає операції, які будуть виконуватися, коли агент виконує дану поведінку і метод done() (повертає булеве значення), який визначає чи завершена поведінка і чи потрібно її видалити з набору поведінок, які має агент. |
| * 1. ***Планування і виконання поведінок.***   Агент може виконувати одночасно (паралельно) декілька поведінко (моделей поведінок). Але важливо зазначити, що планування (розклад) поведінок агента є не приорітетним (так як потоки в Java) а кооперативним. Це означає, що коли поведінка виконується за розкладом то його метод action() викликається і виконується аж до свого завершення. Тому саме програміст визначає коли агент переключається від виконання одної поведінки до виконання іншої поведінки.  Хоча це вимагає додаткових не великих зусиль від програмістів, такий підхід має декілька переваг:   * + Дозволяє використовувати один Java потік для одного агента (що надзвичайно важливо, особливо в середовищах з обмеженими ресурсами, наприклад такими, як мобільні телефони)   + Забезпечує більшу продуктивність, оскільки перехід від одної поведінки до іншої відбувається набагато швидше ніж переключення між потоками Java   + Усуває всі питання синхронізації між паралельними поведінками, які повинні отримати доступ до одних і тих же самих ресурсів (це також підвищує продуктивність) оскільки всі поведінки виконуються одним і тим самим потоком Java (функціонують в одних і тих самих потоках)   + Коли відбувається переключення поведінки, статус агента не містить ніякої збереженої інформації і тому можна отримати його знімок “snapshot”. Це дозволяє реалізувати важливі функції, наприклад збереження статусу агента для подальшого відновлення (стабільність агента) або передачі його до іншого контейнера для віддаленого виконання (мобільність агента). Стабільність та мобільність це основні властивості, які надає JADE.   Алгоритм роботи потоку агента наведено на рис 3. В JADE є єдиний потік на одного агента. Оскільки JADE агенти написані на Java, програмісти звичайно можуть запускати нові потоки у будь який момент часу, але всі вищенаведені переваги , тобі будуть втрачені  Враховуючи описаний механізм планування, важливо підкреслити, що поведіка, приклад, якої наведено нижче, не допустить виконання будь-якої іншої поведінки, оскільки її метод action() ніколи не завершується.  public class OverbearingBehaviour extends Behaviour {  public void action() {  while (true) {  // do something  }  }  public boolean done() {  return true;  }  }  Коли немає поведінок, доступних для виконання потік агента “засинає” щоб не використовувати час процесора. Як тільки з’являється поведінка, доступна для виконання потік “прокидається”.    Рисунок 3. Алгоритм роботи агента |
| * 1. ***Одноразові поведінки, циклічні поведінки і універсальні поведінки.***   Можна виділити три типи поведінок.   1. Одноразові поведінки завершуються відразу і їх метод action() виконується тільки один раз.   jade.core.behaviours.OneShotBehaviour уже реалізовує метод done(), повертаючи значення true і може бути розширений для реалізації такої одноразової поведінки  public class MyOneShotBehaviour extends OneShotBehaviour {  public void action() {  // perform operation X  }  }  Операція Х виконується тільки один раз.   1. Циклічні поведінки ніколи не завершуються і їх метод action() виконує одні і ті ж операції, кожного разу коли він викликається.   jade.core.behaviours.CyclicBehaviour уже реалізовує метод done(), повертаючи значення false може бути розширений для реалізації циклічної поведінки  public class MyCyclicBehaviour extends CyclicBehaviour {  public void action() {  // perform operation Y  }  }  Операція Y виконується безкінечно (поки агент, який має таку поведінку не завершиться).   1. Універсальні поведінки, у які вбудовують статус і виконують різні операції в залежності від цього статусу. Виконання дій завершується, коли зустрічається дана умова.   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;  }  }  Операції X,Y та Z виконуються одна за одною і тоді завершується поведінка. JADE підтримує можливість комбінувати прості поведінки для створення складних поведінко. В документації можна знайти детальну інформацію про наступні поведінки SequentialBehaviour, ParallelBehaviour та FSMBehaviour |
| * 1. ***Планування операцій в задані моменти часу***   JADE надає два готові класи (в пакеті jade.core.behaviours), за допомогою яких, можливо просто реалізовувати поведінки, що виконують певні дії в задані моменти часу.   1. У WakerBehaviour методи action() та done() реалізовані таким чином, що виконується абстрактний метод handleElapsedTimeout() після заданого (вказаного у конструкторі) проміжку часу. Після виконання handleElapsedTimeout() поведінка завершується   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  }  } );  }  }  Операція X виконається через десять секунд після виведення “Adding waker behaviour”   1. У TickerBehaviour методи action() та done() реалізовані таким чином, що виконується абстрактний метод onTick() циклічно через задані у конструкторі проміжки часу. TickerBehaviour ніколи не завершується   public class MyAgent extends Agent {  protected void setup() {  addBehaviour(new TickerBehaviour(this, 10000) {  protected void onTick() {  // perform operation Y  }  } );  }  }  Операція Y виконується періодично, кожні 10 секунд. |
| * 1. ***Поведінки, які необхідні для прикладу «Торгівля книжками»***   Маючи опис базових типів поведінки, потрібно здійснити аналіз для встановлення поведінок, які необхідні агентам Book-buyer agent та Book-seller в прикладі з купівлею книжок |
| * + 1. ***Поведінки агента покупця книжок.***   Агент- покупець періодично опитує агентів продавців про книжку, яку йому необхідно купити. Можна легко досягнути подібної поведінки використовуючи TickerBehaviour, який на кожному такті буде додавати іншу поведінку, яка буде вже безпосередньо виконувати запит до агента продавця. Метод setup() класу BookBuyerAgent буде модифікований наступними чином:  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();  }  }  Потрібно звернути увагу на використання захищеної змінної myAgent - кожна поведінка має вказівник на агента, який її викликає.  Поведінка RequestPerformer, яка безпосередньо виконує запити до агента продавця, буде розглядатися пізніше при ознайомленні зі спілкуванням агентів. |
| * + 1. ***Поведінки агента продавця.***   Агент продавець книжок очікує запити від покупця і обслуговує його. Цими запитами можуть бути запити про наявність книжки та замовлення на купівлю. Подібну поведінку можна отримати, якщо агент продавець буде виконувати дві циклічні поведінки: одну призначену для обслуговування запитів про наявність а другу для обслуговування замовлень на купівлю. Як саме вхідні запити від покупця визначаються і обслуговуються, буде описано пізніше. Також агент продавець повинен виконувати одноразові поведінки оновлення каталогу книжок, що доступні для продажу, коли користувач додає нову книжку за допомогою графічного інтерфейсу.  Клас BookSellerAgent може бути реалізований наступним чином. Класи OfferRequestsServer , PurchaseOrdersServer будуть описані пізніше  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));  }  } );  }  }  Клас BookSellerGui - простий Swing GUI і його код доступний серед вихідний текстів пакету. |
| 1. ***Спілкування агентів. Клас ACLMESSAGE.***   Одна з найважливіших можливостей, які мають JADE агенти – можливість спілкуватися. Парадигма такого спілкування – асинхронна передача повідомлень (рис.4). Кожен агент має поштову скриньку (черга повідомлень агента) куди середовище JADA записує повідомлення надіслані іншими агентами. Кожен раз коли повідомлення розміщується в черзі повідомлень, агент – власник черги сповіщається про отримання нового повідомлення. Те, коли агент забере повідомлення з черги повідомлень для обробки і чи взагалі забере, повністю визначається програмістом.    Рисунок 4. Асинхронний обмін повідомленнями в JADE |
| * 1. ***Мова ACL***   Агенти JADE обмінюються повідомленнями у форматі мови ACL, визначеному міжнародним стандартом взаємодії агентів FIPA (<http://www.fipa.org>). Цей формат передбачає наявність у повідомленнях декількох обов’язкових полів (слотів):  - відправник повідомлення sender  - список отримувачів receivers  - мета спілкування (performative), вказує на те, що відправник хоче досягнути надіславши повідомлення. Повідомлення можуть надсилатися з метою: REQUEST (запит), якщо відправник хоче щоб отримувач виконав певну дію, INFORM(інформування), якщо відправник хоче, щоб отримувач дізнався про деякий факт QUERY\_IF(умовний запит), якщо відправник хоче знати чи зберігаються дані умові, CFP (call for proposal), PROPOSE(пропозиція), ACCEPT\_PROPOSAL (приймання пропозиції), REJECT\_PROPOSAL(відміна пропозиції), якщо відправник і отримувач приймають участь у переговорах та ін.  - вміст (контент) content, власне фактична інформація, яка міститься у повідомленні (тобто дії, які будуть виконуватися в REQUEST повідомленні, факт, який відправник хоче відкрити в INFORM повідомленні)  - мова контенту language, власне який синтаксис використовується для вираження вмісту (відправник і отримувач мають мати можливість кодувати/розкодовувати повідомлення, які сумісні з цим синтаксисом для ефективного спілкування)  - онтологія ontology, словник символів, які використовуються в контенті та їх значення (відправник і отримувач мусять надавати однакового змісту символам в повідомленні для ефективного спілкування)  - додаткові поля для контролю декількох одночасних розмов і вказання пауз для отримання відповідей: conversation-id, reply-with, in-reply-to, reply-by.  Повідомлення в JADE реалізовані як об’єкт класу jade.lang.acl.ACLMessage, який підтримує методи get та set для обробки всіх полів повідомлення. |
| * 1. ***Надсилання повідомлень***   Надсилання повідомлення іншому агенту, полягає у заповненні полів об’єкта ACLMessage та виклику метода send() класа Agent. Інформувати агента на ім’я Peter про те що сьогодні дощ (today it’s raining) можна наступним чином:  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);   * 1. ***Повідомлення у прикладі «Торгівлі книжками»***   У прикладі «Торгівля книжками» зручно використовувати CFP (call for proposal) повідомлення, для повідомлень, які надсилає агент покупець агенту продавцю з запитом про пропозицію на книжку. Повідомлення PROPOSE можуть бути використані для пропозицій агентів продавців, а ACCEPT\_PROPOSAL для згоди покупця і оформлення замовлення. Повідомлення REFUSE будуть використовуватися для надсилання повідомлень агентами продавцями, коли книжка не була знайдена в каталозі. В обох типах повідомлень, які надсилають агенти покупці, вмістом повідомлення буде назва книжки. Вміст повідомлення PROPOSE – ціна книжки. У наступному прикладі повідомлення CFP створюється і надсилається:  // 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); |
| * 1. ***Отримання повідомлень.***   Середовище JADA автоматично поміщає повідомлення в особисту чергу повідомлень отримувача відразу, як воно надходять. Агент може забрати повідомлення з черги повідомлень використовуючи метод receive(). Цей метод повертає перше повідомлення з черги повідомлень (знищує його) або null, якщо черга повідомлень пуста і відразу зупиняється.  ACLMessage msg = receive();  if (msg != null) {  // Process the message  } |
| * 1. ***Блокування поведінки в очікування повідомлення.***   Дуже часто програмістам потрібно реалізовувати поведінки які обробляють повідомлення отримані іншими агентами.  Такими випадками є поведінки OfferRequestsServer та PurchaseOrdersServer коли потрібно обробити повідомлення від агентів-покупців з запитами на покупку та замовленнями на поставку. Така поведінка повинна виконуватись безперервно (циклічні поведінки) і при кожному виконанні їх метод action() мусить перевіряти чи надійшло повідомлення і обробляти його. Ці дві поведінки дуже подібні.  Нижче наведено поведінку OfferRequestsServer.Текст PurchaseOrdersServer доступний серед вихідний текстів пакету.  /\*\*  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  Поведінка OfferRequestsServer реалізована як внутрішній клас класу BookSellerAgent. Таке спрощення дозволяє безпосередньо доступатися до каталогу книжок для продажу,але так робити не обов’язково.  Метод createReply() класу ACLMessage автоматично створює нове ACLMessage повідомлення з необхідними налаштуваннями отримувачів, та всіх контрольних полів розмови (conversation-id, reply-with, in-reply-to)  Якщо поглянути на рис3. то можна зауважити, що коли додається вищенаведена поведінка, потік агента починає безкінечний цикл, який надмірно навантажує процесор. Для уникнення цього, потрібно, щоб метод action() поведінки OfferRequestsServer виконувався тільки тоді, коли отримано нове повідомлення. Для цього можна використати метод block() класу Behaviour. Цей метод позначає поведінку як заблоковану “blocked”, таким чином щоб агент не планував її виконання. Коли нове повідомлення з'являється в черзі повідомлень агента, всі заблоковані поведінки стають знову доступними для виконання і мають можливість обробити отримане повідомлення. Метод action() для цього повинен бути модифікований наступним чином:  public void action() {  ACLMessage msg = myAgent.receive();  if (msg != null) {  // Message received. Process it  ...  }  else {  block();  }  }  Вищенаведений код є типовим (строго рекомендованим) шаблоном для отримання повідомлення в поведінці (всередині). |
| * 1. ***Вибір повідомлень із заданими характеристиками з черги повідомлень.***   Враховуючи що дві поведінки OfferRequestsServer та PurchaseOrdersServer є циклічними, метод action() яких починається з виклику методу myAgent.receive(), можна зауважити проблему: як переконатися, що поведінка OfferRequestsServer вибирає з черги повідомлень агента тільки повідомлення з запитами про наявність книжки, а поведінка PurchaseOrdersServer тільки повідомлення, що містять замовлення на поставку? Для розв’язання цієї проблеми потрібно модифікувати текст програми, вказавши відповідні «шаблони» при виклику метода receive(). Коли вказаний такий шаблон метод receive() повертає перше повідомлення (якщо воно є), яке відповідає цьому шаблону та ігнорує всі інші повідомлення. Такі шаблони реалізуються як екземпляри класу jade.lang.acl.MessageTemplate, який надає множину методів для простого і гнучкого створення шаблонів.  Повідомлення CFP використовуються для повідомлень з запитом про наявність книжки і повідомлення ACCEPT\_PROPOSAL для повідомлень, які передають пропозиції про замовлення (згоду на пропозицію), тому потрібно внести зміни у метод action() класу OfferRequestsServer таким чином, щоб виклик myAgent.receive() ігнорував всі повідомлення крім тих, які містять перформацію (мету повідомлення)  public void action() {  MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.CFP);  ACLMessage msg = myAgent.receive(mt);  if (msg != null) {  // CFP Message received. Process it  ...  }  else {  block();  }  } |
| * 1. ***Складні розмови (комунікації).***   Поведінка RequestPerformer , яка вже згадувалася, представляє приклад поведінки, яка підтримує складну «розмову». Розмова в даному випадку, це послідовність повідомлень, якими обмінюються два або більше агентів з чітко визначеними причинними і часовими залежностями. Поведінка RequestPerformer надсилає повідомлення CFP декільком агентам (всім відомим агентам-продавцям), отримує від всіх відповіді і у випадку, якщо отримано хоча б одну відповідь PROPOSE, пізніше надсилає повідомлення ACCEPT\_PROPOSAL агенту продавцю, який зробив пропозицію і отримує відповідь на це повідомлення. Кожен раз коли проходить обмін повідомленнями потрібно описувати контрольні поля повідомлень, які приймають участь в обміні. Це дозволяє просто і без неоднозначностей створювати шаблони, які відповідають можливим відповідям.  /\*\*  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  Складні комунікації зазвичай здійснюються на основі чітко визначеного протоколу взаємодії. JADE надає підтримку для реалізації обміну повідомленнями згідно таких протоколів в пакеті jade.proto . Розмова, яка реалізована у прикладі відповідає протоколу “Contract-net” і може бути просто реалізована з використанням класа jade.proto.ContractNetInitiator. Детальну інформацію про протоколи взаємодії можна отримати з документації JADE/ |
| * 1. ***Отримання повідомлень в заблокованому стані***   Крім метода receive() клас Agent також підтримує метод blockingReceive(), який як зрозуміло з його назви, є викликом блокування: нічого не повертає поки немає повідомлення в черзі повідомлень агента. Також доступна версія на основі перезавантаження, яка приймає MessageTemplate як параметр і нічого не повертає поки немає повідомлення, яке відповідає вказаному шаблону.  Важливо підкреслити, що метод blockingReceive() фактично блокує потік агента. Тому якщо метод blockingReceive() викликається з середини поведінки, це перешкоджає виконанню всіх інших поведінок аж до завершення виклику blockingReceive(). Внаслідок цього при програмуванні, для отримання повідомлень: потрібно використовувати blockingReceive() в методах setup() та takeDown(); використовувати receive() разом з Behaviour.block() в межах поведінок. |
| 1. ***Служба «Жовті сторінки» . Клас DFSERVICE.***   В наведеному тексті програми було зроблено припущення, що існує фіксована множина агентів-продавців (seller1 та seller2)) і кожен покупець вже про них знає (дивитися AID[] sellerAgents змінну класу BookBuyerAgent). Тепер розглянемо яким чином відкинути це припущення і використовувати службу жовтих сторінок, яку надає JADE платформа, для того щоб агенти-покупці динамічно дізнавалися про агентів-продавців, які доступні на даний момент часу. |
| * 1. ***Агент DF***   Служба “yellow pages” дозволяє агентам публікувати агентам дані про один або більше сервісів, які вони надають, таким чином щоб інші агенти могли знаходити і успішно використовувати ці сервіси, як показано на рис.4.  Службу “yellow pages” в JADE (згідно специфікації FIPA) підтримує агент DF (Directory Facilitator, Менеджер Директорій). Кожна сумісна з FIPA платформа підтримує по замовчуванню DF агент (локальне ім’я “df”) . Інші DF агенти можуть бути активовані і декілька таких агентів разом з тими, що вже знаходяться на платформі по замовчуванню можна об’єднати для підтримки одного розподіленого каталогу жовтих сторінок.   * 1. ***Взаємодія з DF.***   Оскільки DF – агент, то це дозволяє взаємодіяти з ним використовуючи звичайні способи обміну ACL повідомленнями, використовуючи мову вмісту (контенту) SL0 та відповідну онтологію (онтологію FIPA-agent-management), згідно специфікації FIPA. Для спрощення цієї взаємодії JADE надає клас jade.domain.DFService за допомогою якого можна публікувати і шукати сервіси через виклики методів.    Рисунок 5.  Служба «Жовті сторінки» |
|  |
| * + 1. ***Сервіси для публікацій***   Агент, який бажає опублікувати (зробити доступними) один або декілька сервісів повинен надати DF опис з вказанням AID агента, список можливих мов і онтологій, які повинні знати інші агенти для взаємодії з ним та список сервісів для опублікування. Для кожного сервісу, який публікується, надається опис з вказанням типу сервісу, імені сервісу, мов і онтологій необхідних, необхідних для його використання та інші властивості специфічні для цього сервісу.  Класи DFAgentDescription, ServiceDescription, Property, які входять у пакет jade.domain.FIPAAgentManagement представляють ці три абстракції.  Для публікації сервісу агент повинен створити необхідний опис (екземпляр класу DFAgentDescription) і викликати статичний метод register() класу DFService. Зазвичай, але не обов’язково, реєстрація сервісу відбувається в методі setup(), як показано, нижче, у випадку з агентом-продавцем.  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();  }  ...  }  Потрібно зауважити що в цьому прикладі не визначені: мова, онтологія та інші специфічні для сервісу властивості.  Коли агент завершує свою роботу потрібно відмінити реєстрацію опублікованих сервісів.  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.”);  } |
| * + 1. ***Пошук сервісів***   Агент, який хоче здійснити пошук сервісів повинен надати DF опис згідно шаблона. Результатом пошуку є список всіх описів які відповідають наданому опису. Опис відповідає шаблону, якщо всі поля визначені в шаблоні присутні у описі з тими самими значеннями.  Статичний метод search() класу DFService може бути використаний як у наступному прикладі, де агент покупець динамічно знаходить всіх агентів, які надають сервіс типу «книготоргівля».  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());  }  } );  ...  Потрібно зауважити, що оновлення списку відомих агентів-продавців, відбувається перед кожною спробою купити необхідну книжку, оскільки агенти-продавці можуть динамічно з’являтися і зникати в системі. Клас DFService також надає підтримку підписки на DF, для отримання повідомлення, про публікацію агентом необхідної служби (див. методи searchUntilFound(), createSubscriptionMessage()). |
|  |
|  |

**ПОРЯДОК ВИКОНАННЯ РОБОТИ**

1. Ознайомитися з теоретичними відомостями.
2. Реалізувати описану в теоретичних відомостях мультиагентну систему.
3. Виконати запуск тестових агентів.
4. Запропонувати модифікацію описаних в теоретичних відомостях агентів.
5. Реалізувати модифіковану мультиагентну систему.
6. Підготувати і оформити звіт.

**ЗМІСТ ЗВІТУ**

5.1 Титульний аркуш.

5.2 Мета роботи.

5.3 Короткі теоретичні відомості.

5.4 Результати виконання роботи.

5.5 Висновок.

ЛІТЕРАТУРА

1. Bellifemine, F. L., Caire, G., та ін. Developing Multi-Agent Systems with JADE. : Wiley, 2007. – ISBN 0470057475.
2. *Giovanni Caire JADE TUTORIAL.JADE PROGRAMMING FOR BEGINNERS last update: 30 June 2009. JADE 3.7*

ІНТЕРНЕТ ПОСИЛАННЯ

<http://jade.tilab.com>

<http://www.agentlab.ru>

НАВЧАЛЬНЕ ВИДАННЯ

**МЕТОДИЧНІ ВКАЗІВКИ**

до лабораторної роботи № 2

з дисципліни “Методи проектування мультиагентних систем”

для студентів спеціальності 7.030.505 “Системне проектування”

та магістрів за фахом 8.030.505 “Системне проектування”.

для стаціонарної та заочної форм навчання

## **Укладачі:** Романюк Андрій Богданович

## 