Systemy Agentowe

Laboratorium 3

Celem zadania jest skonfigurowanie rozproszonego środowiska wielo-agentowego oraz uruchomienie agenta migrującego pomiędzy kontenerami. Przy tworzeniu agenta można posłużyć się przykładem zaprezentowanym na wykładzie.

Podczas laboratorium zaleca się korzystanie ze środowiska NetBeans IDE oraz projektu budowanego przez Apache Maven. Aby móc korzystać z bibliotek JADE w projekcie należy w pliku `pom.xml` dodać odpowiednią zależność. W przypadku większości zewnętrznych bibliotek wystarczy jedynie dodać zależność do projektu, niestety w przypadku platformy JADE, biblioteki nie znajdują się w centralnym repozytorium. Z tego powodu wymagane jest, prócz zależności, dodanie do pliku `pom.xml` odpowiedniego repozytorium skąd mogą one zostać pobrane.

Zależność:

```
<project>
    ...
    <dependencies>
        <dependency>
            <groupId>com.tilab.jade</groupId>
            <artifactId>jade</artifactId>
            <version>4.3.2</version>
        </dependency>
        ...
    </dependencies>
    ...
</project>
```

Repozytorium:

```
<project>
    ...
    <repositories>
        <repository>
            <id>tilab</id>
            <url>http://jade.tilab.com/maven/</url>
        </repository>
        ...
    </repositories>
    ...
</project>
```


Należy zwrócić uwagę, że w przeciwieństwie to zazwyczaj wykonywanych projektów, nie przygotowuje się tutaj samemu głównej klasy programu a korzysta z tej dostarczonej już przez bibliotekę. Aby skonfigurować poprawne uruchamianie w NetBeans IDE należy wejść we właściwości projektu (prawy klik na projekt, properties), a następnie w zakładce run ustawić odpowiednie opcje. Jako main class należy ustawić `jade.Boot`. Pole arguments pozwala na przesłanie argumentów startowych dla jade, np `-gui`.

Zawartość projektu MigratingAgent:

- `pl.gda.pg.eti.kask.sa.migration.agents` – paczka zawierający agenty:
  - `MigratingAgent.java` – agent migrujący po środowisku;
- `pl.gda.pg.eti.kask.sa.migration.behaviours` – zachowania agentów:
  - `MigratinBehaviour.java` – zachowanie odpowiedzialne za migrację pomiędzy kontenerami,
  - `RequestContainersListBehaviou.java` – zachowanie żądania listy kontenerów dostępny na platformie,
  - `ReceiveContainerListBehaviour.java` – zachowanie odebrania listy kontenerów dostępnych na platformie.

Podstawową klasą agenta w środowisku JADE jest `jade.core.Agent`.

```
public class MigratingAgent extends Agent {

    @Override
    protected void setup() {
        super.setup();
        //register languages
        //register ontologies
        //add behaviours
    }

    @Override
    protected void afterMove() {
        super.afterMove();
        //restore state
        //resume threads
    }

    @Override
    protected void beforeMove() {
        //stop threads
        //save state
        super.beforeMove();
    }
}
```


Powyższy kod przedstawia prostą implementację własnego agenta. Zostały nadpisane trzy metody:

- `setup` – inicjalizacja agenta, wykonuje się po poprawnym stworzeniu agenta i osadzeniu go w środowisku,
- `afterMove` – wykonuje się po poprawnym przeniesieniu agenta do nowego kontenera,
- `beforeMove` – wykonuje się przed usunięciem agenta z kontenera podczas migracji.

Należy zwrócić uwagę że metody inicjalizujące (`setup` i `afterMove`) w pierwszej kolejności wykonują kod z klasy bazowej a metody sprzątające (`beforeMove`) wykonują go na końcu. Zachodzi tu taka sama zasada jak w przypadku konstruktorów i destruktorów, podczas tworzenia obiektu zaczyna się od podstaw w kończy na ostatnio dodanej funkcjonalności a w przypadku niszczenia zaczyna się od najnowszej funkcjonalności a kończy na bazowej. Kolejną istotną rzeczą jest to, że konfiguracja agenta powinna następować w metodzie `setup` zamiast w konstruktorze, ponieważ w przeciwieństwie do konstruktora, metoda setup wykonuje się w momencie gdy agent został już poprawnie stworzony i zarejestrowany w środowisku, co za tym idzie może korzystać funkcji przez nie udostępnionych.

Aby agenty mogły się poprawnie komunikować, muszą one korzystać ze wspólnego słownictwa i jeżyka. Aby umożliwić kompatybilność z rożnymi systemami agentowymi JADE zgodnie ze standardem FIPA pozwala na wykorzystanie kodeka wiadomości zapewniającego gramatykę języka oraz ontologii zapewniającej słownictwo jakim będą się porozumiewać agenty. Obie te dwie rzeczy należy zarejestrować poprzez zarządce treści agenta.

```
@Override
protected void setup() {
    super.setup();
    ContentManager cm = getContentManager();
    cm.registerLanguage(new SLCodec());
    cm.registerOntology(MobilityOntology.getInstance());
}
```

W przykładzie został wykorzystany kodek standardowego języka *SL* oraz ontologia *MobilityOntology* zapewniające słownictwo opisujące m.in. kontenery w środowisku. W przypadku gdy agent migruje, to powinien powtórzyć proces rejestracji kodeków i ontologii za każdym razem jak jest rejestrowany w nowym kontenerze (np. w metodzie `afterMove`).

Środowisko JADE jest zorientowane na zachowania. Oznacza to że wszystkie akcje podejmowane przez agenty powinny być zaimplementowane w postaci pojedynczych zachowań. Na początku możemy wymienić cztery podstawowe klasy zachowań:

- `Behaviour` – podstawowa klasa dla wszystkich zachowań,

- `SimpleBehaviour` – proste zachowanie nie zawierające żadnych pod zachowań,
- `OneShotBehaviour` – zachowanie wykonujące się jeden raz,
- `CyclicBehaviour` – zachowanie wykonujące się cyklicznie.

```
@Override
protected void setup() {
    super.setup();
    ContentManager cm = getContentManager();
    cm.registerLanguage(new SLCodec());
    cm.registerOntology(MobilityOntology.getInstance());
    this.addBehaviour(new RequestContainersListBehaviour(this));
}
```

Zachowania agenta są uruchamiane przez jego wewnętrzny mechanizm zarządzania zachowaniami. Aby dodać zachowanie do agenta należy wykorzystać metodą `addBehaviour`. W przypadku gdy jakieś zachowania powinny zostać dodane wraz z uruchomieniem agenta to najlepiej dodać je w metodzie `setup`.

```
public class RequestContainersListBehaviour extends OneShotBehaviour {

    protected final MigratingAgent myAgent;

    public RequestContainersListBehaviour(MigratingAgent agent) {
        super(agent);
        myAgent = agent;
    }

    @Override
    public void action() {
        ...
    }
}
```

W przypadku implementacji pojedynczego zachowania (`SimpleBehaviour`) i cyklicznego (`CyclicBehaviour`) wystarczy nadpisać metodę action definiująca co ma zostać zrealizowane przez agenta. Jeśli kod zachowania ma mieć dostęp do obiektu agenta należy pamiętać o przekazaniu go w konstruktorze. W przypadku implementowania zachowań cyklicznych należy unikać wszelkich konstrukcji typu `while(true){}`. Zachowania cykliczne będzie automatycznie co jakiś czas uruchamiane przez wewnętrznego zarządce agenta.

```
public class ReceiveContainersLisBehaviour extends SimpleBehaviour {

    private boolean done = false;

    protected final MigratingAgent myAgent;
```



```
    public ReceiveContainersLisBehaviour(MigratingAgent agent) {
        super(agent);
        myAgent = agent;
    }

    @Override
    public void onStart() {
        ...
    }

    @Override
    public void action() {
        ...
    }

    @Override
    public boolean done() {
        return done;
    }

}
```

W przypadku implementacji zachowań z własnym warunkiem stopu należy unikać konstrukcji `while(!done){}`. Należy wykorzystać klasę `SimpleBehaviour` i oprócz metody action należy zaimplementować metodę `done` informujące zarządce zadań czy zachowanie może być już usunięte z agenta.

Przykład migrującego agenta zawiera trzy zachowania. Pierwszej pojedyncze zachowanie (`RequestContainersListBehaviour`) jest uruchamiane wraz agentem i wysyła do agenta *AMS* (*Agent Management System*) żądanie podania listy wszystkich kontenerów składających się na platformę. Po wysłaniu żądania do agenta dodawane jest zachowanie `ReceiveContainersLisBehaviour` z własną funkcję stopu. Będzie ono działać tak długo aż agent AMS nie odeśle listy kontenerów. W momencie odebrania listy kontenerów do agenta zostaje dodane zachowanie `MigratingBehaviour` działające tak długo jak zostały jeszcze jakieś nieodwiedzone przez agenta kontenery.

Podczas komunikacji agenty korzystają ze słownictwa zdefiniowanego w ontologii. Na ontologię składają się trzy podstawowe elementy:

- Concept – reprezentuje pewien byt (np. książka),
- Predicate – reprezentuje jakąś informację o wartości true/false (np. książka kosztuje 33PLN),
- AgentAction – reprezentuje akcję jaką ma zrealizować agent (np. sprzedaj książkę).


```
@Override
public void action() {
    QueryPlatformLocationsAction query;
    query = new QueryPlatformLocationsAction();
    Action action = new Action(myAgent.getAMS(), query);

    String conversationId = UUID.randomUUID().toString();

    ACLMessage request = new ACLMessage(ACLMessage.REQUEST);
    request.setLanguage(new SLCodec().getName());
    request.setOntology(MobilityOntology.getInstance().getName());
    request.addReceiver(myAgent.getAMS());
    request.setConversationId(conversationId);

    try {
        myAgent.getContentManager().fillContent(request, action);
        myAgent.send(request);
        myAgent.addBehaviour(
          new ReceiveContainersLisBehaviour(myAgent, conversationId));
    } catch (Codec.CodecException | OntologyException ex) {
        log.log(Level.WARNING, ex.getMessage(), ex);
    }
}
```

W powyższym przykładzie są realizowane następująco:

- tworzony obiekt `QueryPlatformLocationsAction` implementujący interfejs `AgentAction`, mówiący że agent ma podać listę kontenerów na platformie,
- tworzony jest obiekt `Action` zawierający w sobie opis akcji (interfejs `AgentAction`) i agenta, który ma tę akcję zrealizować (obiekt `AID`),
- generowany jest id konwersacji pozwalający na filtrowanie wiadomości należących do konkretnej konwersacji,
- tworzona jest wiadomość typu żądanie,
- język wiadomości ustawiany jest na standardowy język *SL*,
- ontologia wiadomości jest ustawiana na ontologię opisującą środowisko,
- agent *AMS* ustawiany jest jako odbiorca wiadomości,
- ustawiane jest id konwersacji,
- za pomocą zarządcy treści agenta wiadomość jest wypełniania treścią,
- wysłanie wiadomości,
- dodanie nowego zachowania służącego odebraniu wiadomości.


```
 @Override
public void onStart() {
    super.onStart();
    mt = MessageTemplate.MatchConversationId(conversationId);
}

@Override
public void action() {
    ACLMessage msg = myAgent.receive(mt);
    if (msg != null) {
        done = true;
        try {
            ContentElement ce =
              myAgent.getContentManager().extractContent(msg);
            jade.util.leap.List items = ((Result) ce).getItems();
            List<Location> locations = new ArrayList<>();
            items.iterator().forEachRemaining(i -> {
                locations.add((Location) i);
            });
            locations.remove(myAgent.here());
            myAgent.setLocations(locations);
            myAgent.addBehaviour(new MigratingBehaviour(myAgent));
        } catch (Codec.CodecException | OntologyException ex) {
            log.log(Level.SEVERE, null, ex);
        }
    }
}
```

W powyższym przykładzie realizowane są następująco:

- ustawienie filtra wiadomości na akceptowanie tylko tych z konkretnym id konwersacji,
- nieblokująca metoda odbierania wiadomości,
- jeśli przyszła wiadomość można ustawić flagę oznaczającą zakończenie zachowania (`done`) na `true`,
- rozpakowanie treści wiadomości za pomocą zarządcy treści,
- wiadomość powinna być tupu `Result` implementującego interfejs `Predicate`, pobranie listy wyników,
- usunięcie z listy kontenera na którym znajduje się agent,
- dodanie zachowania migracji do agenta.

Zadania:

- zestawienie środowiska złożonego z przynajmniej czterech kontenerów (na jednej maszynie) i uruchomienie w nim migrującego agenta,
- modyfikacja kodu tak aby agent na sam koniec wracał do swojego pierwszego kontenera,
- modyfikacja kodu tak aby agent poruszał się w kółko po wszystkich kontenerach,
- modyfikacja zadana podana przez prowadzącego, może dotyczyć zmiany kolejności odwiedzania kontenerów lub akcji jaką ma agent wykonać w momencie przybycia do kontenera.

