Systemy Agentowe

Laboratorium 4

Celem zadania jest zapoznanie się mechanizmem definiowania, rejestrowania, wyszukiwania i wywoływania usług w środowisku wieloagentowym. Przy realizacji zadania należy posłużyć się przykładem (Alchemists) zaprezentowanym na wykładzie.

Zawartość przykładu

Zawartość projektu Alchemists:

- `pl.gda.pg.eti.kask.sa.alchemists.agents` – paczka zawierający agenty:
  - `Alchemist.java` – agent sprzedający mikstury,
  - `Herbalist.java` – agent sprzedający zioła,
  - `Mage.java` – agent kupujący produkty,
  - `BaseAgent.java` – bazowa klasa dla agentów w projekcie;
- `pl.gda.pg.eti.kask.sa.alchemists.behaviours` – zachowania agentów:
  - `ActionBehaviour.java` – wykonanie akcji żądanej przez innego agenta i odesłanie wyniku,
  - `FindServiceBehaviour.java` – wyszukanie agentów udostępniających określoną usługę,
  - `ReceiveResultBehaviour.java` – oczekiwanie na wynik akcji żądanej od innego agenta,
  - `RegisterServiceBehaviour.java` – zarejestrowanie usługi w katalogu,
  - `RequestActionBehaviour.java` – żądanie od innego agenta wykonania określonej akcji,
  - `WaitingBehaviour.java` – oczekiwanie na wiadomości nie związanej z żadną aktywną konwersacją,
  - `alchemist` – zachowania alchemika:
    - `AlchemistBehaviour.java` – implementacja `WaitingBehaviour`,
    - `SellPotionBehaviour.java` – implementacja `ActionBehaviour`, sprzedaż mikstury;
  - `herbalist` – zachowania zielarza:

        - `HerbalistBehaviour.java` – implementacja `WaitingBehaviour`,
        - `SellHerbBehaviour.java` – implementacja `ActionBehaviour`, sprzedaż zioła;
    - `mage` – zachowania maga:
        - `MageBehaviour.java` – oczekiwanie na zakończenie innego zachowania,
        - `ReceiveHerbBehaviour.java` – implementacja `ReceiveResultBehaviour`, odebranie zioła,
        - `ReceivePotionBehaviour.java` – implementacja `ReceiveResultBehaviour`, odebranie mikstury,
        - `RequestHerbBehaviour.java` – implementacja `RequestActionBehaviour`, żądanie zioła,
        - `RequestPotionBehaviour.java` – implementacja `RequestActionBehaviour`, żądanie mikstury;
- `pl.gda.pg.eti.kask.sa.alchemists.ontology` – ontologia:
    - `AlchemyOntology.java` – klasa definiująca ontologię,
    - `Herb.java` – koncept opisujący zioło,
    - `Potion.java` – koncept opisujący miksturę,
    - `SellHerb.java` – akcja sprzedaży zioła,
    - `SellPotion.java` – akcja sprzedaży mikstury.

Ontologia

Ontologia jest zbiorem słownictwa wykorzystywanego w komunikacji pomiędzy agentami. Najprostszym sposobem definiowania ontologii jest dziedziczenie po klasie `jade.concept.onto.BeanOntology` a następnie zarejestrowanie wszystkich jej elementów w konstruktorze.

```
@Log
public class AlchemyOntology extends BeanOntology {

    public static final String NAME = "alchemy-ontology";

    @Getter
    private static final AlchemyOntology instance
        = new AlchemyOntology(NAME);

    private AlchemyOntology(String name) {
        super(name);
        try {
            add(Herb.class);
            add(Potion.class);
```



```
            add(SellHerb.class);
            add(SellPotion.class);
        } catch (BeanOntologyException ex) {
            log.log(Level.SEVERE, null, ex);
        }
    }
}
```

Na ontologię składają się trzy elementy:

- koncept – opis pewnego bytu,
- predykat – opisuje cechę konceptu,
- akcja – opisuje akcję wykonywaną przez agenta.

Wykorzystywany przykład składa się z dwóch konceptów opisujących miksturę i zioło oraz dwóch akcji opisujących sprzedaż bytów opisanych przez te koncepty.

Elementy ontologii definiuje się poprzez definicję klas i implementacją odpowiednich interfejsów: `Concept`, `Predicate` i `AgentAction`.

Przykład konceptu:

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode
@ToString
public class Herb implements Concept {

    private String name;

}
```

Przykład akcji:

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode
@ToString
public class SellHerb implements AgentAction {

    private Herb herb;

}
```


Należy pamiętać że każdy z elementów ontologii musi mieć publiczny konstruktor bez parametrów oraz prywatne pola dostępne przez publiczne metody `get` i `set`.

Bazowe klasy

Bazową klasą dla wszystkich agentów w przykładzie jest `BaseAgent`, zawierający jedynie listę aktywnych konwersacji oraz rejestrację języka SL i odpowiedniej ontologii w metodzie `setup`.

```
public class BaseAgent extends Agent {

    @Getter
    protected final List<String> activeConversationIds
        = new ArrayList<>();

    public BaseAgent() {
    }

    @Override
    protected void setup() {
        super.setup();
        getContentManager().registerLanguage(new SLCodec());
        getContentManager().registerOntology(
                 AlchemyOntology.getInstance());
    }

}
```

Globalna lista aktywnych konwersacji ma na celu zapewnienie, że zachowania odbierające wszystkie wiadomości przychodzące do agenta nie odbierą tych będących częścią już wcześniej rozpoczętej konwersacji. Na wiadomości których identyfikator konwersacji znajduje się na liście czekają inne zachowania.

Klasa `WaitingBehaviour` jest abstrakcyjną klasą służącą odbieraniu wiadomości inicjujących konwersację.

```
@Override
public void action() {
    MessageTemplate mt = MessageTemplate.MatchAll();
    for (String id : myAgent.getActiveConversationIds()) {
        mt = MessageTemplate.and(
            mt, MessageTemplate.not(
                MessageTemplate.MatchConversationId(id)));
    }
    ACLMessage message = myAgent.receive(mt);
    if (message != null) {
        try {
            ContentElement ce = myAgent.getContentManager()
                    .extractContent(message);
```

```
            if (ce instanceof Action) {
                action((Action) ce,
                    message.getConversationId(), message.getSender());
            }
        } catch (Codec.CodecException | OntologyException ex) {
            log.log(Level.SEVERE, null, ex);
        }
    }
}

protected abstract void action(Action action, String conversationId,
    AID participant);
```

Powyżej przedstawiono metodę action zachowania, która kolejno:

- tworzy filtr przyjmujący wszystkie wiadomości,
- dodaje do filtru ignorowanie wszystkich aktywnych konwersacji,
- odbiera wiadomość z kolejki,
- jeśli przyszła jakaś wiadomość to jej zawartość jest rozpakowywana,
- jeśli zawartością wiadomości był obiekt typu `Action` to jest on wraz z identyfikatorem konwersacji i identyfikatorem nadawcy przekazywany do abstrakcyjnej metody `action`.

Abstrakcyjna metoda `action` jest nadpisywana w konkretnych zachowaniach przygotowanych już pod konkretnego agenta.

Klasa `RequestActionBehaviour` jest abstrakcyjną klasą służącą wysłania żądania wykonania jakiejś akcji do agenta.

```
public RequestActionBehaviour(E agent, AID participant, T action) {
    this.participant = participant;
    this.action = action;
    this.myAgent = agent;
}

@Override
public void action() {
    Action action = new Action(participant, this.action);
    String conversationId = UUID.randomUUID().toString();

    ACLMessage request = new ACLMessage(ACLMessage.REQUEST);
    request.setLanguage(new SLCodec().getName());
    request.setOntology(AlchemyOntology.getInstance().getName());
    request.addReceiver(participant);
    request.setConversationId(conversationId);

    try {
        myAgent.getContentManager().fillContent(request, action);
        myAgent.getActiveConversationIds().add(conversationId);
        myAgent.send(request);
        ReceiveResultBehaviour resultBehaviour =
```

```
                createResultBehaviour(conversationId);
        if (getParent() != null
                && getParent() instanceof SequentialBehaviour) {
            ((SequentialBehaviour)getParent()).addSubBehaviour(
                    resultBehaviour);
        } else {
            myAgent.addBehaviour(resultBehaviour);
        }
    } catch (Codec.CodecException | OntologyException ex) {
        log.log(Level.WARNING, ex.getMessage(), ex);
    }
}

protected abstract ReceiveResultBehaviour createResultBehaviour(
        String conversationId);
```

Powyżej przedstawiono metodę `action` zachowania, która kolejno:

- tworzy żądanie wykonania akcji zawierające agenta, który ma ją wykonać i opis akcji,
- generuje identyfikator konwersacji,
- tworzy wiadomość typu żądanie i ustawia wszystkie wymagane własności,
- wypełnia wiadomość i dodaje identyfikator konwersacji do globalnej list aktywnych konwersacji,
- wysyła wiadomość,
- za pomocą abstrakcyjnej metody `createResultBehaviour` tworzy zachowanie służące odebraniu wyniku żądanej akcji,
- zależnie od tego czy aktualne zachowanie wykonuje się jako element jakiegoś złożonego zachowania czy bezpośrednio z poziomu agenta nowe zachowanie jest dodawane albo do rodzica albo bezpośrednio do agenta.

Abstrakcyjna metoda `createResultBehaviour` jest nadpisywana w konkretnych zachowaniach przygotowanych już pod konkretnego agenta.

Klasa `ReceiveResultBehaviour` jest abstrakcyjną klasą służącą odebraniu wyniku żądanej akcji.

```
public ReceiveResultBehaviour(E agent, String conversationId) {
    super(agent);
    this.myAgent = agent;
    this.conversationId = conversationId;
}

@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);
            handleResult((Predicate) ce, msg.getSender());
        } catch (Codec.CodecException | OntologyException ex) {
            log.log(Level.SEVERE, null, ex);
        }
    }
}

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

protected abstract void handleResult(Predicate predicate,
        AID participant);
```

Powyżej przedstawiono metodę action zachowania, która kolejno:

- odbiera wiadomość należącą do tej konkretnej konwersacji,
- jeśli przyszła wiadomość to ustawia flagę zakończenia na `true`,
- wypakowuje wiadomość i przekazuje jej wynik do abstrakcyjnej metody `handleResult`.

Abstrakcyjna metoda `handleResult` jest nadpisywana w konkretnych zachowaniach przygotowanych już pod konkretnego agenta.

Klasa `ActionBehaviour` jest abstrakcyjną klasą służącą wykonaniu żądanej akcji przez agenta.

```
public ActionBehaviour(E agent, T action, String conversationId,
        AID participant) {
    super(agent);
    this.myAgent = agent;
    this.action = action;
    this.conversationId = conversationId;
    this.participant = participant;
}

@Override
public void action() {
    Predicate result = performAction();
    ACLMessage msg;
    if (result != null) {
        msg = new ACLMessage(ACLMessage.INFORM);
```

```
        } else {
            msg = new ACLMessage(ACLMessage.REFUSE);
        }
        msg.setLanguage(new SLCodec().getName());
        msg.setOntology(AlchemyOntology.getInstance().getName());
        msg.setConversationId(conversationId);
        msg.addReceiver(participant);
        try {
            if (result != null) {
                myAgent.getContentManager().fillContent(msg, result);
            }
            myAgent.send(msg);
        } catch (Codec.CodecException | OntologyException ex) {
            log.log(Level.SEVERE, null, ex);
        }
    }

    protected abstract Predicate performAction();
```

Powyżej przedstawiono metodę `action` zachowania, która kolejno:

- wywołuje abstrakcyjną metodę `performAction` zwracającą predykat zawierający wynik wykonanej akcji,
- jeśli nie ma wyniku akcji to tworzy wiadomość odmowy wykonania, w innym przypadku wiadomość informująca o wyniku,
- ustawia wszystkie wymagane własności wiadomości i wypełnia ją zawartością,
- wysyła wiadomość.

Abstrakcyjna metoda `performAction` jest nadpisywana w konkretnych zachowaniach przygotowanych już pod konkretnego agenta i pod konkretną akcję.

Usługi

Zarządzanie katalogiem usług na platformie JADE odbywa się za pośrednictwem agenta DF, który podobnie jak agent AMS jest uruchamiany na głównym kontenerze wraz ze startem platformy.

Klasa `RegisterServiceBehaviour` realizuje pojedyncze zachowanie rejestrujące usługę.

```
public RegisterServiceBehaviour(Agent agent, String serviceType) {
    super(agent);
    this.serviceType = serviceType;
}

@Override
public void action() {
    DFAgentDescription dfad = new DFAgentDescription();
```

```
    dfad.setName(myAgent.getAID());
    ServiceDescription sd = new ServiceDescription();
    sd.setType(serviceType);
    sd.setName(myAgent.getName() + "-" + serviceType);
    dfad.addServices(sd);
    try {
        DFService.register(myAgent, dfad);
    } catch (FIPAException ex) {
        log.log(Level.SEVERE, null, ex);
    }
}
```

Powyżej przedstawiono metodę action zachowania, która kolejno:

- tworzy opis agenta,
- tworzy opis usługi,
- dodaje opis usługi do opisu agenta,
- rejestruje opis agenta.

Klasa `FindServiceBehaviour` realizuje pojedyncze zachowanie wyszukujące usługę.

```
public FindServiceBehaviour(Agent agent, String serviceType) {
    super(agent);
    this.serviceType = serviceType;
}

@Override
public void action() {
    DFAgentDescription dfad = new DFAgentDescription();
    ServiceDescription sd = new ServiceDescription();
    sd.setType(serviceType);
    dfad.addServices(sd);
    try {
        DFAgentDescription[] services = DFService.search(
                myAgent, dfad);
        onResult(services);
    } catch (FIPAException ex) {
        log.log(Level.SEVERE, null, ex);
    }
}

protected abstract void onResult(DFAgentDescription[] services);
```

Powyżej przedstawiono metodę action zachowania, która kolejno:

- tworzy pusty opis agenta,
- tworzy opis usługi,
- dodaje opis usługi do opisu agenta,
- wyszukuje agentów udostępniających usługę,

- wywołuje abstrakcyjną metodę `onResult`.

Abstrakcyjna metoda `onResult` jest nadpisywana w konkretnych zachowaniach przygotowanych już pod konkretnego agenta i pod konkretną usługę.

Konkretne zachowania

Implementacje zarówno klas agentów jak i klas zachowań dla alchemika i zielarza (`Alchemist/Hebalist`) są zrobione analogicznie i różnią się jedynie kontekstem (mikstury, zioła). Każdy z nich dziedziczy bo bazowej klasie `BaseAgent` i zawiera globalną listę rzeczy, które może sprzedawać.

```
public class Alchemist extends BaseAgent {

    @Getter
    private final List<Potion> potions = new ArrayList<>();

    public Alchemist() {
    }

    @Override
    protected void setup() {
        super.setup();
        potions.add(new Potion("Shrouding Potion"));
        potions.add(new Potion("Heroic Potion"));
        addBehaviour(new RegisterServiceBehaviour(this, "alchemist"));
        addBehaviour(new AlchemistBehaviour(this));
    }

}
```

Agent po uruchomieniu wypełnia swoją listę, rejestruje swoją usługę sprzedaży oraz uruchamia zachowanie odbierające wiadomości inicjujące konwersacje.

Zachowania `AlchemistBehaviour` i `HerbalistBehaviour` dziedziczą po `WaitingBehaviour` i implementują jedynie metodę action odpowiedzialną za wykonanie żądanej akcji.

```
public class AlchemistBehaviour extends WaitingBehaviour<Alchemist>{

    public AlchemistBehaviour(Alchemist agent) {
        super(agent);
    }

    @Override
    protected void action(Action action, String conversationId,
            AID participant) {
        if (action.getAction() instanceof SellPotion) {
            myAgent.addBehaviour(new SellPotionBehaviour(myAgent,
                    (SellPotion) action.getAction(),
                     conversationId, participant));
```

```
        }
    }

}
```

Zarówno alchemik jak i zielarz obsługują jedynie akcję sprzedaży (`SellPotion/SellHerb`). W przypadku gdy wiadomość inicjująca zawiera obsługiwane żądanie do agenta dodawane jest zachowania realizacji akcji (`SellPotionBehaviour/SellHerbBehaviour`).

Klasy `SellPotionBehaviour` i `SellHerbBehaviour` dziedziczą po klasie bazowej `ActionBehaviour` i implementują jedynie metodę `performAction`.

```
public class SellHerbBehaviour
        extends ActionBehaviour<SellHerb, Herbalist> {

    public SellHerbBehaviour(Herbalist agent, SellHerb action,
            String conversationId, AID participant) {
        super(agent, action, conversationId, participant);
    }

    @Override
    protected Predicate performAction() {
        if (myAgent.getHerbs().contains(action.getHerb())) {
            return new Result(action, action.getHerb());
        } else {
            return null;
        }
    }

}
```

Ostatnim agentem jest mag (`Mage`) szukający odpowiednich ziół i mikstur.

```
public class Mage extends BaseAgent {

    public Mage() {
    }

    @Override
    protected void setup() {
        super.setup();
        SequentialBehaviour behaviour = new SequentialBehaviour(this);

        behaviour.addSubBehaviour(...);
        behaviour.addSubBehaviour(...);

        addBehaviour(behaviour);
        addBehaviour(new MageBehaviour(behaviour, this));
    }

}
```

Podczas uruchomienia agent tworzy dwa zachowania. Pierwszym jest zachowanie sekwencyjne, które będzie miało na celu znalezienie i kupienie wszystkich wymaganych elementów. Drugim jest zachowanie sprawdzające czy poprzednie zostało już wykonane.

Na początku sekwencyjne zachowanie składa się z dwóch pod zachowań szukających usługi alchemika i zielarza.

```
behaviour.addSubBehaviour(
        new FindServiceBehaviour(this, "alchemist") {
    @Override
    protected void onResult(DFAgentDescription[] services) {
        if (services != null && services.length > 0) {
            AID alchemist = services[0].getName();
            SellPotion action
                    = new SellPotion(new Potion("Heroic Potion"));
            RequestPotionBehaviour request
                 = new RequestPotionBehaviour(Mage.this, alchemist,
                         action);
            ((SequentialBehaviour) getParent())
                    .addSubBehaviour(request);

        }
    }
});
```

Wyszukiwanie zostało dodane jako klasa anonimowa z nadpisaną metodą `onResult`, która jeśli zostanie znaleziona usługa dodaje zachowanie wysłania żądania wykonania akcji sprzedaży.

Zadania

Zadania do wykonania:

- uruchomienie przykładu,
- modyfikacja kodu alchemika i zielarza tak aby lista mikstur/ziół które posiada były przekazywane jako argumenty startowe agenta a nie na sztywno ustawione w kodzie,
- modyfikacja kodu maga tak aby mógł szukać listy różnych ziół i mikstur podawanych jako argumenty startowe agenta,
- stworzenie nowego sprzedawcy (odpowiednie elementy ontologii, zachowanie sprzedaży, zachowanie odbierania wiadomości, agent) podanego przez prowadzącego,
- dodanie zachowań pozwalających kupowanie magowi od nowego sprzedawcy.

Uwagi

Kod może być pomocny w realizacji dalszych zadań laboratoryjnych.