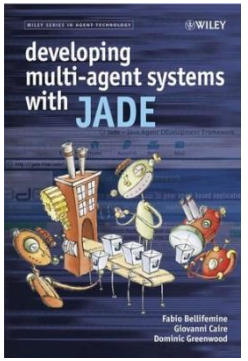
JADE
Java Agent Development Framework

MiASI2, TWO, 2018-2019

Materiały

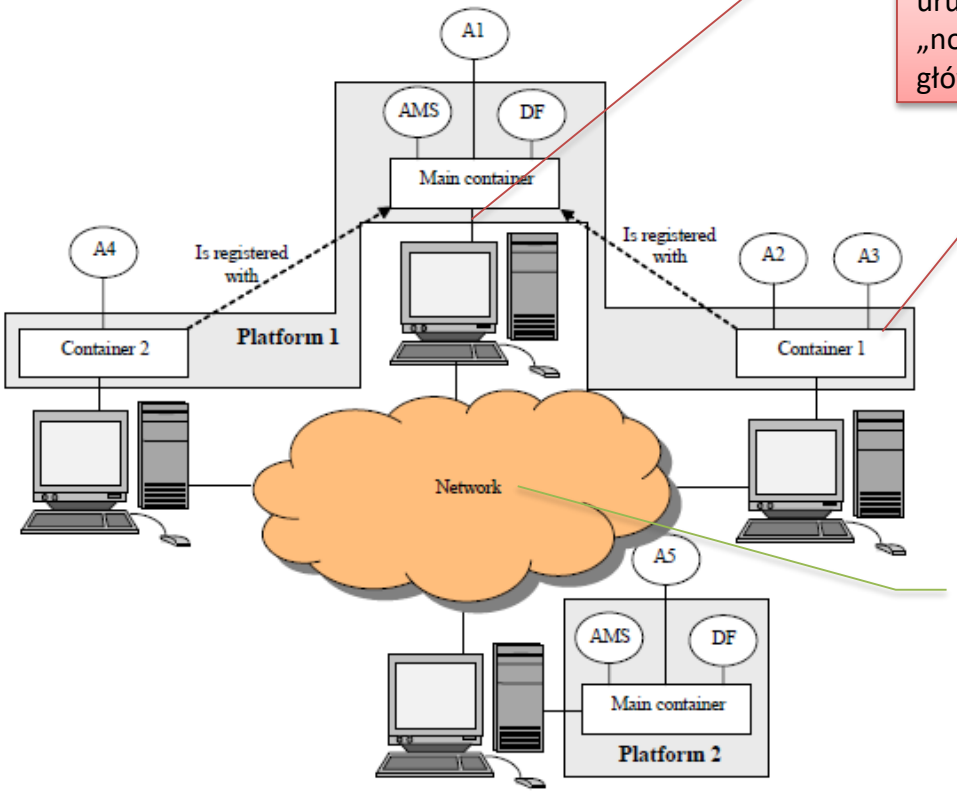
- Strona projektu JADE
  http://jade.tilab.com/
  (Telecom Italia, University of Parma, Motorola Labs)
- Dokumentacja
  - Programming Tutorial
  - Programmer's Guide
  - Administrator's Guide
- Publikacje
  - F.L. Bellifemine, G. Caire, D. Greenwood: Developing Multi-Agent Systems with JADE. Wiley, 2007.



JADE

- JADE – środowisko typu *middleware* do budowy systemów wieloagentowych
- Elementy składowe
  - Biblioteka klas (Java) do budowy systemów wieloagentowych (platformy stacjonarne i mobilne – Android)
  - Środowisko uruchomieniowe i komunikacyjne dla agentów
  - Zestaw narzędzi (GUI) do zarządzania agentami i ich monitorowania
- Dostępny na licencji *open source*

Architektura JADE



Główny kontener – główny (pierwszy uruchomiony) kontener w platformie. Pozostałe „normalne” kontenery rejestrują się w kontenerze głównym

Kontener – działająca instancja środowiska uruchomieniowego JADE

Komunikacja między agentami w różnych kontenerach i na różnych platformach

```
java -cp <classpath> jade.Boot –gui
java -cp <classpath> jade.Boot -container -host avalon.tilab.com –agents john:myPackage.MyAgentClass
```

Agenci AMS i DF

- Funkcjonują w głównym kontenerze i startują automatycznie wraz z kontenerem
- AMS (Agent Management System)
  - Zarządza identyfikatorami/nazwami agentów zapewniając ich unikalność (usługa *White Pages*)
  - Pozwala na tworzenie i usuwanie agentów na wszystkich kontenerach w ramach platformy
- DF (Directory Facilitator)
  - Usługa *Yellow Pages*
  - „Ogłoszenia” i ich wyszukiwanie

Przykład – handel książkami

Agenci sprzedają i kupują książki w imieniu swoich użytkowników.

Każdy agent kupujący ma za zadanie kupić jedną książkę (tytuł przekazany jako parametr w linii poleceń). Cyklicznie pyta on agentów sprzedających o dostępność książki i prosi o oferty sprzedaży (cena). Po otrzymaniu ofert agent kupujący wybiera najlepszą (najtańszą) i wysyła zamówienie do agenta sprzedającego. Po zrealizowaniu zamówienia agent kupujący kończy działanie.

Każdy agent sprzedający ma proste GUI pozwalające na dodawanie nowych książek do sprzedaży (tytuły i ceny). Agent ten oczekuje na pytania od agenta kupującego. W przypadku zapytania ofertowego, agent sprzedający sprawdza, czy ma daną książkę w swoim katalogu i jeśli tak, przesyła cenę. W przeciwnym razie informuje o braku dostępności. Po otrzymaniu zamówienia, agent sprzedający sprawdza, czy książka jest nadal w katalogu i realizuje zamówienie.

Implementacja agenta

```java
package test;

import jade.core.Agent;

public class SimpleAgent1 extends Agent {
    @Override
    protected void setup() {
        System.out.printf("Hello, world! from %s\n", getAID().getName());
    }

    @Override
    protected void takeDown() {
        System.out.printf("Taking down agent %s...\n", getAID().getName());
    }

}
```

Metoda uruchamiana gdy agent rozpoczyna działanie

Metoda uruchamiana gdy agent kończy działanie. Wymuszenie zakończenia działania za pomocą metody **`doDelete()`**

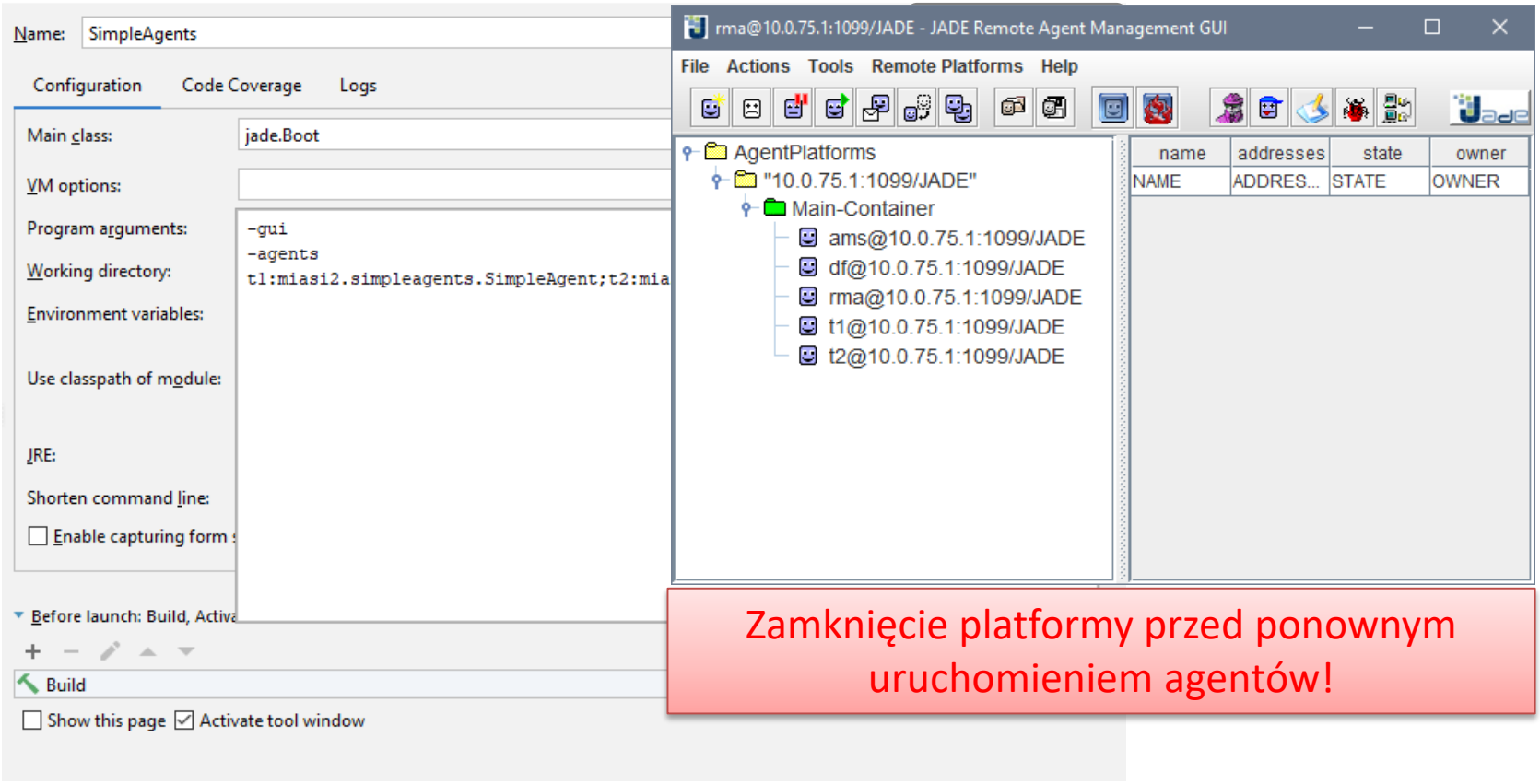
Identyfikator agenta

- Każdy agent jest identyfikowany za pomoca unikalnego (globalnie) identyfikatora (`jade.core.AID`)
- Schemat identyfikatora: `<nickname>@<platform>`
- Tworzenie identyfikatora na podstawie „lokalnej” nazwy agenta (→ doklejenie nazwy aktualnej platformy „domowej”)

```
String nickname = “Peter”;
AID id = new AID(nickname, AID.ISLOCALNAME);
```

Uruchamianie agentów

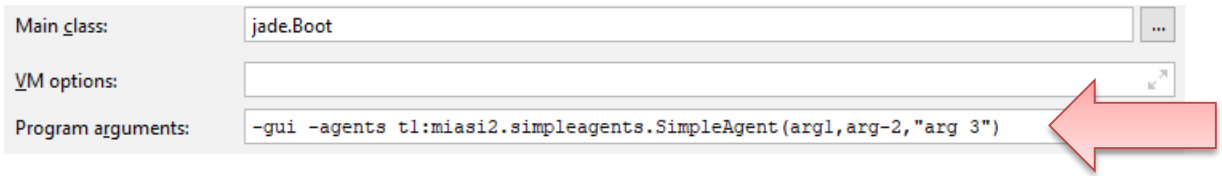
```
java -cp <classpath> jade.Boot -gui
-agents t1:miasi2.simpleagents.SimpleAgent;t2:miasi2.simpleagents.SimpleAgent
```



Przekazywanie argumentów do agenta

Dostęp do parametrów przekazanych z linii poleceń

```
public class SimpleAgent3 extends Agent {
    @Override
    protected void setup() {
        System.out.printf("Hello, world! from %s\n", getAID().getName());

        Object[] args = getArguments();
        if (args != null) {
            for (int i = 0; i < args.length; i++)
                System.out.printf("%d. %s\n", i + 1, args[i].toString());
        }
    }

    @Override
    protected void takeDown() {
        System.out.printf("Taking down agent %s...\n", getAID().getName());
    }

}
```



Bez spacji między parametrami!

Przekazywanie argumentów do agenta

```
public class BookBuyerAgent extends Agent {
    // The title of the book to buy
    private String targetBookTitle;
    // The list of known seller agents
    private AID[] sellerAgents = { new AID("seller1", AID.ISLOCALNAME),
            new AID("seller2", AID.ISLOCALNAME) };

    // 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);
        } else {
            // Make the agent terminate immediately
            System.out.println("No book title specified");
            doDelete();
        }
    }

    // Put agent clean-up operations here
    protected void takeDown() {
        // Printout a dismissal message
        System.out.println("Buyer-agent " + getAID().getName()
                + " terminating.");
    }
}
```

Zachowania agenta

- Zachowanie (`Behaviour`) implementuje „algorytm” pozwalający na realizację pewnego zadania przez agenta
- Każda klasa implementująca zachowanie musi implementować dwie metody:
  - `action()` – właściwa akcja (algorytm) związany z zachowaniem
  - `done()` – informacja, czy akcja się zakończyła, czy też nie
- JADE udostępnia klasy implementujące bardziej złożone schematy zachowań (możliwa zmiana „obowiązkowych metod)

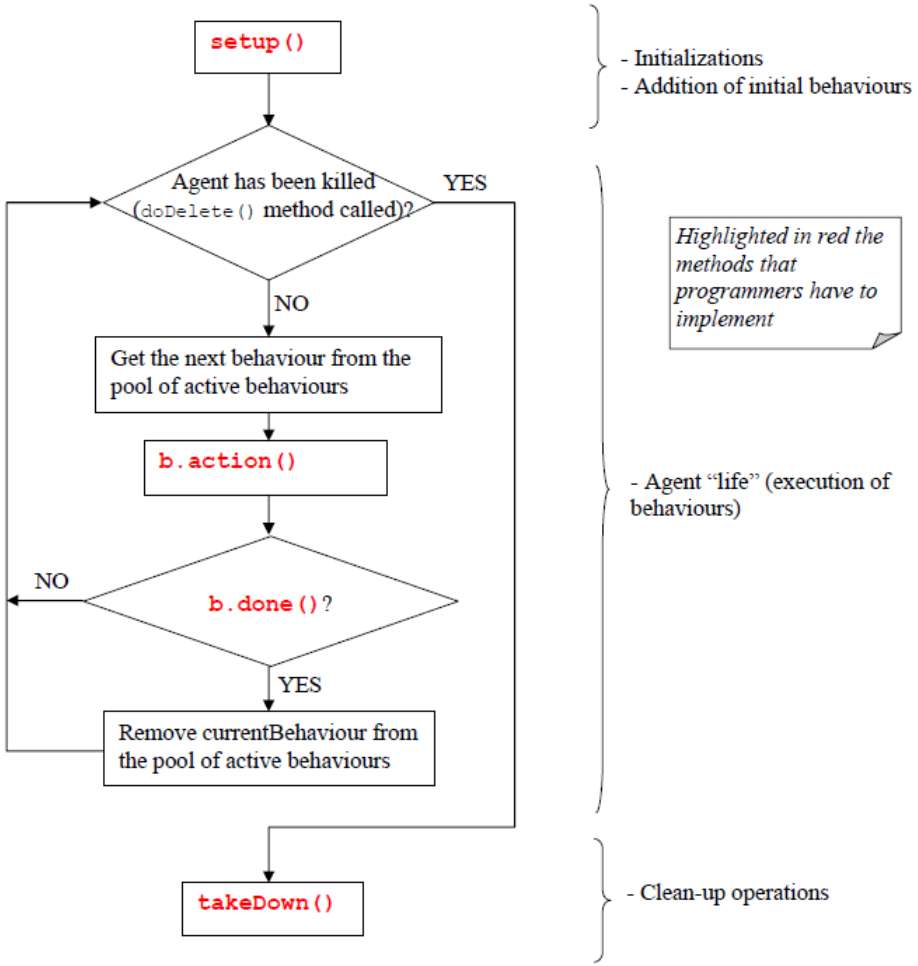
Realizacja zachowań

- Jeden agent może realizować równolegle wiele zachowań
- Ale każdy agent posiada swój **jeden** wątek i wszystkie zachowania są realizowane w tym wątku
- Konieczna kooperacja między zachowaniami → nieblokująca implementacja metody `action()`

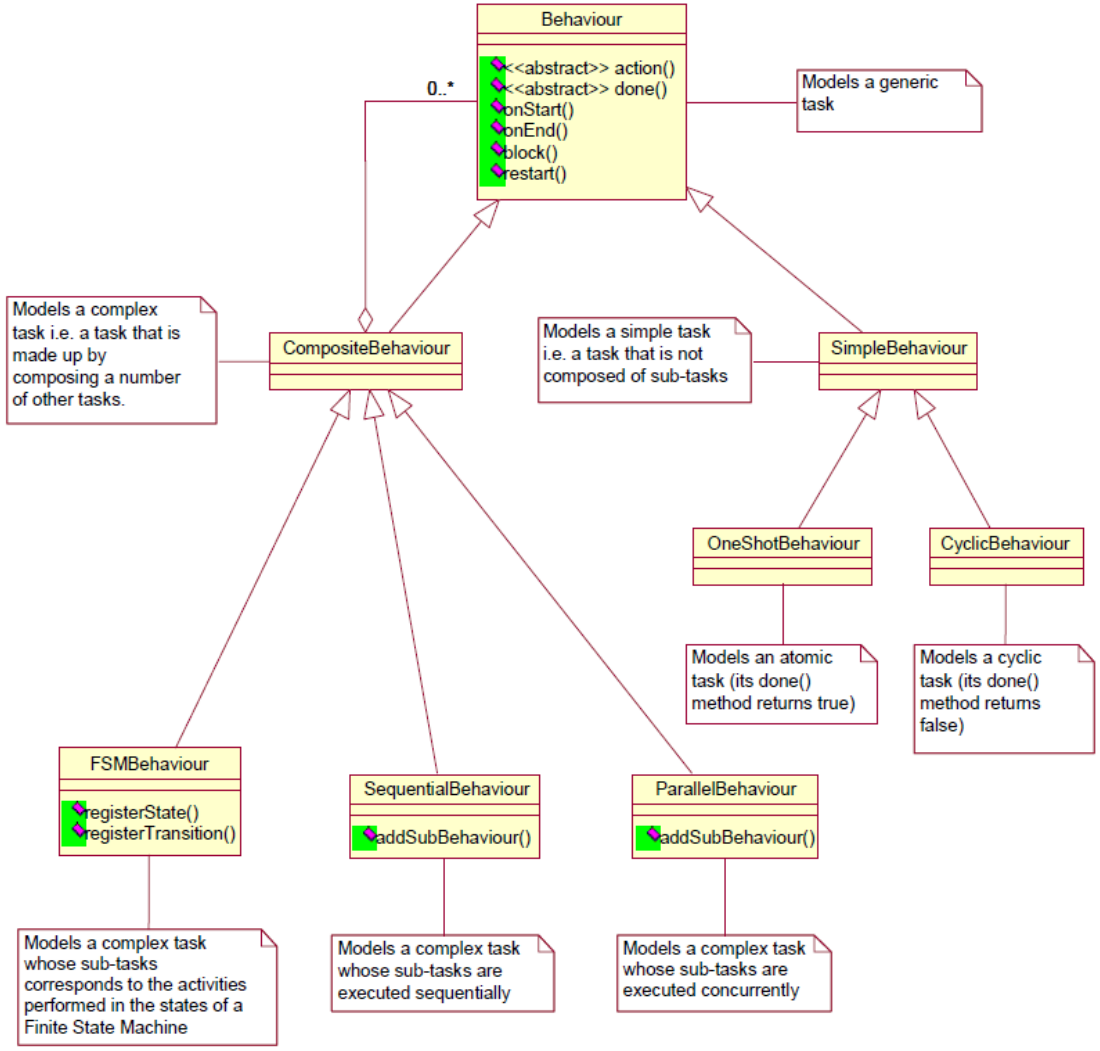
```
public class OverbearingBehaviour extends Behaviour {
    public void action() {
        while (true) {
            // do something
        }
    }

    public boolean done() {
        return true;
    }
}
```

- Zalety jednego wątku
  - Brak synchronizacji przy dostępie do zasobów
  - Możliwość migracji agentów pomiędzy platformami (łatwy zapis stanu agenta)
  - Możliwość uruchamiania agentów w „lekkich” środowiskach (CDC)

Cykl życia agenta



Dostępne rodzaje zachowań



Przykłady zachowań

- `OneShotBehaviour` – zachowanie, które wykona się tylko raz

```
public class MyOneShotBehaviour extends OneShotBehaviour {

    @Override
    public void action() {
        // Perform operation X
    }

    // No need for done() (always returns true)

}
```

- `CyclicBehaviour` – zachowanie, które będzie wykonać się cyklicznie (bez końca)

```
public class MyCyclicBehaviour extends CyclicBehaviour {

    @Override
    public void action() {
        // Perform operation X
    }

    // no need for done() (always returns false)

}
```

Przykłady zachowań

- `WakerBehaviour` – zachowanie, które wykona się tylko raz po określonym (w ms) czasie

```
public class MyWakerAgent extends Agent {
    protected void setup() {
        System.out.println("Adding waker behaviour");
        addBehaviour(new WakerBehaviour(this, 10000) {
            protected void handleElapsedTimeout() {
                // perform operation X
            }
        });
    }
}
```

- `TickerBehaviour` – zachowanie, które będzie wykonywać się cyklicznie w określonych (w ms) odstępach czasu

```
public class MyTickerAgent extends Agent {
    protected void setup() {
        addBehaviour(new TickerBehaviour(this, 10000) {
            protected void onTick() {
                // perform operation Y
            }
        });
    }
}
```

Uwaga na zmienione nazwy metod (dedykowane metody zamiast `action()`)

Przykłady zachowań

- Zachowania „generyczne” (np. wielostanowe)
- Inne zachowania
  - `SequentialBehaviour`
  - `ParallelBehaviour`
  - `FSMBehaviour`
  - …

```
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;
    }
}
```

Zachowanie kupującego

```
public class BookBuyerAgent extends Agent {
    // The title of the book to buy
    private String targetBookTitle;
    // The list of known seller agents
    private AID[] sellerAgents = { new AID("seller1", AID.ISLOCALNAME),
            new AID("seller2", AID.ISLOCALNAME) };

    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();
        }
    }
```

Próba zakupienia książki (`RequestPerformer`) co 60 sekund

Zachowanie sprzedającego

```
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));
            }
        });
    }
}
```

Cykliczne odpowiedzi na zapytania o dostępność książki (`OfferRequestServer`)

Cykliczne odpowiedzi na żądania zakupu książki (`PurchaseOrderServer`)

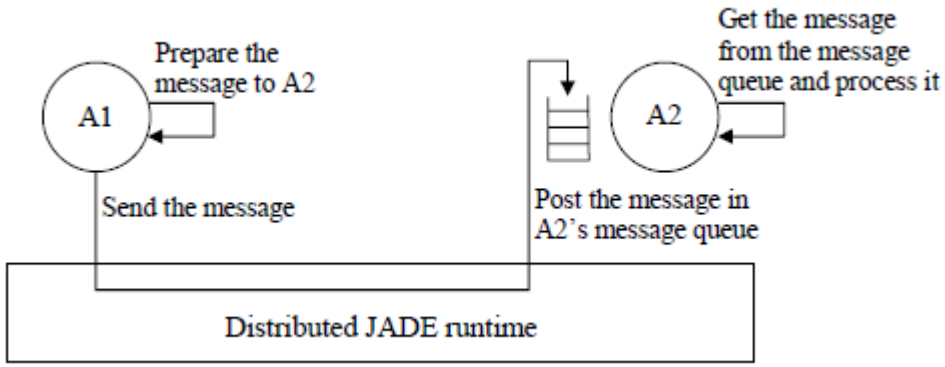
JADE oraz GUI

- JADE i GUI działają w różnych wątkach – zapewnienie odpowiedniej separowalności
- GUI zmienia stan agenta poprzez dodanie odpowiedniego zachowania (`OneShotBehaviour`)
- Agent zmienia stan GUI poprzez stworzenie obiektu `Runnable` i uruchomienie go za pomocą `SwingUtilities`

```
public void catalogueUpdated() {
    // TODO Auto-generated method stub
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            DefaultListModel<String> model = new DefaultListModel<String>();
            for (String title : _myAgent.getCatalogue().keySet()) {
                Integer price = _myAgent.getCatalogue().get(title);
                model.addElement(String.format("%s [%s]", title, price));
            }
            _lstBooks.setModel(model);
        }
    });
}
```

Komunikacja między agentami

- Asynchroniczna wymiana komunikatów między agentami
- Każdy agent ma swoją „skrzynkę pocztową” i jest informowany, kiedy pojawią się w niej nowe wiadomości
- Agent sam decyduje kiedy pobrać i przetworzyć wiadomość



Elementy wiadomości

- `sender` – nadawca wiadomości
- `receivers` – odbiory wiadomości
- `performative` – klasa wiadomości (`REQUEST`, `INFORM`, `CFP`, `ACCEPT_PROPOSAL`, `REJECT_PROPOSAL`, …)
- `content` – treść wiadomości (uzupełnienie pola `performative`)
- `lanaguage`, `ontology` – język i ontologia użyte do sformułowania treści wiadomości (opcjonalnie)
- `conversation-id`, `reply-with`, `in-reply-to` – pola służące do zarządzania konwersacjami

```
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);
```

Konwersacja między kupującym a sprzedającym

Odbieranie wiadomości

```
ACLMessage msg = receive();
if (msg != null) {
  // Process the message
}
```

```
public void action() {
ACLMessage msg = myAgent.receive();
if (msg != null) {
  // Message received. Process it
  ...
}
else {
  block();
  }
}
```

- Metoda `receive()` zwraca pierwszą dostępną wiadomość lub `null`, jeśli kolejka jest pusta
- W przypadku pustej kolejki niepotrzebne obciążenie procesora (ciągłe wywoływanie metody `receive()`)
- Zablokowanie zachowania (metoda `block()`) w sytuacji, gdy kolejka wiadomości jest pusta.
- Zablokowane zachowania nie są wykonywane, pojawienie się nowej wiadomości w kolejce powoduje ich odblokowanie

Odpowiadanie na wiadomości

```
private class OfferRequestsServer extends CyclicBehaviour {
    public void action() {
        ACLMessage msg = myAgent.receive();
        if (msg != null) {
            // CFP 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);
        }
        else {
            block();
        }
    }
}  // End of inner class OfferRequestsServer
```

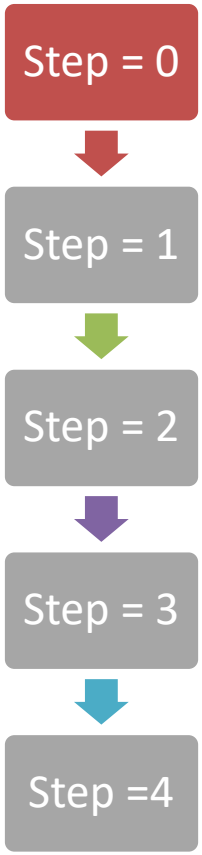
Problem w przypadku dwóch działających równolegle zachowań (`OfferRequestServer` i `PurchaseOrderServer`), które mogą sobie „podbierać” komunikaty

Szablony wiadomości

- Możliwość definiowania szablonów (`MessageTemplate`) określających, które filtrują wiadomości z kolejki
- Szablon podawany jako parametr metody receive()

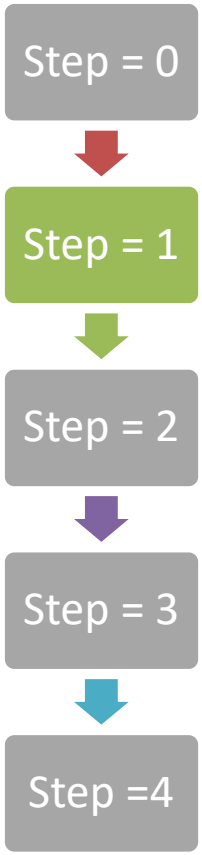
```
public void action() {
  MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.CFP);
  ACLMessage msg = myAgent.receive(mt);
  if (msg != null) {
    // CFP Message received. Process it
    ...
  }
  else {
  block();
  }
}
```

Złożone konwersacje




```
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;
```

Złożone konwersacje

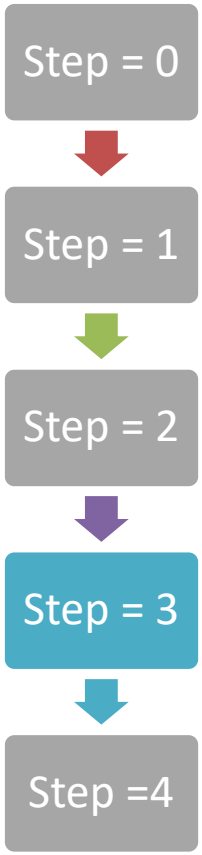



```
case 1:
    // Receive all proposals/refusals from seller agents
    ACLMessage reply = myAgent.receive(mt);
    if (reply != null) {
        // Reply received
        if (reply.getPerformative() == ACLMessage.PROPOSE) {
            // This is an offer
            int price = Integer.parseInt(reply.getContent());
            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;
```

Złożone konwersacje

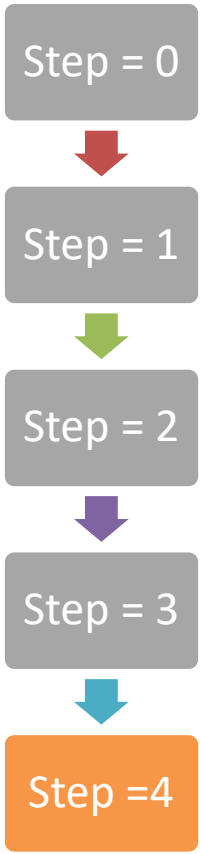

```
case 2:
    // Send the purchase order to the seller that provided the best offer
    ACLMessage order = new ACLMessage(ACLMessage.ACCEPT_PROPOSAL);
    order.addReceiver(bestSeller);
    order.setContent(targetBookTitle);
    order.setConversationId("book-trade");
    order.setReplyWith("order"+System.currentTimeMillis());
    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;
```

Złożone konwersacje




```
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 from agent "+reply.getSender().getName());
            System.out.println("Price = "+bestPrice);
            myAgent.doDelete();
        }
        else {
            System.out.println("Attempt failed: requested book already sold.");
        }

        step = 4;
    }
    else {
        block();
    }
    break;
}
```

Złożone konwersacje

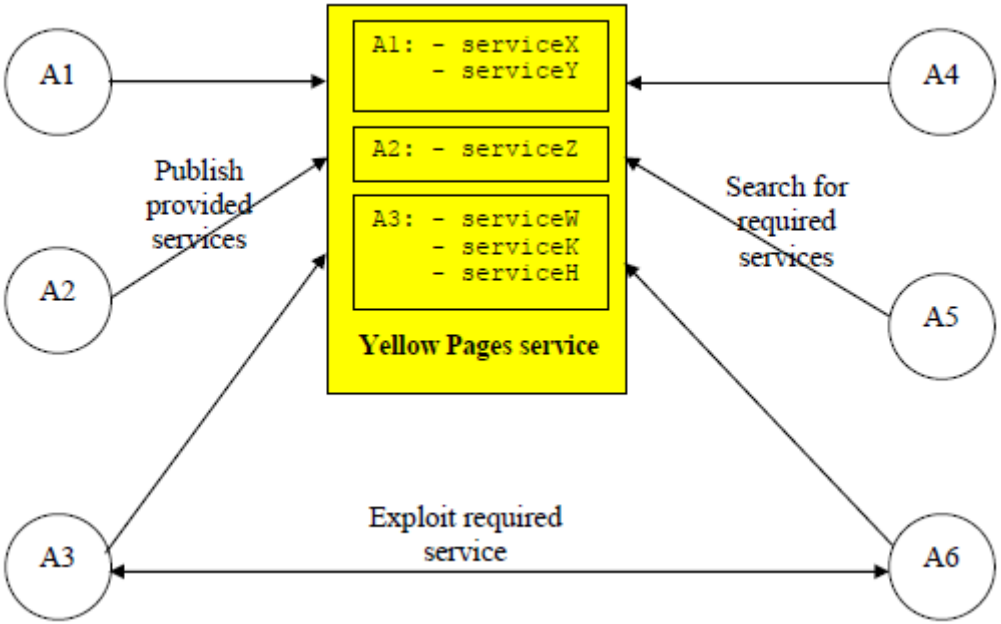



```
public boolean done() {
    if (step == 2 && bestSeller == null) {
        System.out.println("Attempt failed: "+targetBookTitle+" not available for sale");
    }
    return ((step == 2 && bestSeller == null) || step == 4);
}
```

Zakończenie pracy w sytuacji, gdy nie ma sprzedającego (`step == 2`), albo po próbie kupna książki (`step ==4`)

Yellow Pages / Agent DF

- Mechanizm *Yellow Pages* pozwala agentom na ogłaszanie udostępnianych usług oraz na przeszukiwanie tych ogłoszeń



- Klasa `DFService` do obsługi *Yellow Pages* (ukrywa agenta DF)

Rejestracja i wyrejestrowanie usługi

```
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();
  }
...
}
```

Rejestracja usługi w *Yellow Pages* podczas startu agenta

Wyrejestrowanie usługi z *Yellow Pages* po zakończeniu działania agenta

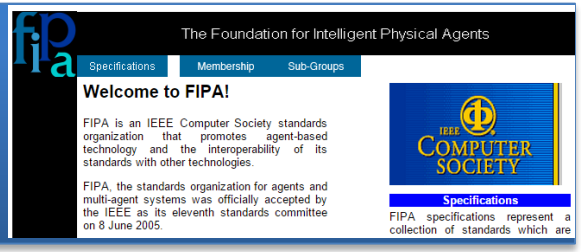
```
protected void takeDown() {
  // Deregister from the yellow pages
  try {
  DFService.deregister(this);
  }
  catch (FIPAException fe) {
  fe.printStackTrace();
  }
  // Printout a dismissal message
  System.out.println("Seller-agent "+getAID().getName()+" terminating.");
}
```

Wyszukiwanie usług

```
// Add a TickerBehaviour that schedules a request to seller agents every minute
addBehaviour(new TickerBehaviour(this, 60000) {
    protected void onTick() {
        System.out.println("Trying to buy "+targetBookTitle);
        // 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);
            System.out.println("Found the following seller agents:");
            sellerAgents = new AID[result.length];
            for (int i = 0; i < result.length; ++i) {
                sellerAgents[i] = result[i].getName();
                System.out.println(sellerAgents[i].getName());
            }
        }
        catch (FIPAException fe) {
            fe.printStackTrace();
        }

        // Perform the request
        myAgent.addBehaviour(new RequestPerformer());
    }
} );
```

Możliwość otrzymywania powiadomień od agenta DF o nowych usługach (*push* zamiast *pull*)

Protokoły FIPA



- Możliwość wykorzystania klas realizujących podstawowe protokoły wprowadzone przez FIPA
  - Contract-Net
  - Propose
  - Subscribe
- Obsługa wybranych zdarzeń w ramach protokołu bez konieczności obsługi całego procesu
- Pary zachowań …`Initiator` i …`Responder`

FIPA Contract Net: Initiator

```
private class BuyBookBehaviour extends ContractNetInitiator {

    public BuyBookBehaviour(Agent a, ACLMessage cfp) {
        super(a, cfp);
        // TODO Auto-generated constructor stub
    }

    protected void handleAllResponses(Vector responses, Vector acceptances) {
        int bestPrice = -1;
        AID bestSeller = null;
        ACLMessage bestReply = null;

        for (Object r : responses) {
            ACLMessage response = (ACLMessage) r;
            if (response.getPerformative() == ACLMessage.PROPOSE) {
                ACLMessage reply = response.createReply();
                reply.setPerformative(ACLMessage.REJECT_PROPOSAL);
                acceptances.add(reply);
                int price = Integer.parseInt(response.getContent());
                if (bestReply == null || price < bestPrice) {
                    bestPrice = price;
                    bestReply = reply;
                    bestSeller = response.getSender();
                }
            }
        }
        if (bestReply != null) {
            bestReply.setPerformative(ACLMessage.ACCEPT_PROPOSAL);
            System.out.printf("Seller %s offered the best price [%d] for book '%s'\n",
                    bestSeller.getLocalName(), bestPrice, _requestedBookTitle);
        }
    }
```

FIPA Contract Net: Initiator

```
private class BuyBookBehaviour extends ContractNetInitiator {

    public BuyBookBehaviour(Agent a, ACLMessage cfp) {
        super(a, cfp);
        // TODO Auto-generated constructor stub
    }

    protected void handleAllResponses(Vector responses, Vector acceptances) {


    }

    protected void handleInform(ACLMessage inform) {
        System.out.printf("Book '%s' has been purchased from seller %s\n",
                _requestedBookTitle, inform.getSender().getLocalName());
        doDelete();
    }

    protected void handleFailure(ACLMessage failure) {
        System.out.printf("Seller %s has finally failed to sell book '%s'\n",
                failure.getSender().getLocalName(), _requestedBookTitle);
    }
};
```

FIPA Contract Net: Initiator

```
protected void onTick() {
    ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
    cfp.setProtocol(FIPANames.InteractionProtocol.FIPA_CONTRACT_NET);
    cfp.setConversationId("book-trading");
    cfp.setContent(_requestedBookTitle);

    DFAgentDescription template = new DFAgentDescription();
    ServiceDescription sd = new ServiceDescription();
    sd.setType("book-selling");
    template.addServices(sd);
    try {
        DFAgentDescription[] result = DFService.search(myAgent, template);
        for (int i = 0; i < result.length; i++)
            cfp.addReceiver(result[i].getName());
    }
    catch (FIPAException ex) {
        ex.printStackTrace(System.err);
    }
    myAgent.addBehaviour(new BuyBookBehaviour(myAgent, cfp));
}
```

FIPA Contract Net: Responder

```
private class SellBooksBehaviour extends ContractNetResponder {

    public SellBooksBehaviour(Agent a, MessageTemplate mt) {
        super(a, mt);
        // TODO Auto-generated constructor stub
    }

    protected ACLMessage handleCfp(ACLMessage cfp)
            throws RefuseException, FailureException, NotUnderstoodException {
        String title = cfp.getContent();
        if (_catalogue.containsKey(title)) {
            Integer price = _catalogue.get(title);
            System.out.printf("Seller %s is ready to sell book '%s' for %d\n",
                    myAgent.getAID().getLocalName(), title, price.intValue());
            ACLMessage propose = cfp.createReply();
            propose.setPerformative(ACLMessage.PROPOSE);
            propose.setContent(price.toString());
            return propose;
        } else
            throw new RefuseException("no book available");
    }

    protected ACLMessage handleAcceptProposal(ACLMessage cfp, ACLMessage propose, ACLMessage accept)
            throws FailureException {
        String title = cfp.getContent();
        if (_catalogue.containsKey(title)) {
            ACLMessage inform = accept.createReply();
            inform.setPerformative(ACLMessage.INFORM);
            _catalogue.remove(title);
            System.out.printf("Seller %s has sold book '%s'\n",
                    myAgent.getAID().getLocalName(), title);
            updateGui();
            return inform;
        } else
            throw new FailureException("book has been sold");
    }

}
```

FIPA Contract Net: Responder

```
MessageTemplate template = MessageTemplate.and(
        MessageTemplate.and(
                MessageTemplate.MatchProtocol(FIPANames.InteractionProtocol.FIPA_CONTRACT_NET),
                MessageTemplate.MatchPerformative(ACLMessage.CFP)),
        MessageTemplate.MatchConversationId("book-trading"));

addBehaviour(new SellBooksBehaviour(this, template));
```

DF: subskrypcja zamiast *pull*

- Agent może „zapisać się” na powiadomienia wysyłane przez DF (możliwość określenia kryteriów zainteresowań)
- DF automatycznie powiadamia zainteresowanych agentów o zmianach (pojawienie się lub zamkniecie usługi)
- Obsługa mechanizmu subskrypcji za pomocą klasy `SubscriptionInitiator` (dedykowane zachowanie)

Subscription Initiator

```
private class SubscribeToDFBehaviour extends SubscriptionInitiator {
    public SubscribeToDFBehaviour(Agent a, DFAgentDescription template) {
        super(a, DFService.createSubscriptionMessage(a, a.getDefaultDF(), template, new SearchConstraints()));
    }

    protected void handleInform(ACLMessage inform) {
        System.out.printf("Buyer agent %s: received update from DF\n", getAID().getName());
        try {
            DFAgentDescription[] results = DFService.decodeNotification(inform.getContent());
            for (int i = 0; i < results.length; i++) {
                DFAgentDescription descr = results[i];
                Iterator<ServiceDescription> it = descr.getAllServices();
                if (it.hasNext()) {
                    System.out.printf("   adding a new seller agent %s\n", descr.getName());
                    _sellers.add(descr.getName());
                } else if (_sellers.contains(descr.getName())) {
                    System.out.printf("   removing an existing seller agent %s\n", descr.getName());
                    _sellers.remove(descr.getName());
                } else
                    System.out.printf("   problmatic (unknown) seller agent %s\n", descr.getName());
            }
        }
        catch (FIPAException e) {
            e.printStackTrace();
        }
    }
}
```

Zdefiniowanie subskrypcji i obsługa powiadomień

```
@Override
protected void takeDown() {
    System.out.printf("Buyer-agent %s is terminating...\n", getAID().getName());
    _subscriptionBehavior.cancel(getDefaultDF(), true);
}
```

Anulowanie subskrypcji po zakończeniu działania agenta

Zadanie na dziś

Proszę tak zmodyfikować przykładowy program, aby

1. agent kupujący (`BookBuyerAgent`) pozwalał na zakup wielu książek i kończył działanie po zakupie ich wszystkich
2. program wykorzystywał protokół Contract-Net oraz odpowiadał na automatyczne powiadomienia DF-a