

Technická univerzita Košice  
Fakulta elektrotechniky a informatiky  
Katedra počítačov a informatiky

Semestrálny projekt  
Agentový systém JADE

**Zadávatel:** Ing. Marek Paralič, PhD.  
**Vypracoval:** Patrik Bóna  
**Školský rok:** 2003/2004

# Obsah

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Úvod</b>                                       | <b>1</b>  |
| <b>2</b> | <b>FIPA</b>                                       | <b>1</b>  |
| <b>3</b> | <b>JADE</b>                                       | <b>1</b>  |
| 3.1      | Kontajnery a platformy . . . . .                  | 1         |
| 3.2      | AMS a DF agenty . . . . .                         | 2         |
| 3.3      | Trieda Agent . . . . .                            | 2         |
| 3.3.1    | Vytvorenie agenta . . . . .                       | 2         |
| 3.3.2    | Kompilácia a spustenie agenta . . . . .           | 2         |
| 3.3.3    | Ukončenie života agenta . . . . .                 | 3         |
| 3.3.4    | Identifikácia agenta . . . . .                    | 3         |
| 3.3.5    | Predávanie argumentov agentom . . . . .           | 3         |
| 3.4      | Agentové úlohy, implementácia správanií . . . . . | 3         |
| 3.4.1    | Plánovanie a vykonávanie správanií . . . . .      | 4         |
| 3.4.2    | Rozdelenie správanií . . . . .                    | 4         |
| 3.4.3    | Plánovanie operácií na daný čas . . . . .         | 6         |
| 3.5      | Komunikácia prostredníctvom správ . . . . .       | 7         |
| 3.5.1    | Jazyk ACL . . . . .                               | 7         |
| 3.5.2    | Posielanie správ . . . . .                        | 7         |
| 3.5.3    | Prijímanie správ . . . . .                        | 8         |
| 3.5.4    | Filtrované prijímanie správ . . . . .             | 8         |
| 3.6      | Služba yellow pages . . . . .                     | 9         |
| 3.6.1    | Zverejňovanie služieb . . . . .                   | 9         |
| 3.6.2    | Vyhľadávanie služieb . . . . .                    | 10        |
| <b>4</b> | <b>Ukážkové agenty</b>                            | <b>10</b> |
| 4.1      | Agent meteorológ . . . . .                        | 11        |
| 4.2      | Agent zvedavec . . . . .                          | 11        |
| 4.3      | Kompilácia a spustenie agentov . . . . .          | 11        |
| <b>5</b> | <b>Záver</b>                                      | <b>12</b> |
| <b>6</b> | <b>Použitá literatúra</b>                         | <b>13</b> |
| <b>7</b> | <b>Príloha</b>                                    | <b>14</b> |

# 1 Úvod

Mojou úlohou v tomto semestrálnom projekte, je štúdia multiagentového systému JADE, pričom sa zameriam hlavne na možnosti, ktoré poskytuje JADE pri tvorbe agentov a na komunikáciu medzi agentmi prostredníctvom jazyka ACL (Agent Communication Language) vytvoreného organizáciou FIPA, pretože v tomto a mojom predchádzajúcom semestrálnom projekte (agentový systém Grasshopper) budem pokračovať formou diplomovej práce, ktorá je zameraná na možnosti interakcie medzi agentmi rôznych platforiem, konkrétne JADE a Grasshopper. Takisto súčasťou semestrálneho projektu budú ukážkové agenty, na ktorých ukážem použitie popisovaných prostriedkov, ktoré JADE pri tvorbe agentov ponúka.

## 2 FIPA

FIPA (The Foundation of Intelligent Physical Agents), je nezisková organizácia, ktorá vznikla v roku 1996 a je zameraná na tvorbu štandardov, pre súčinnosť rôznorodých agentov. Tento cieľ je dosahovaný medzinárodne schvaľovacími špecifikáciami, ktoré maximalizujú schopnosť interakcie medzi agentmi rôznych agentových systémov. Primárne sa zameriava na ACL (Agent Communication Language), jazyk ktorý umožňuje interakciu medzi agentmi rôznych platforiem. JADE plne implementuje štandardy FIPA.

## 3 JADE

JADE je softvérový Open Source systém plne implementovaný v jazyku Java, ktorý implementuje platformu pre multiagentové systémy. JADE obsahuje:

- **Prostredie pre beh agentov**, ktoré musí byť na danom hostiteľskom počítači aktívne ešte pred spustením akéhokoľvek agenta.
- **Knižnice tried**, ktoré môže programátor použiť k vývoju agentov.
- **Grafické prostredie**, umožňujúce administráciu a monitorovanie činnosti bežiacich agentov.

### 3.1 Kontajnery a platformy

Každá bežiacia inštancia prostredia JADE sa nazýva kontajner a môže obsahovať niekoľko agentov. Množina aktívnych kontajnerov sa nazýva platforma. Na každej platforme musí byť vždy špeciálny kontajner (hlavný kontajner), v ktorom sa registrujú všetky ostatné kontajneri pri ich štarte. To znamená, že pri štarte platformy musí existovať hlavný kontajner a všetky ostatné kontajneri musia byť normálne (nie hlavné) a musí im byť povedané

na akom počítači a porte sa nachádza hlavný kontajner, v ktorom sa majú zaregistrovať.

Programátor nemusí vedieť ako prostredie JADE funguje, stačí mu len pred vykonávaním jeho agentov spustiť prostredie.

## 3.2 AMS a DF agenty

Hlavný kontajner sa od ostatných líši aj tým, že obsahuje aj dva špeciálne agenty, ktoré sa automaticky spúšťajú pri vzniku kontajneru.

Prvým z týchto agentov je agent **AMS** (Agent Management System), ktorý sa stará o to aby každý agent na platforme mal jedinečné meno a stará sa o správu platformy, tým že je možné vytvárať a ukončovať agentov na vzdialených kontajneroch pomocou požiadavky na agenta AMS.

Ďalším špeciálnym agentom je agent **DF** (Directory Facilitator), ktorý poskytuje tzv. yellow pages službu, tzn. že agentom umožňuje nájsť iné agenty, ktoré potrebujú pre splnenie svojich úloh.

## 3.3 Trieda Agent

### 3.3.1 Vytvorenie agenta

Vytvorenie JADE agenta je také jednoduché ako definovanie triedy, ktorá rozširuje triedu `jade.core.Agent` a implementáciou metódy `setup()`, ktorá slúži na inicializáciu agenta.

```
import jade.core.Agent;

public class HelloWorldAgent extends Agent {
    protected void setup() {
        System.out.println("Hello World! I am " +getAID().getName());
    }
}
```

### 3.3.2 Kompilácia a spustenie agenta

Vyššie uvedený jednoduchý agent môže byť skompilovaný napríklad nasledujúcim spôsobom, ktorý je uvedený pre unixovú platformu a predpokladá sa, že balíky JADE sa nachádzajú v adresári `/home/head/jade/jade/lib/`.

```
$ export CLASSPATH=/home/head/jade/jade/lib/Base64.jar:\
> /home/head/jade/jade/lib/iiop.jar:\
> /home/head/jade/jade/lib/jade.jar:\
> /home/head/jade/jade/lib/jadeTools.jar:.
$ javac HelloWorldAgent.java
```

A následne ho môžeme spustiť spolu so štartom systému JADE.

```
$ java jade.Boot smith:HelloWorldAgent
```

### 3.3.3 Ukončenie života agenta

Aj keď náš jednoduchý agent nič nerobí, po uvítacom výpise je ešte stále aktívny. Ak ho chceme ukončiť tak musí byť zavolaná metóda `doDelete()`. Podobne ako pri spúšťaní agenta je volaná metóda `setup()`, tak pri jeho ukončovaní je volaná metóda `takeDown()`, aby mohol agent uvoľniť zdroje, ktoré predtým alokoval.

### 3.3.4 Identifikácia agenta

Každý agent je definovaný pomocou tzv. „identifikátoru agenta“, ktorý reprezentuje inštanciu triedy `jade.core.AID`. Metóda `getAID()` triedy `Agent` nám umožňuje získať identifikátor agenta. `AID` objekt obsahuje globálne jednoznačné meno agenta a zoznam adries. Meno agenta je vo forme `<prezývka>@<meno-platformy>`, teda ak máme agenta s prezývkou *Smith*, žijúceho na platforme *Matrix*, tak jeho globálne jednoznačné meno bude *Smith@Matrix*. Zoznam adries obsahuje adresy platforiem, na ktorých agent žije.

Keď poznáme prezývku agenta tak jeho `AID` môžeme získať nasledujúcim spôsobom:

```
String prezývka = "Smith";  
AID id = new AID(prezývka, AID.ISLOCALNAME);
```

Konštanta `ISLOCALNAME` indikuje to, že prvý parameter reprezentuje prezývku na lokálnej platforme a nieje to globálne jednoznačné meno agenta.

### 3.3.5 Predávanie argumentov agentom

JADE umožňuje agentom pri ich štarte predávať argumenty z príkazového riadku. Tieto argumenty môžu byť prijaté ako pole triedy `Object` vrátené metódou `getArguments()` z triedy `Agent`.

Argumenty sa z príkazového riadku agentu predávajú nasledujúcim spôsobom, pričom jednotlivé argumenty od seba oddeľujeme medzerou.

```
$ java jade.Boot "smith:AgentSmith(argument1 argument2 ...)"
```

## 3.4 Agentové úlohy, implementácia správania

Správanie reprezentuje úlohu, o ktorú sa má agent postarať a je implementované ako objekt triedy rozširujúcu triedu `jade.core.behaviours.Behaviour`. Aby agent danú úlohu vykonal je potrebné agentu pridať správanie pomocou metódy `addBehaviour()` z triedy `Agent`. Správanie môže byť agentu pridané kedykoľvek, či už pri štarte, alebo počas behu agenta.

Každá trieda rozširujúca triedu `Behaviour` musí implementovať metódu `action()`, ktorá definuje operáciu, ktorá sa má splniť počas vykonávania daného správania. Ďalej musí mať takáto trieda implementovanú metódu `done()` vracajúcu `boolean` hodnotu, ktorá špecifikuje, či bolo správanie dokončené. Ak bolo správanie dokončené (metóda `done()` vrátila hodnotu `true`), tak sa automaticky vyradí zo zoznamu správání, ktoré sa majú vykonať.

Každé správanie má chránenú premennú `myAgent`, ktorá slúži ako referencia na agenta, ktorý správanie vykonáva. Ďalej treba spomenúť metódu `block()`, ktorá vyradí správanie z fronty vykonávania, pokiaľ agent neprijme nejakú správu.

### 3.4.1 Plánovanie a vykonávanie správání

Agent môže vykonávať súbežne niekoľko správání. Každopádne je dôležité poznamenať, že plánovanie nie je preemptívne ale kooperatívne. To znamená, že keď je naplánované vykonanie správania, tak sa zavolá jeho metóda `action()`, ktorá beží pokiaľ sa neukončí a až potom sa môže začať vykonávať ďalšie správanie.

Takéto riešenie má niekoľko výhod:

- Umožňuje mať len jedno Java vlákno pre každého agenta, čo je výhodné pri prostrediach s obmedzenými zdrojmi, ako napríklad mobilné telefóny.
- Poskytuje lepšiu výkonnosť, pretože prepínanie medzi správaniami je oveľa rýchlejšie ako prepínanie medzi Java vláknami.
- Nie je potrebná žiadna synchronizácia medzi správaniami, ktoré prístupujú k tomu istému zdroju.
- Je možné vytvoriť aktuálny záznam stavu agenta, čo umožňuje implementáciu rôznych vylepšení, ako napr. uloženie stavu agenta, pre neskoršie obnovenie (perzistencia agenta), alebo premiestnenie agenta do iného kontajneru (mobilita agenta).

### 3.4.2 Rozdelenie správání

Môžeme rozlišovať medzi tromi druhmi správání.

1. „Jednorázové správanie“ – metóda `action()` tohto správania sa vykoná len raz. Trieda `jade.core.behaviours.OneShotBehaviour`, už má implementovanú metódu `done()` vracajúcu hodnotu `true`.

Ako vidno na nasledujúcom príklade implementovať jednorázové správanie môžeme rozšírením spomínanej triedy.

```

public class JednorazoveSpravanie extends OneShotBehaviour {
    public void action() {
        // operacia X
    }
}

```

Operácia X sa vykoná len raz.

2. „Cyklické správanie“ – toto správanie sa nikdy neukončí a metóda `action()` tohto správania vykonáva vždy tú istú operáciu. Cyklické správanie môžeme implementovať rozšírením triedy `jade.core.behaviours.CyclicBehaviour`, ktorá má implementovanú metódu `done()` vracajúcu hodnotu `false`.

Nasledujúci príklad znázorňuje implementáciu cyklického správania rozšírením spomínanej triedy.

```

public class CyklickeSpravanie extends CyclicBehaviour {
    public void action() {
        // operacia Y
    }
}

```

Operácia Y sa bude vykonávať donekonečna, alebo pokiaľ agent nezruší dané správanie.

3. „Generické správanie“ – pri tomto správaní sa vykonávajú rôzne operácie v závislosti na aktuálnom stave správania. Správanie je splnené vtedy, keď je splnená daná podmienka.

Implementácia generického správania rozšírením triedy `Behaviour`:

```

public class DvojKrokovSpravanie extends Behaviour {
    private int krok = 0;
    public void action() {
        switch(krok) {
            case 0:
                // operacia X
                krok++;
                break;
            case 1:
                // operacia Y
                krok++;
                break;
        }
    }
}

```

```

    }

    public boolean done() {
        return krok == 2;
    }
}

```

Najprv sa vykoná operácia X, potom operácia Y a tým pádom je správanie ukončené.

JADE umožňuje kombinovať tieto jednoduché správania a pomocou nich vytvárať správania komplexnejšie. Pre hlbší pohľad nato aké hotové správania JADE poskytuje odporúčam [3].

### 3.4.3 Plánovanie operácií na daný čas

JADE poskytuje v balíku `jade.core.behaviours` dve triedy, pomocou ktorých sa dá jednoducho implementovať správanie, ktoré sa má vykonať v danom čase.

1. Prvou z nich je trieda `WakerBehaviour`, ktorej metódy `action()` a `done()` sú implementované tak, aby po uplynutí daného časového limitu (špecifikovaného v konštruktore) bola vykonaná abstraktná metóda `handleElapsedTimeout()`, po ktorej vykonaní je správanie ukončené.

Takéto správanie môžeme pridať nasledujúcim spôsobom:

```

addBehaviour(new WakerBehaviour(this, 1000) {
    protected void handleElapsedTimeout() {
        // operacia X
    }
});

```

Operácia X bude vykonaná jednu sekundu po pridaní správania.

2. Ďalšou je trieda `TickerBehaviour`, ktorej metódy `action()` a `done()` sú implementované tak, aby bola periodicky vykonaná abstraktná metóda `onTick()` po uplynutí danej časovej periódy (zadanej v konštruktore). Toto správanie sa nikdy neukončí.

Takéto správanie môžeme pridať nasledujúcim spôsobom:

```

addBehaviour(new TickerBehaviour(this, 5000) {
    protected void onTick() {
        // operacia Y
    }
});

```

Operácia Y bude vykonaná každých 5 sekúnd.

## 3.5 Komunikácia prostredníctvom správ

### 3.5.1 Jazyk ACL

Správy vymieňané medzi agentmi v JADE majú formát špecifikovaný podľa jazyka ACL, ktorý vytvorila FIPA. Správa je v JADE implementovaná ako objekt triedy `jade.lang.acl.ACLMessage`. ACL formát správy obsahuje niekoľko položiek:

- Odosielateľ správy (**sender**)
- Zoznam príjemcov správy (**receivers**)
- Zámer komunikácie, alebo performatíva (**performative**) – indikuje zámer prečo agent správu poslal. Tento zámer sa zadáva ako číselná hodnota v konštruktoze pri vytváraní inštancie správy. Tieto číselné konštanty sú definované ako statické v triede `ACLMessage`.
- Obsah správy (**content**)
- Obsahový jazyk (**language**) – syntax použitá v obsahu správy.
- Ontológia (**ontology**) – zoznam symbolov v obsahu a ich význam.
- Položky slúžiace na kontrolu viacerých navzájom si konkurujúcich komunikácií ako **conversation-id**, **reply-with**, **in-reply-to**, **reply-by**.

Okrem parametrov, ktoré sú definované v špecifikácii môžeme prostredníctvom metódy `addUserDefinedParameter()` pridávať vlastné parametre. Zoznam týchto parametrov potom môžeme získať pomocou volania metódy `getAllUserDefinedParameters()`.

Hodnota položky sa dá nastaviť pomocou metódy `set<položka>` a zistiť sa dá pomocou metódy `get<položka>`.

### 3.5.2 Posielanie správ

Na poslanie správy inému agentu, stačí nastaviť potrebné položky v objekte `ACLMessage` a následne zavolať metódu `send()` z triedy `Agent`. Nasledujúci kód pošle agentu s lokálnym menom *Neo* informatívnu správu *Matrix has you*.

```
ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
msg.addReceiver(new AID("Neo", AID.ISLOCALNAME));
msg.setLanguage("English");
msg.setOntology("Matrix-ontology");
msg.setContent("Matrix has you");
send(msg);
```

### 3.5.3 Prijímanie správ

JADE automaticky uloží správu do fronty správ príjemcu ihneď po jej doručení. Agent môže správy z fronty správ vyberať volaním metódy `receive()`. Táto metóda vracia prvú správu z fronty správ, alebo `null` ak je fronta správ prázdna. Jednoduchý kód prijímajúci správu je uvedený v nasledujúcom príklade.

```
ACLMessage msg = receive();
if(msg != null) {
    // spracovanie spravy
}
```

Na prijímanie správ môžeme použiť aj blokovaciu metódu `blockingReceive()` z triedy `Agent`. Táto metóda na rozdiel od metódy `Receive()` v prípade prázdnej fronty správ nevracia hodnotu `null`, ale čaká pokiaľ sa vo fronte správ nevyskytne správa. Je dôležité poznamenať, že táto metóda v prípade prázdnej fronty správ blokuje vlákno agenta, preto by sa nemala volať z implementovaného správania, pretože zablokuje vykonanie všetkých ostatných správanií pokiaľ sa sama neukončí.

JADE umožňuje prijímanie správ aj prostredníctvom správania implementovaného v triede `ReceiverBehaviour`. Takisto je možné implementovať vlastné správanie na príjem správ a vytvoriť tak správanie pre komplexnú konverzáciu, ktorá prebieha podľa definovaného interakčného protokolu. JADE bohaté poskytuje implementáciu konverzácií pomocou interakčných protokolov obsiahnutých v balíku `jade.proto`. Viac o správaniach pre prácu so správami a o interakčných protokoloch sa nachádza v [3].

### 3.5.4 Filtrované prijímanie správ

Na filtrované prijímanie správ nám slúžia šablóny správ. Šablóna správy je reprezentovaná triedou `MessageTemplate` z balíka `jade.lang.ac1`. Šablóny správ môžeme použiť napríklad s metódami `receive(MessageTemplate)`, `blockingReceive(MessageTemplate)`...

V šablóne môžeme pre každý atribút správy nastaviť podmienku podľa príslušnej metódy. Správa vyhovuje šablóne v tom prípade, ak sa každá nastavená položka v šablóne má tú istú hodnotu, ako zodpovedajúca položka v správe. Z jednotlivých šablón môžeme pomocou logických operácií vytvárať komplexnejšie šablóny, alebo si môžeme implementovať interface `MessageTemplate.MatchExpression` obsahujúci metódu `match()`, ktorú implementujeme tak, aby podľa toho, či správa vyhovuje vracala `boolean` hodnotu.

Vytvorenie šablóny `finalTemplate` akceptujúcej správy s performatívou *inform* a jazykom *english*, ktoré nemajú ontológiu *bad-ontology* je uvedené v nasledujúcom príklade.

```

MessageTemplate m1 = MessageTemplate.MatchPerformative(ACLMessage.INFORM);
MessageTemplate m2 = MessageTemplate.MatchLanguage("english");
MessageTemplate m3 = MessageTemplate.MatchOntology("bad-ontology");
MessageTemplate m1andm2 = MessageTemplate.and(m1,m2);
MessageTemplate notm3 = MessageTemplate.not(m3);
MessageTemplate finalTemplate = MessageTemplate.and(m1andm2, notm3);

```

### 3.6 Služba yellow pages

Ako už bolo spomenuté vyššie (3.2), agenty môžu prostredníctvom DF agenta zverejňovať služby, ktoré ponúkajú iným agentom a takisto pomocou DF agenta môžu agenty potrebnú službu vyhľadať. Táto služba, ktorú DF agent poskytuje sa nazýva yellow pages.

Podľa FIPA špecifikácie je komunikácia s DF agentom umožnená pomocou ACL správ pričom ako obsahový jazyk sa používa jazyk *SLO* a ako ontológia sa používa *FIPA-agent-management ontológia*. Na uľahčenie tejto komunikácie JADE poskytuje triedu `jade.domain.DFService`, pomocou ktorej je možné zverejňovať a vyhľadávať požadované služby volaním metód danej triedy.

#### 3.6.1 Zverejňovanie služieb

Ak chce agent zverejniť služby, ktoré poskytuje, tak musí oboznámiť DF agenta so svojím AID, jazykmi a ontológiami, ktoré su potrebné pre interakciu s agentom a v neposlednom rade so zoznamom služieb. Pre každú ponákanú službu musí popis obsahovať, typ a meno služby, jazyky a ontológie potrebné pre využitie služby a vlastnosti služby. Triedy `DFAgentDescription`, `ServiceDescription` a `Property`, ktoré sa nachádzajú v balíku `jade.domain.FIPAAgentManagement` reprezentujú spomínané abstrakcie.

Aby agent mohol zverejniť ponákanú službu, musí vytvoriť správny popis (ako inštanciu triedy `DFAgentDescription`) a zavolať statickú metódu `register()` z triedy `DFService`. Príklad registrácie služby *testovacia-sluzba* s názvom *nasa-testovacia-sluzba* môžeme vidieť na nasledujúcom príklade.

```

DFAgentDescription dfd = new DfAgentDescription();
dfd.setName(getAID());
ServiceDescription sd = new ServiceDescription();
sd.setType("testovacia-sluzba");
sd.setName("nasa-testovacia-sluzba");
dfd.addServices(sd);
try {
    DFService.register(this, dfd);
} catch (FIPAException fe){
    fe.printStackTrace();
}

```

Dobrým zvykom pri ukončení agenta je zrušenie registrácie ponúkaných služieb v metóde `takeDown()`. Jednoduchá ukážka takejto metódy je na nasledujúcom príkade.

```
protected void takeDown() {
    try {
        DFService.deregister(this);
    } catch (FIPAException fe) {
        fe.printStackTrace();
    }
    System.out.println(getAID().getName() + " terminated.");
}
```

### 3.6.2 Vyhľadávanie služieb

Ak si agent želá vyhľadávať služby, tak musí DF agentu poskytnúť popisnú šablónu. Výsledok vyhľadávania bude zoznam všetkých služieb, ktorých popis zodpovedal popisnej šablóne. Popis služby zodpovedá šablóne vtedy, ak každá položka šablóny má rovnakú hodnotu ako položka z popisu.

Na nasledujúcom príklade môžeme vidieť kód, ktorý vyhľadá a vypíše všetky agenty ponúkajúce typ služby *testovacia-sluzba*.

```
DFAgentDescription template = new DFAgentDescription();
ServiceDescription sd = new ServiceDescription();
sd.setType("testovacia-sluzba");
template.addServices(sd);
try {
    DFAgentDescription[] result = DFService.search(this, template);
    for(int i = 0; i < result.length; i++) {
        System.out.println(result[i].getName().getName());
    }
} catch(FIPAException fe) {
    fe.printStackTrace();
}
```

## 4 Ukážkové agenty

Ako praktickú ukážku toho čo JADE ponúka a toho čo som v tejto práci popisoval, som vytvoril dva agenty. Prvým je agent *meteorológ*, ktorý monitoruje stav počasia v danom meste a poskytuje informácie o počasi tom, kto o ne požiada. Druhým je agent *zvedavec*, ktorý je zvedavý na stav počasia v rôznych mestách a preto tieto informácie požaduje od rôznych meteorológov.

Obidva agenty sa nachádzajú v balíku *pocasio*, ich zdrojové texty s komentármi ako agenty pracujú sú uvedené v prílohe.

## 4.1 Agent meteorológ

Agent meteorológ monitoruje počasie v meste, ktoré mu zadáme ako parameter pri spúšťaní agenta. Pri spustení sa agent pomocou služby yellow pages zaregistruje ako poskytovateľ služby *stav-pocasia* a spustí grafické prostredie, pomocou ktorého môžeme zadávať aktuálne počasie.

Ak agent prijme správu s požiadavkou na informáciu o počasí, tak agentu, ktorý túto správu poslal automaticky odpovie. Spracovanie správ je implementované v triede `SpracovanieSprav`, ktorá rozširuje triedu `CyclicBehaviour`.

## 4.2 Agent zvedavec

Tento agent, po svojom štarte pomocou yellow pages služby vyhľadá všetkých agentov meteorológov a pošle im správu s požiadavkou na informáciu o aktuálnom stave pocasia. Toto je implementované ako trojkrokové správanie vo vnorenej triede `ZistenieStavuPocasia`, ktorá rozširuje triedu `Behaviour`. O to aby sa toto správanie vykonávalo každých 60 sekúnd sa stará správanie `TickerBehaviour`, ktoré poskytuje samotný JADE.

## 4.3 Kompilácia a spustenie agentov

Pred samotnou kompiláciou agentov musíme nastaviť cestu k balíkom JADE ako je uvedené v časti 3.3.2. Potom môžeme agenty skompilovať nasledujúcim spôsobom.

```
$ javac Meteorolog.java
$ javac Zvedavec.java
```

Na nasledujúcom príklade môžeme vidieť spustenie dvoch agentov meteorológov, jedného pre mesto Košice, druhého pre Chopok a jedného agenta zvedavca.

```
$ java jade.Boot "m1:pocasia.Meteorolog(Kosice)"
$ java jade.Boot -container "m2:pocasia.Meteorolog(Chopok)"
$ java jade.Boot -container zvedavec:pocasia.Zvedavec
```

## 5 Záver

V tejto práci som sa venoval multiagentovému systému JADE, pričom som moje úsilie zamerlal hlavne rozbor toho, čo JADE poskytuje programátorovi pri tvorbe agentov.

Jednoducho som popísal ako asi funguje prostredie JADE, ukázal som ako je možné vytvoriť agenta pre tento systém, ako agenta skompilovať, spustiť, predávať mu pri štarte parametre. Ďalej som ukázal, ako pomocou správania implementovať úlohy, ktoré má agent vykonávať. Takisto bolo ukázané, ako môžu agenty navzájom medzi sebou komunikovať prostredníctvom ACL správ a ako môžu prostredníctvom služby yellow pages, vyhľadať iné agenty poskytujúce požadovanú službu, alebo zverejniť služby, ktoré agent ponúka.

Vytvoril som aj dva ukážkové agenty, na ktorých môžeme vidieť praktické využitie spomínaných možností, ktoré JADE pri tvorbe agentov ponúka a to implementáciu správania, komunikáciu pomocou ACL správ, zverejňovanie a vyhľadávanie služieb prostredníctvom služby yellow pages.

Keďže rozsiahlejší popis systému JADE by presiahol rámec tejto práce, tak na dôkladnejšie preštudovanie tohto systému odporúčam spomínanú použitú literatúru.

## 6 Použitá literatura

[1] JADE Programming tutorial for beginners

<http://sharon.cselt.it/projects/jade/doc/JADEProgramming-tutorial-for-beginners.pdf>

[2] JADE Administrators's guide

<http://sharon.cselt.it/projects/jade/doc/administratorsguide.pdf>

[3] JADE Programmer's guide

<http://sharon.cselt.it/projects/jade/doc/programmersguide.pdf>

## 7 Príloha

Zdrojový text agenta Meteorolog:

```
/*
 * Semestrálny projekt,
 * Multiagentový systém JADE
 * subor: Meteorolog.java
 * autor: Patrik Bona <bona@intrak.sk>
 * zadavateľ: Ing. Marek Paralic, PhD.
 * sk. rok: 2003/2004
 * posledná zmena: 12. 1. 2004
 */

package pocasie;

import jade.core.Agent;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;
import jade.domain.FIPAAException;
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.*;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.lang.String;

/** Trieda implementuje agenta Meteorolog */
public class Meteorolog extends Agent {
    private GUI gui;
    private String mesto;

    /** Metóda slúži na inicializáciu agenta */
    protected void setup() {
        System.out.println("Hello, I am " +getAID().getName());

        // kontrola zadanych argumentov
        Object[] args = getArguments();
        if(args == null || args.length != 1) {
            System.out.println("Agent bol spustený s nesprávnymi argumentami !!!");
            doDelete();
            return;
        }
    }

    mesto = (String) args[0];

    // pridanie spravy starajúceho
    // sa o spracovanie sprav
    addBehaviour(new SpracovanieSprav());
}
```

```

// registracia u DF agenta (yellow pages sluzba)
DFAgentDescription dfd = new DFAgentDescription();
dfd.setName(getAID());
ServiceDescription sd = new ServiceDescription();
sd.setType("stav-pocasia");
sd.setName("meteorologicky-stav-pocasia");
dfd.addServices(sd);
try {
    DFService.register(this, dfd);
} catch (FIPAException fe){
    fe.printStackTrace();
}

// zobrazenie grafickeho prostredia pre vstup udajov
gui = new GUI(this);
gui.show();
}

/** Metoda sa vola pri ukoncovani zivotu agenta */
protected void takeDown() {
    // pokusime zrusit registraciu sluzieb
    // u DF agenta
    try {
        DFService.deregister(this);
    } catch (FIPAException fe) {
        // fe.printStackTrace();
    }

    gui.dispose();
    System.out.println(getAID().getName() + " terminated.");
}

/** Trieda implementuje pouzivatel'ske prostredie
 * pre zadavanie stavu pocasia... */
private class GUI extends JFrame {
    private JTextField stupneField, komentarField;
    private Meteorolog myAgent;

    /** Metoda vrati <code>String</code> udavajuci stav pocasia */
    public String getStav() {
        return mesto + "\t\t" + stupneField.getText() +
            " stupnov celzia \t(" + komentarField.getText() + ")";
    }
}

GUI(Meteorolog m) {
    super(mesto);
    myAgent = m;
}

```



Zdrojový text agenta Zvedavec:

```
/*
 * Semestrálny projekt,
 * Multiagentový systém JADE
 * subor: Zvedavec.java
 * autor: Patrik Bona <bona@intrak.sk>
 * zadavateľ: Ing. Marek Paralic, PhD.
 * sk. rok: 2003/2004
 * posledná zmena: 12. 1. 2004
 */

package pocasie;

import jade.core.Agent;
import jade.core.behaviours.*;
import jade.lang.acl.ACLMessage;
import jade.domain.FIPAException;
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.*;
import jade.core.AID;

import java.util.Vector;
import java.lang.String;

/**Trieda implementuje agenta Zvedavec */
public class Zvedavec extends Agent {
    /** metóda slúži na inicializáciu agenta */
    protected void setup() {
        System.out.println("Hello World! I am " +getAID().getName());

        // pridáme správanie na zistenie stavu počasie
        addBehaviour(new ZistenieStavuPocasia());

        // a zaistíme nech je správanie ZistenieStavuPocasia
        // spustené každých 60 sekúnd
        addBehaviour(new TickerBehaviour(this, 60000) {
            protected void onTick() {
                myAgent.addBehaviour(new ZistenieStavuPocasia());
            }
        });
    }

    /** metóda sa volá pri ukončení života agenta */
    protected void takeDown() {
        System.out.println(getAID().getName() + " terminated.");
    }

    /** trieda implementuje trojkrokové správanie
```

```

* zaistujuce zistovanie stavu pocasia */
private class ZistenieStavuPocasia extends Behaviour {
    private int krok = 0;
    /** pocet najdenych meteorologov */
    private int mcnt;
    private int count = 0;
    private Vector v = new Vector();
    public void action() {
        switch(krok) {
            // v prvom kroku najdeme agentov meteorologov
            // a posleme im poziadavku na informaciu
            // o stave pocasia
            case 0:
                DFAgentDescription template = new DFAgentDescription();
                ServiceDescription sd = new ServiceDescription();
                sd.setType("stav-pocasia");
                template.addServices(sd);
                try {
                    DFAgentDescription[] result = DFService.search(myAgent, template);
                    ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
                    if(result.length == 0) {
                        System.out.println("Nenasiel som ziadneho meteorologa !");
                        krok = 3;
                        break;
                    }
                }
                for(int i = 0; i < result.length; i++) {
                    msg.addReceiver(result[i].getName());
                }
                mcnt = result.length;
                // kedze obidva agenty su zjednodusene priklady pouzitia
                // systemu JADE, tak budeme nastavovat len obsah spravy,
                // teda nenastavime komunikacny jazyk, ontologiu...
                msg.setContent("pocasia");
                send(msg);
            } catch(FIPAException fe) {
                fe.printStackTrace();
            }
            krok = 1;
            break;
            // v druhom kroku spracujeme odpovede, ktore prisli
            // a pridame ich do vectoru v
            case 1:
                ACLMessage msg = receive();
                if(msg != null) {
                    if(msg.getPerformative() != ACLMessage.REFUSE) {
                        v.add(msg.getContent());
                    }
                    count++;
                } else { // ak sme neprijali spravu

```

```

        // tak zablokujeme spravanie
        // do prichodu dalsej spravy
        block();
    }
    if(mcnt == count ) {
        krok = 2;
    }
    break;
// v tretom kroku do konzoly vypiseme
// informacie a aktualnom stave pocasia
case 2:
    System.out.println("Aktualne pocasia (" +
        getAID().getName() + ")");
    //System.out.println(v.toString());
    String pom = "";
    for(int i = 0; i < v.size(); i++) {
        System.out.println(v.get(i));
        pom = pom + v.get(i);
    }
    System.out.println("-----");
    krok = 3;
}
}

// ak sa skoncil tretí krok
// tak spravanie je ukoncene
public boolean done() {
    return(krok == 3);
}
}
}

```