

# Jade Java Agent DEvelopment framework

Loïc Pélissier  
Régis Lhoste

## Plan

- **Présentation générale**
- Les agents JADE
- La plateforme
- Concrètement
- Les principaux intérêts
- Implémentation dans les détails

## Présentation générale

- Plateforme de développement et exécution Multi-Agents
- JAVA
- Licence LGPL V2
- Architecture distribuée
- TILab S.p.A.(anciennement CSELT)  
Turin Italie
- Compatible avec le standard FIPA

## FIPA (Foundation for Intelligent Physical Agent)

- Formé en 1996 afin de construire des standards pour les agents hétérogènes, en interaction et les SMA
- Design de spécifications afin de faciliter l'interopérabilité entre les différents SMA développés par différentes sociétés et organisations
- Relations fortes avec d'autres standard et organisations comme OMG (Object Management Group)
- [www.fipa.org](http://www.fipa.org)

## Différentes plateformes FIPA

- Zeus
  - British Telecom
  - Java
  - FIPA 97-98
- FIPA – OS
  - Emorphia (Nortel Networks)
  - Java
  - FIPA 2000 (?)
- ...

## Plan

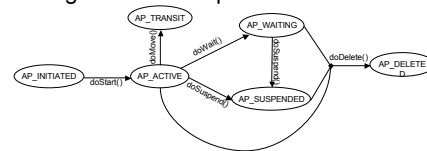
- Présentation générale
- **Les agents JADE**
- La plateforme
- Concrètement
- Les principaux intérêts
- Implémentation dans les détails

## Un agent selon JADE

- Conforme au standard FIPA 2002 (Jade 3)
- Cycle de vie
- Possède un ou plusieurs Comportements (Behaviours) qui définissent ses actions
- Communique et interagit avec les autres agents grâce à des Messages (ACLMessage)
- Rend des Services

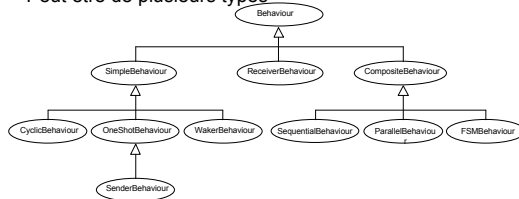
## Cycle de vie

- Un agent possède toujours un état (AP\_ACTIVE, AP\_DELETED...)
- Cycle de vie géré par cet état
- Changement d'état possible



## Un Comportement (Behaviour)

- Définit une action d'un agent
- Ordonnancement et exécution de tâches multiples
- Peut être de plusieurs types



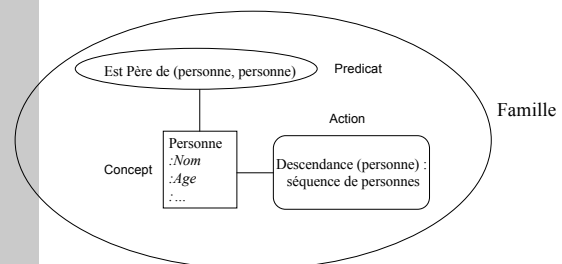
## Un Message

- Communication entre agents
- Conforme au standard FIPA ACL
- Les messages ACL présentent les caractéristiques suivantes :
  - Typage (INFORM, QUERY...)
  - Transport possible d'objets
  - Sémantique pour les données (Ontologies)

## Ontologies

- Représentation structurée de la connaissance.
  - Concepts (Représentation abstraite d'objets)
  - Prédicats (Condition binaire sur des concepts)
  - Actions (Traitements proposés par des agents sur des concepts)
- Partager la connaissance en fixant le domaine du discours.
- Construction graphique avec Protégé
  - lien vers Jade via plug-in (BeanGenerator)

## Ontologies exemple



## Un Service

- Action enregistrée et dispensée par la plateforme
- Comportements d'un ou plusieurs agents répondant à une demande.
- Notion de pages jaunes
- Proche de la notion de « WebServices »

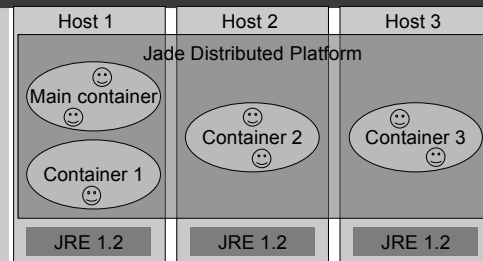
## Plan

- Présentation générale
- Les Agents JADE
- **La plateforme**
- Concrètement
- Les principaux intérêts
- Implémentation dans les détails

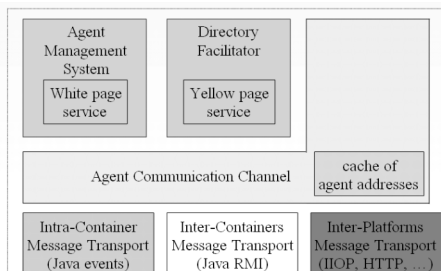
## Présentation de la plateforme

- Une plateforme de départ (avec ou sans GUI)
- 3 agents de base (Conforme FIPA)
  - Agent Management System
  - Directory Facilitator
  - Agent Communication Channel
- Quelques outils
- API

## Containers (Environnement d'exécution pour les agents)



## Une plateforme Jade



## Agent Management System (AMS)

- Gestion du cycle de vie des agents
- Maintient une liste de tous les agents qui résident sur la plate-forme (White pages)
- Contrôle l'accès ainsi que l'utilisation du canal de communication des agents (ACC)

## Directory Facilitator

- Enregistre les descriptions des agents ainsi que les services qu'ils offrent
- Les agents peuvent enregistrer leurs services auprès d'un DF ou demander à DF de découvrir les services offerts par d'autres agents (Yellow Pages)

## Agent Communication Channel

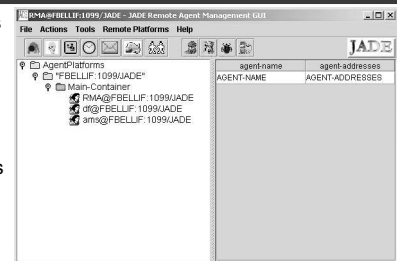
- Gère les communications entre les agents
  - Intra plateformes, Intra containers : Java Events
  - Intra plateformes, Inter containers : RMI
  - Inter Plateformes : IIOP Corba
- Messages ACL (FIPA)

## Outils

- Jade GUI
- DF Agent GUI
- Dummy Agent
- Sniffer Agent
- Introspector Agent

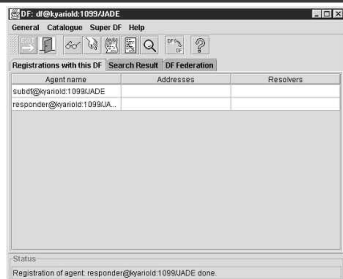
## Jade GUI

- Contrôler les agents
  - Créer
  - Tuer
  - Suspendre
  - ...
- Démarrer les autres outils



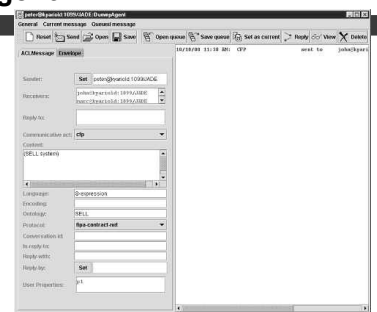
## DF Agent GUI

- Inspecter les Yellow Pages (services enregistrés)



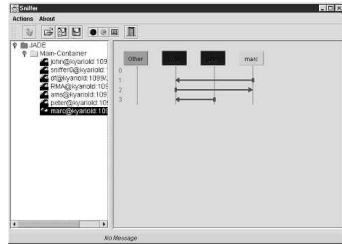
## Dummy Agent

- Système de tests
- Envoyer et recevoir des messages ACL



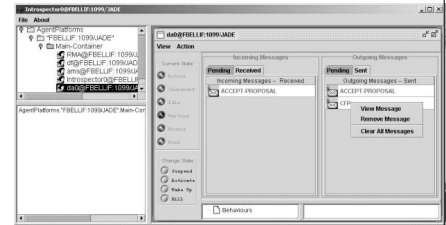
## Sniffer Agent

- Surveillance des échanges de messages dans une plateforme



## Introspector Agent

- Surveiller état (cycle de vie) d'un agent
- Ainsi que les messages reçus ou émis par cet agent



## API

- Java
- Surcharge de classes de base
- Communication transparente
- Compatibilité FIPA pour la communication avec des agents non Jade
- Compatibilité avec Jess et Protege

## Plan

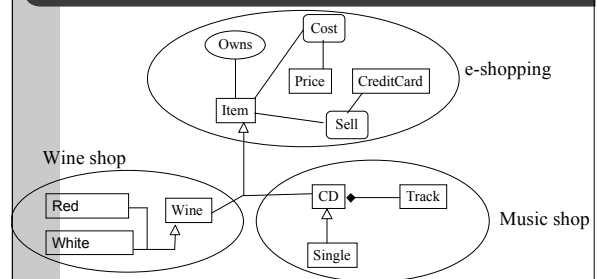
- Présentation générale
- Les agents JADE
- La plateforme
- Concrètement**
- Les principaux intérêts
- Implémentation dans les détails

## Exemple

Un application de e-commerce de musique

- Création des ontologies
- Création des comportements
- Création et enregistrement des agents
- Exécution

## Création des ontologies



## Création des comportements

- Deux agents distincts (Vendeur et Acheteur) impliquent deux ensembles de comportements
- Deux types de comportements sont utilisés :
  - Cyclic : Attente d'un message
  - OneShot : Exécution d'une action

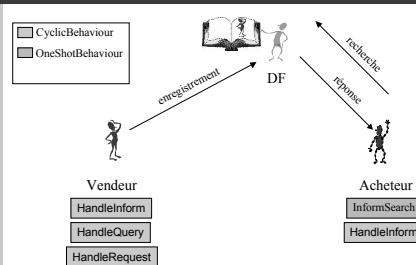
## Comportements du vendeur

| Action effectuée                           | Nom           | Type    |
|--------------------------------------------|---------------|---------|
| Attente d'une requête de disponibilité     | HandleInform  | Cyclic  |
| Informe l'acheteur sur la disponibilité    | InformOwns    | OneShot |
| Attente d'une requête de prix              | HandleQuery   | Cyclic  |
| Envoi du prix                              | InformCost    | OneShot |
| Attente d'une demande de transaction       | HandleRequest | Cyclic  |
| Notification du résultat de la transaction | InformDone    | OneShot |

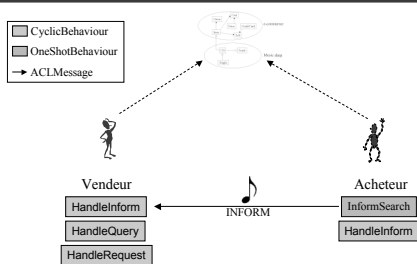
## Comportements de l'acheteur

| Action effectuée      | Nom          | Type    |
|-----------------------|--------------|---------|
| Trouver un vendeur    | InformSearch | OneShot |
| Attente d'une réponse | HandleInform | Cyclic  |
| Demander un prix      | QueryPrice   | OneShot |
| Effectuer la vente    | RequestSale  | OneShot |

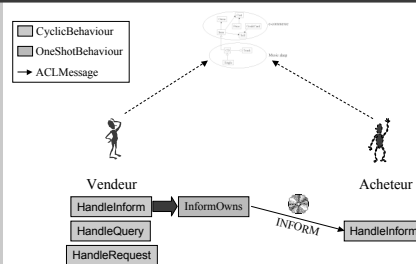
## Création et enregistrement des agents



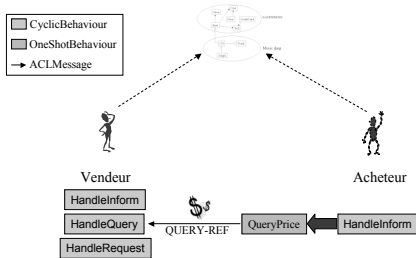
## Déroulement de la transaction



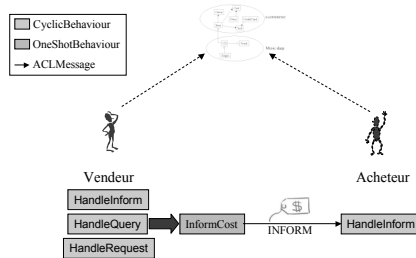
## Déroulement de la transaction



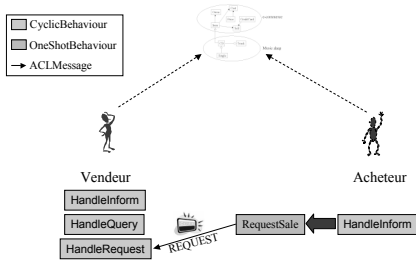
## Déroulement de la transaction



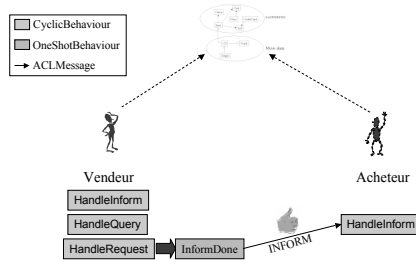
## Déroulement de la transaction



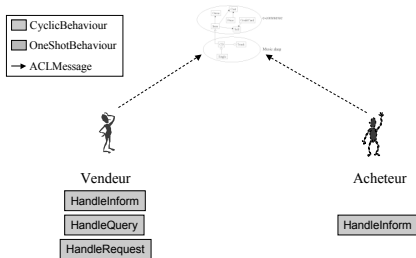
## Déroulement de la transaction



## Déroulement de la transaction



## Déroulement de la transaction



## Plan

- Présentation générale
- Les agents JADE
- La plateforme
- Concrètement
- **Les principaux intérêts**
- Implémentation dans les détails

## Les principaux intérêts

- Compatible FIPA
- Communauté de plus en plus importante
- Exécution distribuée
- Exécution concurrente des agents
- Communication transparente par message (ACL)
- Notion de services
- LEAP
- Relativement facile à utiliser

## Perspectives

- AgentCity MAIA
- ELIN
  - Architecture de filtrage
- Robotique Mobile
  - Communication entre robots
  - LEAP

## Plan

- Présentation générale
- Les agents JADE
- La plateforme
- Concrètement
- Les principaux intérêts
- ***Implémentation dans les détails***

## Développer un système à agents Jade

## Installer Jade

- Installer Java (jdk 1.2 ou supérieure)
  - Attention au « path » et au « classpath »
- Installer le package Jade
  - Copier les classes
  - Mettre à jour le « classpath » :
    - \jade
    - \jade\lib
    - \jade\lib\jop.jar
    - \jade\lib\jade.jar
    - \jade\lib\jadeTools.jar
- Installer vos agents comme des classes java classiques
  - Attention au « classpath »

## Exécuter Jade

- Lancer Jade avec la ligne de commandes :
  - java jade.Boot
- Lancer Jade et la GUI
  - java jade.Boot -gui
- Lancer un agent au démarrage
  - java jade.Boot -gui [nom de l'agent]:[classe de l'agent]
- Lancer un agent avec des paramètres
  - java jade.Boot -gui [nom de l'agent]:[classe de l'agent]([Paramètres])

## Créer un agent

- Etendre la classe `jade.core.Agent`
  - `public class monAgent extends Agent`
- Traiter les paramètres de démarrage
  - `Object [].getArguments()`
- Dans la méthode `setup()` (obligatoire)
  - Enregistrer les langages de contenu
  - Enregistrer les Ontologies
  - Enregistrer les Services auprès du DF
  - Démarrer les Comportements (behaviors)

## Créer un comportement

- Créer (étendre la classe « comportement »)  
`public class myBehaviour extends SimpleBehaviour`
- Créer le constructeur avec la super classe  
`public myBehaviour(Agent agent) {  
    super(agent);  
}`
- Il existe différents types de comportement à étendre (simple, oneshot, cyclic, ...)
- Créer la méthode « action » (obligatoire) qui correspond à l'exécution du comportement  
`public void action() {<code du comportement>}`

## Ajouter et démarrer un comportement

- Dans la méthode `Setup()` de l'agent  
Ajouter le comportement  
`addBehaviour(new myBehaviour(this));`
- Un comportement ajouté est démarré automatiquement

## Créer et enregistrer un service

- Créer une description du DF Agent  
`DFAgentDescription dfd = new DFAgentDescription();  
dfd.setName(this.getAID());`
- Créer la description du service  
`ServiceDescription sd = new ServiceDescription();  
sd.setType(<nom_type_service>);  
sd.setName(<nom_service>);`
- Enregistrer le service auprès du DF Agent  
`dfd.addServices(sd);  
try {DFService.register(this, dfd);}  
catch (FIPAException e) {<code d'erreur>}`

## Rechercher des agents proposant un service

- Créer une description du DF Agent  
`DFAgentDescription dfd = new DFAgentDescription();`
- Retrouver les agents dans un tableau  
`DFAgentDescription[] result = DFService.search(this, dfd);`
- Parcourir le tableau pour retrouver les agents  
`for (int i=0; i<result.length; i++) {  
    Iterator iter = result[i].getAllServices();  
    while (iter.hasNext()) {  
        ServiceDescription sd = (ServiceDescription)iter.next();  
        <traitement sur les services>}}  
}`
- On récupère toutes les descriptions de service

## Envoyer un message simple

- Créer l'agent qui envoie ce message
- Créer une instance de la classe `ACLMessage` avec un template (Inform, Query,...)
  - `ACLMessage msg = new ACLMessage(ACLMessage.INFORM)`
- Remplir l'ensemble des agents receveurs dans l'instance `ACLMessage`.
  - `msg.addReceiver(new AID(< nom du receveur, AID.ISLOCALNAME>)`
- Remplir le texte du contenu du message
  - `msg.setContent(< Salut ça va ? >);`
- Envoyer le message au receveur avec
  - `send(ACLMessage m);`

## Recevoir un message simple

- Créer l'agent receveur
- Créer le comportement (behaviour) de réception de message.
  - Déclarer le ou les Templates (Inform, Query,...) de message a recevoir

```
private static final MessageTemplate mt =
MessageTemplate.MatchPerformative(ACLMessage
e.INFORM);
```
  - Dans la méthode action()
    - Créer une instance du message et le recevoir

```
ACLMessage msg = myAgent.receive(mt);
```
    - La reception peut être bloquante, dans ce cas, utiliser `blockingReceive(mt)`
- Ajouter le behaviour a l'agent (dans `setup()`)
  - `addBehaviour(new ReceiverBehaviour(this));`

## Créer une ontologie de description de message (1/3)

- Créer l'objet Java représentant le contenu du message.
  - Créer l'ontologie de description représentant le message son nom et son vocabulaire (exemple avec la description d'un objet « Personne »)
    - ```
public class PersonOntology extends Ontology {
    public static final String ONTOLOGY_NAME = "Person-ontology";
    public static final String PERSON = "Person";
    public static final String PERSON_NAME = "name";
    public static final String PERSON_AGE = "age";
}
```
- Le vocabulaire correspond aux champs de l'objet "personne"

## Créer une ontologie de description de message (2/3)

- Une instance

```
private static Ontology theInstance = new PersonOntology();
public static Ontology getInstance() {
    return theInstance;
}
```
- Le constructeur fait office de "traducteur"

```
private PersonOntology() {
    super(ONTOLOGY_NAME, BasicOntology.getInstance());
    try {
        add(new PredicateSchema(PERSON), Person.class);
        PredicateSchema ps = (PredicateSchema) getSchema(PERSON);
        ps.add(PERSON_NAME, (PrimitiveSchema)
            getSchema(BasicOntology.STRING));
        ps.add(PERSON_AGE, (PrimitiveSchema)
            getSchema(BasicOntology.INTEGER));
        catch (OntologyException oe) {
            oe.printStackTrace();
        }
    }
}
```

## Créer une ontologie de description de message (3/3)

```
import jade.content.onto.*;
import jade.content.schema.*;
public class PersonOntology extends Ontology {

    public static final String ONTOLOGY_NAME = "Person-ontology";
    public static final String PERSON = "Person";
    public static final String PERSON_NAME = "name";
    public static final String PERSON_AGE = "age";

    private static Ontology theInstance = new PersonOntology();
    public static Ontology getInstance() {
        return theInstance;
    }

    private PersonOntology() {
        super(ONTOLOGY_NAME, BasicOntology.getInstance());
        try {
            add(new PredicateSchema(PERSON), Person.class);
            PredicateSchema ps = (PredicateSchema) getSchema(PERSON);
            ps.add(PERSON_NAME, (PrimitiveSchema) getSchema(BasicOntology.STRING));
            ps.add(PERSON_AGE, (PrimitiveSchema) getSchema(BasicOntology.INTEGER));
        }
        catch (OntologyException oe) {
            oe.printStackTrace();
        }
    }
}
```

## Envoyer un message avec une ontologie

- Créer l'ontologie de message
- Créer l'agent envoyeur
- Déclarer le content manager, le codec et l'ontologie

```
private ContentManager manager = (ContentManager) getContentManager();
private Codec codec = new SLCodec();
private Ontology ontology = PersonOntology.getInstance();
```
- Envoyer le message en ajoutant l'ontologie

```
ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
msg.addReceiver(new AID("~/<nom_receveur>", AID.ISLOCALNAME));
msg.setLanguage(codec.getName());
msg.setOntology(ontology.getOntology());
manager.fillContent(msg, <ontologie>;
send(msg);
```

## Recevoir un message avec une ontologie (1/2)

- Créer le comportement de reception

```
public class ReceiverBehaviour extends CyclicBehaviour {
```
- Déclarer CM, et le template du message

```
private ContentManager manager = myAgent.getContentManager();
private final static MessageTemplate mt =
MessageTemplate.MatchPerformative(ACLMessage.INFORM);
```
- Constructeur

```
public ReceiverBehaviour(Agent agent) { super(agent);}
```
- Action

```
public void action() {
    ACLMessage msg = myAgent.receive(mt);
    if (msg != null) {
        try {
            ContentElement ce = manager.extractContent(msg);
            if (ce instanceof MyOntology) {
                MyOntology myOntologie = (MyOntology) ce;
            }
        }
    }
}
```

## Recevoir un message avec une ontologie (2/2)

- Créer l'agent receveur

```
public class ReceiverAgent extends Agent {
```
- Declarer les CM, codec et Ontology

```
private ContentManager manager = (ContentManager) getContentManager();
private Codec codec = new SLCodec();
private Ontology ontology = MyOntology.getInstance();
```
- Ajouter le comportement

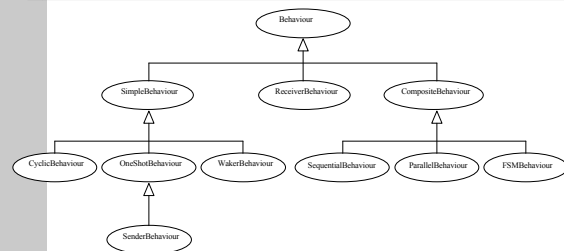
```
protected void setup() {
    manager.registerLanguage(codec);
    manager.registerOntology(ontology);
    addBehaviour(new ReceiverBehaviour(this));
}
```

## Retour sur les Behaviours et les Protocoles

## Behaviours

- Une super classe
  - Behaviour
- De nombreuses spécialisations
  - cf. graphe d'héritage ci-après

## Graphe d'héritage



## Retour sur Behaviour

- Agent :
  - void setup()
  - void takeDown()
  - ACLMessage receive([MessageTemplate mt])
  - ACLMessage blockingReceive([MessageTemplate mt[, long Timeout]])
  - void addBehaviour(Behaviour b)

## Fonctionnement des Behaviours

- Constructeur :
  - Behaviour([Agent a])
- Méthodes de gestion du cycle de vie :
  - void onStart()
  - int onEnd()
  - void action()
  - boolean done()

## Un peu plus loin

- Méthodes intéressantes :
  - void block([long millisecondes])
  - void reset()
  - Behaviour root()
- Attributs:
  - Agent myAgent
  - CompositeBehaviour parent

## Problème des CompositeBehaviour

- Pas d'échange entre les différents Behaviours composant le CompositeBehaviour
- Solution :
  - DataStore

## DataStore

- DataStore = HashMap
  - Partageable par plusieurs Behaviours
- Dans Behaviour :
  - setDataStore(DataStore ds)
  - DataStore getDataStore()

## Fonctionnement du DataStore

- Méthodes :
  - Object get(Object Key)
  - Object put(Object Key, Object Value)
  - Object remove(Object Key)
  - boolean containsKey(Object Key)
  - boolean isEmpty()
  - int size()
  - etc...

## Problème du DataStore

- Pas de typage
  - méthode get retourne un Object
  - nécessite un contrôle fort sur les données

## FSMBehaviour

- Finite State Machine Behaviour
  - Automate dont les nœuds sont des behaviours
- Enchaîner séquentiellement des Behaviours dans un ordre fixé ou non par des transitions

## Fonctionnement

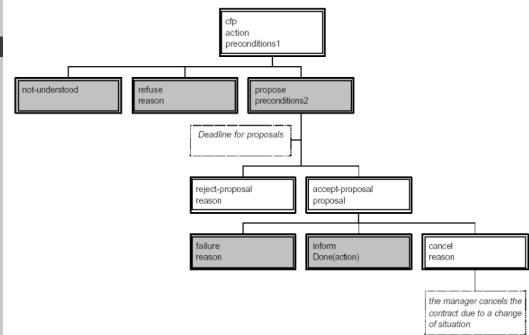
- Méthodes supplémentaires
  - Gestion des états de l'automate :
    - registerFirstState(Behaviour b, String name)
    - registerLastState(Behaviour b, String name)
    - registerState(Behaviour b, String name)
  - Gestion des transitions :
    - registerDefaultTransition(String from, String to)
    - registerTransition(String from, String to, int event)

## Subtilités

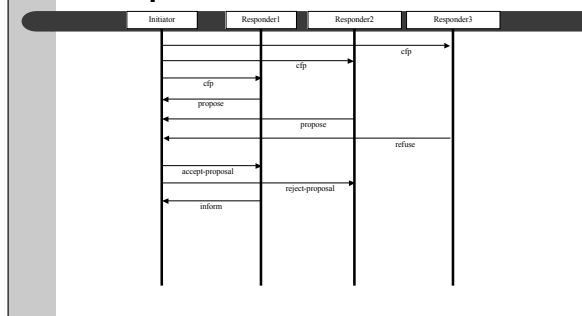
- Partager les données entre les Behaviours fils
  - Créer un DataStore dans le FSMBehaviour
  - Le partager par tous les états
- Déterminer les valeurs des transitions
  - ACLMessage performative = int

## Protocoles

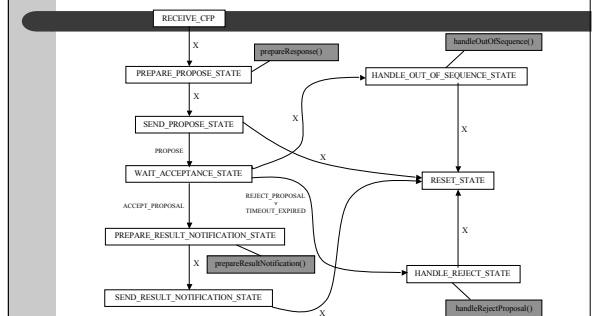
- ContractNet :
  - Implémenté par deux FSMBehaviour :
    - ContractNetInitiator
    - ContractNetResponder



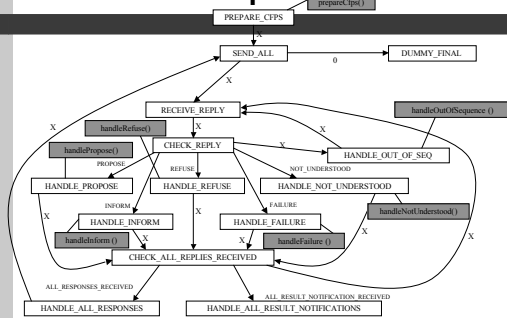
## Exemple



## ContractNetInitiator



## ContractNetResponder



## A noter

- Notion de « Conversation »
- Slot protocol à fixer dans les messages
- Doit prendre en compte :
  - Reply-By (Deadline/Timeout)
  - Reply-with (???)

## Bibliographie

- <http://jade.cselt.it/>
- Tutoriel de Jiri Vokrinek.  
<http://agents.felk.cvut.cz/teaching/ui2/JADEtutorial.doc>