

# Plateforme JADE

## Behaviour

Claude Moulin

Université de Technologie de Compiègne

IA04

# Sommaire

1 Behaviours prédéfinis

2 Behaviours utiles

# Behaviours simples

- Classe `OneShotBehaviour`. Cette classe abstraite modélise les behaviours atomiques qui ne sont exécutés qu'une seule fois et qui ne peuvent pas être bloqués. La méthode `done()` retourne toujours `true`.
- Classe `CyclicBehaviour`. Cette classe abstraite modélise les behaviours atomiques qui s'exécutent indéfiniment. La méthode `done()` retourne toujours `false`.
- Pour ces deux behaviours, il faut créer une classe qui hérite de la classe de base et surcharger la méthode `action()`.

## Autre behaviour simple (avec délai)

- Classe `WakerBehaviour`. Cette classe abstraite modélise les behaviours atomiques qui ne sont exécutés qu'une seule fois mais après un certain délai.
- Pour ce behaviour il faut créer une classe héritant de `WakerBehaviour` et surcharger la méthode `onWake()` déclenchée après le délai donné en construction.
- Le mécanisme de délai est encapsulé dans la classe de base.

## Autre behaviour simple : cyclique

- Classe `TickerBehaviour`. Cette classe abstraite modélise les behaviours atomiques qui s'exécutent indéfiniment mais qui se déclenchent à intervalle régulier.
- Pour ce behaviour il faut créer une classe héritant de `TickerBehaviour` et surcharger la méthode `onTick()` déclenchée selon l'intervalle de temps donné en construction.
- Le mécanisme de timer est encapsulé dans la classe de base.

# Behaviours composites

- Classe `CompositeBehaviour`. Cette classe abstraite est la classe de base des behaviours composites.
- Classe `SequentialBehaviour`. Cette classe exécute ses sous-behaviours de façon séquentielle et se termine lorsque le dernier est exécuté. On peut utiliser cette classe lorsqu'une tâche complexe se décompose en une suite de tâches.
- Classe `ParallelBehaviour`. Cette classe exécute ses sous-behaviours de façon concurrente et se termine lorsque soit :
  - tous ses sous-behaviours sont terminés,
  - l'un quelconque de ses sous-behaviours est terminé,
  - n de ses sous-behaviours sont terminés.

# Machine à états

- Classe `FSMBehaviour`
- Cette classe exécute ses sous-behaviours selon une machine à états finis définie par l'utilisateur.
- Chaque sous-behaviour représente l'activity exécutée dans un état de l'automate.
- Lorsque le sous-behaviour correspond à un état se termine, sa valeur de terminaison (retournée par la méthode `onEnd()` est utilisée pour déterminer la transition et atteindre le prochain état exécuté au prochain passage dans le scheduling.
- Certains sous-behaviours sont marqués comme états finals.
- Le `FSMBehaviour` se termine après terminaison de l'un des sous-behaviours état final.

# Sommaire

- 1 Behaviours prédéfinis
- 2 Behaviours utiles



# Initiateur simple

- Objectif : behaviour qui envoie un message et qui attend une réponse à ce message. Peu importe le type de la réponse. Le template de réponse est basé sur le conversation-id.
- Le message est envoyé dans le constructeur du behaviour.
- Paramètres : l'agent et le message à envoyer.
- Fonctionnement : déclenche la méthode `handleResponse` lorsque le message de réponse est parvenu.
- Le behaviour s'arrête lorsque un message est reçu.
- Classe abstraite ; la méthode `handleResponse` doit être surchargée.

# Initiateur simple

```
public abstract class BaseInitiatorBehaviour
                                extends Behaviour {

    boolean over = false;
    MessageTemplate mt;

    public BaseInitiatorBehaviour (Agent a,
                                    ACLMessage message) {

        super(a);
        String id = UUID.randomUUID().toString();
        mt = MessageTemplate.MatchConversationId(id);
        message.setConversationId(id);
        myAgent.send(message);
    }

    protected abstract void
                                handleResponse (ACLMessage message) ;

    // ... Méthodes action et done ...
}
```

# Initiateur simple

```
public abstract class BaseInitiatorBehaviour
                                extends Behaviour {
    ...
    @Override
    public void action() {
        ACLMessage response = myAgent.receive(mt);
        if (response != null) {
            handleResponse(response);
            over = true;
        } else
            block();
    }
    @Override
    public boolean done() {
        return over;
    }
}
```

# Initiateur CONFIRM REFUSE

- Objectif : behaviour qui envoie un message, qui attend une réponse à ce message basée sur le conversation-id de type CONFIRM ou REFUSE.
- Le message est envoyé dans le constructeur du behaviour.
- Paramètres : l'agent et le message à envoyer.
- Fonctionnement : déclenche la méthode :
  - `handleConfirm` lorsque une réponse CONFIRM est parvenue.
  - `handleRefuse` lorsque une réponse REFUSE est parvenue.
  - `handleOther` lorsque une autre réponse est parvenue.
- Le behaviour s'arrête lorsque un message est reçu.
- Une des méthodes doit être surchargée.

# Initiateur CONFIRM REFUSE

```
public class InitiatorBehaviour extends
    BaseInitiatorBehaviour {
    public InitiatorBehaviour(Agent a, ACLMessage message) {
        super(a, message); }
    protected void handleConfirm(ACLMessage response) {}
    protected void handleRefuse(ACLMessage response) {}
    protected void handleOther(ACLMessage response) {}
    @Override
    public void handleResponse(ACLMessage response) {
        ...
        // Choisit la méthode en fonction du type du message
    }
}
```

# Initiateur CONFIRM REFUSE

```
public class InitiatorBehaviour extends
                                BaseInitiatorBehaviour {
public InitiatorBehaviour(Agent a, ACLMessage message) {
    super(a, message); }
// Méthodes abstraites
@Override
public void handleResponse(ACLMessage response) {
    if (response.getPerformative() == ACLMessage.CONFIRM)
        handleConfirm(response);
    else
        if (response.getPerformative() == ACLMessage.REFUSE)
            handleRefuse(response);
        else
            handleOther(response);
    }
}
```

# Utilisation dans le factorielle

- Un behaviour cyclique attend un nombre de la console Jade
- Ce behaviour met en place un `AskStoreBehaviour` dont la classe hérite de `InitiatorBehaviour`.
- Ce dernier envoie un message au `StoreAgent` un message demandant si le nombre reçu est connu.
- La méthode `handleConfirm` est activée lorsque le nombre est connu et elle reçoit la valeur du factorielle.
- La méthode `handleRefuse` est activée lorsque le nombre est inconnu. Il s'agit alors de déclencher le calcul de factorielle.

## Calcul d'un factorielle

- Le behaviour qui a reçu la réponse de l'agent `StoreAgent` doit déclencher un calcul de factorielle.
- Le calcul est une suite de demandes de produits et d'attentes de résultats.
- Ceci est réalisé par un behaviour dont la classe hérite de `BaseInitiatorBehaviour`.
- Ce behaviour envoie un message à un multiplicateur et attend une réponse
- La méthode `handleResponse` traite la réponse. Si le calcul est terminé il affiche le résultat sinon il met en place un behaviour de même type qui enverra un autre calcul à faire.
- il faut ajouter dans le constructeur du behaviour les paramètres nécessaires.



# Généralisation : AchieveREInitiateur

- JADE fournit la classe `AchieveREInitiateur` pour les comportement d'envoi et d'attente de message.
- La classe fournit les méthodes `handleInform`, `handleRefuse`, `handleAgree`, `handleNotUnderstood`, déclenchées en fonction du format du message reçu.
- Il est nécessaire de surcharger les méthodes désirées.

# Initiateur à durée limitée

- Objectif : behaviour qui envoie un message et qui attend une réponse à ce message pendant une durée limitée. Peu importe le type du message.
- Le message est envoyé dans le constructeur du behaviour.
- Paramètres : l'agent, le message à envoyer et la durée.
- Fonctionnement : déclenche la méthode `handleAnswer` lorsque un message de réponse est parvenu et `handleEmpty` lorsque la durée d'attente est écoulée sans message reçu.
- Le behaviour s'arrête lorsque un message est reçu ou lorsque la durée est atteinte.
- Les méthodes `handleAnswer` et `handleEmpty` doivent être surchargées.

## Initiateur à durée limitée

```
public abstract class DelayBehaviour
                                extends Behaviour {

    int maxDelay;
    long initTime;
    MessageTemplate mt;
    boolean over = false;
    public DelayBehaviour(Agent a, ACLMessage message,
                          int maxDelay) {

        super(a);
        this.maxDelay = maxDelay;
        initTime = System.currentTimeMillis();
        String id = UUID.randomUUID().toString();
        mt = MessageTemplate.MatchConversationId(id);
        message.setConversationId(id);
        myAgent.send(message);
    }

    ... méthodes abstraites ...
}
```

## Initiateur à durée limitée

```
public abstract class DelayBehaviour
                                extends Behaviour {
    ...
    protected abstract void
        handleAnswer(ACLMessage response());
    protected abstract void handleEmpty();
    @Override
    public void action() {
        long currentTime = System.currentTimeMillis();
        ACLMessage response = myAgent.receive(mt);
        if (response != null) {
            handleAnswer(response); over = true;
        } else if (currentTime - initTime > maxDelay) {
            handleEmpty(); over = true;
        }
    }
}
... Méthode done
```

# Initiateur à durée limitée

```
public abstract class DelayBehaviour
    extends Behaviour {
    ...

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

# Utilisation possible dans le factorielle

- L'agent `fact` possède un behaviour de type `FactWaitBehaviour` qui hérite de `DelayBehaviour`.
- La construction du behaviour envoie une demande de multiplication.
- La méthode `handleAnswer` soit envoie un nouveau calcul soit affiche le résultat (factorielle terminée).
- La méthode `handleEmpty` redemande le même calcul, puisque aucune réponse n'est arrivée durant le temps imparti.