

Przetwarzanie Zespołowe

Laboratorium nr 5

Środowisko agentowe JADE
część 2 – Migracja agentów

Boński Tomasz

Wstęp

Podstawową zaletą środowisk agendowych jest możliwość przemieszczania się agentów pomiędzy kontenerami. Dzięki temu agent taki ma możliwość przeszukiwania i gromadzenia o wiele szerszego zakresu informacji niż tylko te dostępne lokalnie. W trakcie tych zajęć utworzony zostanie prosty agent, którego zadaniem będzie przemieszczenie się pomiędzy wszystkimi kontenerami wchodzącymi w skład platformy i pozostawienie w nich jakiegoś śladu.

Przygotowanie środowiska

Uruchomienie środowiska następuje identycznie jak na poprzednim laboratorium. Należy pamiętać, by CLASSPATH tym razem wskazywał nie na katalog z przykładami a na katalog, gdzie zawarty jest kod tworzonego agenta mobilnego.

Tworzenie agenta mobilnego

Kodowanie agenta najlepiej przeprowadzić w jakimś IDE, które ułatwi nam tą jednak dość skomplikowaną sprawę. W Netbeans należy utworzyć standardową aplikację Java, a następnie dodać (klikając prawym klawiszem na Libraries i wybierając Add Jar/Folder) wszystkie biblioteki wchodzące w skład JADE (czyli te, które dodawano wcześniej do CLASSPATH). Dzięki temu Netbeans będzie podpowiadał klasy i pozwalał na weryfikację kodu na bieżąco. Automatycznie też zapewni importowanie odpowiednich klas wykorzystywanych w trakcie zajęć.

Agenta mobilnego stworzymy jak każdego innego agenta, rozpoczynamy od klasy agenta która dziedziczy po klasie Agent.

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

Agent ten będzie migrował pomiędzy różnymi kontenerami, więc potrzebne jest nam pole przechowujące kontenery, jakie pozostały do odwiedzenia. Jak atrybut klasy dodajemy więc:

```
private Vector locations;
```

a wewnątrz metody setup() (będącej swoistym konstruktorem) inicjalizację tego pola.

```
locations = new Vector();
```

Aby Agent mógł się przemieszczać konieczne jest pobranie listy dostępnych lokacji. Procedura ta polega na odpytaniu Głównego Kontenera o ontologie platformy. Odbywa się ona poprzez standardową wymianę komunikatów w języku SL0. Z kolei interpretacja otrzymanej listy odbywać się będzie na podstawie ontologii opisującej konfigurację platformy. Stąd konieczne jest zarejestrowanie w agencie następujących usług:

```
// rejestracja obsługi języka komunikatów SLO
getContentManager().registerLanguage(new SLCodec(), FIPANames.ContentLanguage.FIPA_SLO);
// rejestracja ontologii
getContentManager().registerOntology(MobilityOntology.getInstance());
```

Oba te wywołania należy dodać do metody setup, gdyż są częścią inicjalizacji agenta.

Środowisko JADE definiuje kilka standardowych metod agenta, które są automatycznie wykonywane w wyniku zajścia jakiejś akcji. Ich bazowe implementacje są dziedziczone po klasie Agent, jednak zalecane jest ich zaimplementowanie, co umożliwi np. komunikację z użytkownikiem czy wykonanie jakichś istotnych operacji przed zmigrowaniem do innego kontenera. Są to między innymi:

- `takeDown()` - metoda wykonywana bezpośrednio przed zabiciem agenta (najczęściej wywołanym poprzez metodę `doDelete()`);
- `beforeClone()` - metoda wykonywana bezpośrednio przed sklonowaniem agenta albo poprzez akcję interfejsu albo poprzez wykonanie metody `doClone(Location kont, String nazwa)`;
- `afterClone()` - metoda wykonana bezpośrednio po sklonowaniu agenta
- `beforeMove()` - metoda wykonana bezpośrednio przed przeniesieniem agenta do docelowego kontenera (zazwyczaj za pomocą metody `doMove(Location kontener)`);
- `afterMove()` - metoda wykonywana bezpośrednio po przeniesieniu agenta
- `afterLoad()` - metoda wykonywana bezpośrednio po załadowaniu agenta
- `beforeFreeze()` - metoda wywoływana bezpośrednio przed wstrzymaniem działania agenta, zazwyczaj wykonywanego metodą `doFreeze()`
- `afterThaw()` - metoda wykonywana bezpośrednio po powrocie agenta ze stanu wstrzymania
- `beforeReload()` - metoda wykonywana bezpośrednio przed reinicjalizacją agenta
- `afterReload()` - metoda wykonana bezpośrednio po reinicjalizacji agenta

W naszym przypadku, aby w pełni wykorzystać i zobrazować wędrówkę agenta, konieczne jest zaimplementowanie metod `beforeMove()` oraz `afterMove()` tak, by agent informował nas o swoim bieżącym położeniu. Jeżeli agent po wędrówce miałby móc odbierać komunikaty tak jak na oryginalnym kontenerze, konieczne jest ponowne wykonanie rejestracji odpowiednich usług. W naszym przykładzie to nie będzie konieczne, jedna jeżeli agent miałby odświeżać listę kontenerów po każdej migracji konieczne by było ponowne zarejestrowanie obsługi języka SLO, ontologii platformy oraz pobranie dostępnych lokalizacji (o tym później).

```
protected void beforeMove() {
    System.out.println(getLocalName() + " is now moving elsewhere.");
}

protected void afterMove() {
    System.out.println(getLocalName() + " is just arrived to this location.");
}
```

Metoda `getLocalName()` zwraca nazwę agenta nadaną mu w trakcie jego tworzenia.

W naszej implementacji potrzebne będą jeszcze metody `set` i `get` umożliwiające dostęp do wektora

zawierającego możliwe do odwiedzenia lokacje.

```
public Vector getLocations() {
    return locations;
}

public void setLocations(Vector locations) {
    this.locations = locations;
}
```

Nasz agent jest już prawie gotowy. Potrzebne są jeszcze dwa kluczowe elementy: pobieranie listy dostępnych lokacji oraz właściwe przemieszczanie się agenta w obrębie platformy. Obie te czynności są przykładem zachowania agenta i będą zaimplementowane poprzez klasy dziedziczące po Behaviour.

Zajmijmy się najpierw tym drugim elementem. Jego implementacja jest o wiele prostsza. Opiera się o znane już z poprzedniego laboratorium wywołanie klasy TickerBehaviour, które dodajemy wewnątrz metody setup naszego agenta.

```
addBehaviour(new TickerBehaviour(this, 6000) {
    protected void onTick() {
        if (((MobileAgent) myAgent).getLocations().size() > 0) {
            Location location =
                (Location) ((MobileAgent) myAgent).getLocations().remove(locations.size() - 1);

            System.out.println("Agent " + myAgent.getLocalName() +
                " trying to move to new location " + location.toString());

            myAgent.doMove(location);
        } else {

            System.out.println("Agent " + myAgent.getLocalName() +
                " visited all possible places");
            myAgent.doDelete();
        }
    }
});
```

Co się tutaj dzieje? Do agenta dodajemy zachowanie (addBehaviour) będące klasą typu TickerBehaviour wykonywaną co określoną ilość czasu (w milisekundach). Zadaniem tej klasy jest sprawdzenie co wskazany odcinek czasu (poprzez automatyczne wykonanie metody onTick()) czy na liście znanych lokacji coś się jeszcze znajduje, a jeżeli tak to pobranie wartości z końca listy i przeniesienie tam agenta (doMove(Location lokacja;)). Jeżeli lista jest pusta metoda ta spowoduje zakończenie działania agenta poprzez wykonanie standardowej metody doDelete(); Na liście lokacji znajdują się odpowiednio przygotowane obiekty typu Location pobrane przez pierwsze z wspomnianych zachowań.

Aby agent zaczął poprawnie działać pozostaje zaimplementować obsługę pobierania dostępnych lokacji. Operacja ta jest na tyle skomplikowana, że utworzymy dla niej osobną klasę. Proszę więc w środowisku Netbeans utworzyć (w tym samym pakiecie co klasa agenta) klasę GetAvailableLocationsBehaviour.

Deklaracja tej klasy powinna wyglądać następująco:

```
public class GetAvailableLocationsBehaviour extends SimpleAchieveREInitiator
{

}
}
```

Klasa ta dziedziczy po SimpleAchieveREInitiator. Jest to specjalna klasa typu Behaviour stosowana do komunikacji z platformą poprzez język zgodny ze standardem FIPA.

Ciało tej klasy składa się z 3 elementów:

- pola przechowującego nadchodzącą wiadomość
- konstruktora definiującego parametry komunikacji itp.
- handlerów zajmujących się odbieraniem informacji ze środowiska

Pierwszy z tych elementów jest trywialny – pole to znany z poprzednich zajęć obiekt typu ACLMessage. Należy go po prostu zdefiniować:

```
private ACLMessage request;
```

Drugim wspomnianym elementem jest konstruktor klasy. Jego opis wykonany zostanie w postaci komentarzy do kodu:

```
public GetAvailableLocationsBehaviour(MobileAgent a) {

    //wywołujemy konstruktor klasy nadrzędnej, jako jej parametry musimy podać
    //referencję do agenta oraz typ wysyłanej wiadomości (Request - żądanie, gdyż
    //wysyłamy żądanie o opis środowiska)
    super(a, new ACLMessage(ACLMessage.REQUEST));

    //tworzymy obiekt żądania
    request = (ACLMessage) getDataStore().get(REQUEST_KEY);

    //następnie inicjujemy je zgodnie ze standardem FIPA odpowiednio:
    //usuwając wszystkich odbiorców
    request.clearAllReceiver();
    //dodając jako odbiorcę agenta zarządzającego platformą
    request.addReceiver(a.getAMS());
    //definiując język konwersacji
    request.setLanguage(FIPANames.ContentLanguage.FIPA_SLO);
    //określając ontologię opisującą środowisko
    request.setOntology(MobilityOntology.NAME);
    //oraz definiując protokół komunikacji
    request.setProtocol(FIPANames.InteractionProtocol.FIPA_REQUEST);

    //następnie wysyłamy wiadomość
    try {
        Action action = new Action();
        action.setActor(a.getAMS());
        //definiując co chcemy tak naprawdę od AMS uzyskać (listę lokalizacji)
```

```

        action.setAction(new QueryPlatformLocationsAction());
        a.getContentManager().fillContent(request, action);
    } catch (Exception fe) {
        fe.printStackTrace();
    }
    //aby możliwe było wysyłanie innych komunikatów należy zresetować obiekt
    //żądania
    reset(request);
}

```

Kolejnym etapem konstrukcji klasy jest zdefiniowanie uchwytów obsługujących wiadomości zwrotne. Konieczne jest zaimplementowanie 5 uchwytów:

- uchwytu obsługującego niezrozumiałe komunikaty

```

protected void handleNotUnderstood(ACLMessage reply) {
    System.out.println(myAgent.getLocalName() + " handleNotUnderstood : "
        + reply.toString());
}

```

- uchwytu obsługującego odmowy

```

protected void handleRefuse(ACLMessage reply) {
    System.out.println(myAgent.getLocalName() + " handleRefuse : "
        + reply.toString());
}

```

- uchwytu obsługującego błędy

```

protected void handleFailure(ACLMessage reply) {
    System.out.println(myAgent.getLocalName() + " handleFailure : "
        + reply.toString());
}

```

- uchwytu obsługującego potwierdzenia

```

protected void handleAgree(ACLMessage reply) {
}

```

- oraz najważniejszego z naszego punktu widzenia, uchwytu obsługującego informacje będące odpowiedzią na nasze zapytania

```

protected void handleInform(ACLMessage inform) {
    //pobranie treści wiadomości informacyjnej
    String content = inform.getContent();

    try {
        //i wyłuskanie z niego właściwej odpowiedzi będącej listą
        //dostępnych lokacji
        Result results = (Result) myAgent.getContentManager().extractContent(inform);

        //listę tą konwersujemy na wektor (by było łatwiej ją obsłużyć)
        Vector locations = new Vector();
        Iterator iter = results.getItems().iterator();
    }
}

```

```

    for (; iter.hasNext();) {
        Object obj = iter.next();
        locations.add((Location) obj);
    }
    //i przekazujemy do naszego agenta za pośrednictwem zdefiniowanej
//w nim metody setLocations
    ((MobileAgent)myAgent).setLocations(locations);

} catch (Exception e) {
    e.printStackTrace();
}
}

```

Agent jest prawie gotowy. Wystarczy jeszcze tylko do metody setup() agenta dopisać aktywację właśnie zaimplementowanego zachowania. Przed wcześniej zdefiniowanym zachowaniem wykonującym właściwą migrację dopisujemy linijkę:

```
addBehaviour(new GetAvailableLocationsBehaviour(this));
```

Agent jest już gotowy. Po jego kompilacji i uruchomieniu powinien on czytać listę kontenerów wchodzących w skład platformy a następnie „przejsć się” po ich wszystkich kończąc na kontenerze głównym.

Zadanie

1. Zaimplementować agenta mobilnego
2. Uruchomić go na platformie składającej się z przynajmniej 2 kontenerów położonych na przynajmniej 2 fizycznych komputerach
3. Agent ma informować o wykonywanych czynnościach
4. Zmodyfikować kod agenta tak, by po migracji pokazywał graficzne okienko z powitaniem