# JADE Priority-Based Scheduler Add-On

Juan A. Suárez Romero (jsuarezr@udc.es)
Amparo Alonso Betanzos (ciamparo@udc.es)
Bertha Guijarro Berdiñas (cibertha@udc.es)

October 27, 2006

# Requirements

This add-on has been tested with *Sun JDK 1.4* and *JADE 3.4*.

# Description

In the current version of *JADE*[1] an agent can contain several behaviours. Each agent contains an internal scheduler that manages its behaviuors in a round-robin manner. That is, the scheduler selects the first created behaviour and executes it[2]. Then it selects the next one, and also executes it, and so on. When the last behaviour is executed, the scheduler restart again with the first behaviour, and so forth.

The user are not able to assign more priority to one behaviour than to others: all behaviours are executed with the same priority. In some situations, as can be seen in several posts to the *JADE mail list*, it would be desirable to assign different priorities to each behaviour so the scheduler executes more frequently those behaviours with a higher priority.

The aim of this add-on is to provide a scheduler to *JADE* that is able to manage the behaviours using priorities. As it can be seen soon, this is done by providing two new `CompositeBehaviours` in which the children are scheduled using priorities. One of these behaviuors schedule its children in a parallel way, while the other behaviour schedule its children in a sequential way.

## Concurrent management of behaviours with priorities

This add-on provides a new behaviour named `ParallelPriorityBehaviour` that schedules its children in a parallel manner. This behaviour is similar to `ParallelBehaviour`, but in this case using priorities to select the executing behaviours. So those behaviours with a higher priority are executed more frequently than other behaviours with a lesser priority.

Each child is assigned by the user with a priority equal or greater than `0`. The greater the number, the lesser the priority. So a child with priority value of `3` has more priority than other behaviour with priority value of `5`. The highest priority a behaviour can reach is `0`. This priority assigned by the user is called *static priority*, and can be changed by the user at any moment.

Internally, the `ParallelPriorityBehaviour`'s scheduler uses another priority that is called *dynamic priority*. As its name indicates, this priority is increased dynamically by the scheduler in order to avoid the starvation of a behaviour. So the dynamic behaviour of a child is a number between the static priority value of this child and `0`.

The idea of the algorithm is that the scheduler executes those children with a dynamic priority value of `0`. If there are several behaviours that have dynamic priority of `0` then they are executed in the order in which they were inserted. Also, if the selected behaviour is blocked then the scheduler does not execute it. At last, if there are not children with these dynamic priority of `0` then all dynamic priorities are increased.

[1] At this moment the current version of JADE is 3.4.

[2] Executing a behaviour means calling the action() method of the behaviour.

In order to understand better the functioning of the algorithm, we will use an example. We will create a `ParallelPriorityBehaviour` with four children $A$, $B$, $C$ and $D$. Each child is assigned with (static) priority of `4`, `1`, `0` and `1`, respectively. These children are cyclic behaviours that ends when they are executed twice (see listing 1). The `ParallelPriorityBehaviour` ends when all its children end (see listing 2).

Listing 1: Source code for children

```
import jade.core.behaviours.Behaviour;

public class ChildBehaviour extends Behaviour {
  int counter=2;
  String id;

  public ChildBehaviour(String id) {
    super();
    this.id=id;
  }

  public void action() {
    System.out.print (id+"->");
    counter--;
  }

  public boolean done() {
    return counter==0;
  }
}
```

Listing 2: Source code for the agent using `ParallelPriorityBehaviour`

```
import jade.core.Agent;
import jade.core.behaviours.ParallelPriorityBehaviour;

public class ParallelAgent extends Agent {
  protected void setup() {
    ParallelPriorityBehaviour ppb = new
        ParallelPriorityBehaviour();
    this.addBehaviour(ppb);
    ppb.addSubBehaviour(new ChildBehaviour("A"), 4);
    ppb.addSubBehaviour(new ChildBehaviour("B"), 1);
    ppb.addSubBehaviour(new ChildBehaviour("C"), 0);
    ppb.addSubBehaviour(new ChildBehaviour("D"), 1);
  }
}
```

Below we show the steps followed by the scheduler. These steps are also summarized in table 1. In this table the superindex of each child is its static priority, while the subindex is its dynamic priority.

1. At first the dynamic priority of each child is assigned to the static priority value.
2. The scheduler selects those sub-behaviours that have dynamic priority of `0`, and executes it. In this case the selected sub-behaviour is $C$.
3. Again the scheduler selects those sub-behaviours with dynamic priority of `0`, that is $C$ once again.
4. Because $C$ has been executed twice, it ends. So the scheduler removes it.
5. The scheduler tries to select the children with dynamic priority of `0`. But now there are not such sub-behaviours, so the scheduler increases the dynamic priority of all sub-behaviours step by step, until one or more of them reaches a value `0` in its dynamic priority.
6. Now that there are several sub-behaviours with dynamic priority of `0`, the scheduler executes the child that was inserted first ($B$).
7. After executing once the child $B$, its dynamic priority is reset to its static priority.
8. Again the scheduler selects the sub-behaviour that has dynamic priority of `0`, that is $D$, and executes it.
9. Once $D$ is executed, its dynamic priority is reset.
10. Again, because there are not children with dynamic priority of `0`, all dynamic priorities are increased.
11. The scheduler selects the first inserted behaviour with dynamic priority of `0`, and executes it (that is, $B$ again).
12. Because this sub-behaviour has been executed twice, it ends, and the scheduler drops it.
13. The scheduler selects the next child with dynamic priority of `0`, that is $D$.
14. Also this sub-behaviour ends because it has been executed twice. Thus the scheduler removes it.
15. Because the remaining child does not have a dynamic priority of `0`, this priority is increased.
16. Now the scheduler can execute the sub-behaviour $A$.
17. After this, the dynamic priority value is reset.
18. Once again, the dynamic priority value is increased until it reaches `0`.
19. The scheduler executes $A$.
20. As $A$ was executed twice, it is removed, and the `ParallelPriorityBehaviour` ends.

| STEP | CHILDREN |
|---|---|
| 1 | $\langle A_4^4,\ B_1^1,\ C_0^0,\ D_1^1\rangle$ |
| 2 | $\langle A_4^4,\ B_1^1,\ \underline{C_0^0},\ D_1^1\rangle$ |
| 3 | $\langle A_4^4,\ B_1^1,\ \overline{C_0^0},\ D_1^1\rangle$ |
| 4 | $\langle A_4^4,\ B_1^1,\ \overline{D_1^1}\rangle$ |
| 5 | $\langle A_3^4,\ B_0^1,\ D_0^1\rangle$ |
| 6 | $\langle A_3^4,\ \underline{B_0^1},\ D_0^1\rangle$ |
| 7 | $\langle A_3^4,\ \overline{B_1^1},\ D_0^1\rangle$ |
| 8 | $\langle A_3^4,\ B_1^1,\ \underline{D_0^1}\rangle$ |
| 9 | $\langle A_3^4,\ B_1^1,\ \overline{D_1^1}\rangle$ |
| 10 | $\langle A_2^4,\ B_0^1,\ D_0^1\rangle$ |
| 11 | $\langle A_2^4,\ \underline{B_0^1},\ D_0^1\rangle$ |
| 12 | $\langle A_2^4,\ \overline{D_0^1}\rangle$ |
| 13 | $\langle A_2^4,\ \underline{D_0^1}\rangle$ |
| 14 | $\langle A_2^4\rangle$ |
| 15 | $\langle A_0^4\rangle$ |
| 16 | $\langle \underline{A_0^4}\rangle$ |
| 17 | $\langle \overline{A_4^4}\rangle$ |
| 18 | $\langle A_0^4\rangle$ |
| 19 | $\langle \overline{A_0^4}\rangle$ |
| 20 | $\langle\rangle$ |

Table 1: Example of scheduling in `ParallelPriorityBehaviour`

It is worth to note that even when the scheduler increases the dynamic priority of behaviours, it does not in account whether these behaviours are blocked or not. Only when a child is selected to be executed, the scheduler skips it if this sub-behaviour is blocked.

Also, as in the case of the current `ParallelBehaviour`, if all children are blocked then the `ParallelPriorityBehaviour` becomes blocked, until one or more children becomes again runnable.

An interesting feature is that the new `ParallelPriorityBehaviour` can replace the current `ParallelBehaviour`. If the user does not specify any priority on children, the `ParallelPriorityBehaviour` behaves exactly like the `ParallelBehaviour`.

## Sequential management of behaviours with priorities

Another composite behaviour provided by this add-on is `SequentialPriorityBehaviour`. This new behaviour manages its children in a sequential way, executing first those children with the highest priority. In this way, this behaviour is similar to the current `SequentialBehaviour`.

Each child is assigned with a priority by the user. The scheduler selects the first inserted behaviour with the highest priority, and executes it until it ends. During the execution, can happen several things. The first is that a new child with a higher priority is inserted, or that an already existent child with lesser priority changes to get a higher priority. In both cases, the scheduler stops the execution of the current behaviour and it selects the new child, starting to execute it until it ends.

The other issue happens when the selected child is blocked or becomes blocked. If we were using a `SequentialBehaviour` this composite behaviour would become blocked until the selected child becomes unblocked. In the case of the new `SequentialPriorityBehaviour` this policy can be changed. The default policy is the same as in the `SequentialBehaviour`, that is, the `SequentialPriorityBehaviour` becomes blocked until the selected child becomes runnable. The other policy is to skip the blocked children. So when the `SequentialPriorityBehaviour` selects a blocked sub-behaviour, the scheduler skips it and selects the next first inserted behaviour with the highest priority that is runnable. In both cases if a new behaviour with a higher priority appears, the scheduler selects the new one.

As in the last section, In order to understand the functioning of this new behaviour, we will use an example similar to the one used in the last section. In this case we will create a `SequentialPriorityBehaviour` using the policy of skipping the blocked children (see listing 3). Also we will create four children, $A$, $B$, $C$ and $D$, with priority `4`, `1`, `0` and `1`, respectively. This sub-behaviours are the same as in the last section, and they end when they are executed twice (see listing 1). To make the example more complete, children $A$ and $B$ are blocked. In table 2 can be noticed the steps followed by the algorithm.

Listing 3: Source code for the agent using `SequentialPriorityBehaviour`

```
import jade.core.Agent;
import jade.core.behaviours.SequentialPriorityBehaviour;

public class SequentialAgent extends Agent {
  protected void setup() {
    SequentialPriorityBehaviour spb = new
        SequentialPriorityBehaviour();
    this.addBehaviour(spb);
    spb.addSubBehaviour(new ChildBehaviour("A"), 4);
    spb.addSubBehaviour(new ChildBehaviour("B"), 1);
    spb.addSubBehaviour(new ChildBehaviour("C"), 0);
    spb.addSubBehaviour(new ChildBehaviour("D"), 1);
  }
}
```

1. At first time, the scheduler searches for the highest priority behaviour that has been inserted before. In this case, the child $C$ is selected, so the scheduler executes it.

2. Because child $C$ does not still ends, the scheduler continues executing it.

3. Now, as $C$ was executed twice, it ends. Therefore, the scheduler removes $C$.

4. The scheduler searches again for the highest priority behaviour that has been inserted first. In this case the selected behaviour is $B$. But as $B$ is blocked and the policy is to skip the blocked children, the scheduler searches for the next highest priority behaviour. And, in this case, the selected behaviour is $D$, that is not blocked. So the scheduler executes it.

| STEP | CHILDREN |
|---|---|
| 1 | $\langle A^4,\ B^1,\ \underline{C^0},\ D^1\rangle$ |
| 2 | $\langle A^4,\ B^1,\ \underline{C^0},\ D^1\rangle$ |
| 3 | $\langle A^4,\ B^1,\ D^1\rangle$ |
| 4 | $\langle A^4,\ B^1,\ \underline{D^1}\rangle$ |
| 5 | $\langle A^4,\ \underline{B^1},\ D^1\rangle$ |
| 6 | $\langle A^4,\ \underline{B^1},\ D^1\rangle$ |
| 7 | $\langle A^4,\ D^1\rangle$ |
| 8 | $\langle A^4,\ \underline{D^1}\rangle$ |
| 9 | $\langle A^4,\rangle$ |
| 10 | $\langle \underline{A^4}\rangle$ |
| 11 | $\langle \underline{A^4}\rangle$ |
| 12 | $\langle\rangle$ |

Table 2: Example of scheduling in `SequentialPriorityBehaviour`

5. After the execution of $D$, we suppose that $A$ and $B$ becomes runnable. Thus, the highest priority behaviour inserted first is $B$, and is runnable. So the scheduler selects $B$ and executes it.

6. Again, $B$ is the highest priority behaviour, so the schedulers reruns it.

7. As $B$ ends, it is removed.

8. The scheduler searches for the new highest priority behaviour, which is $D$. So the scheduler executes it.

9. As $D$ has been executed twice, it is removed from the `SequentialPriorityBehaviour`.

10. Now the only remaining child is $A$, and because it is runnable, the scheduler executes it.

11. As $A$ does not still end, it is executed once again.

12. $A$ ends, so it is removed. And because all children have finished, the `SequentialPriorityBehaviour` ends.

It is worth to note that if all children are blocked, the `SequentialPriorityBehaviour` becomes blocked.

Also, if we do not assign priorities and we use the default policy (that is, do not skip the blocked sub-behaviours), the `SequentialPriorityBehaviour` behaves exactly like the current `SequentialBehaviour`.

# Credits

Last update: October 27, 2006

Authors:

- Juan Antonio Suárez Romero
- Amparo Alonso Betanzos

• Bertha Guijarro Berdiñas

Laboratory for Research and Development in Artificial Intelligence (LIDIA). Computer Science Department. University of A Coruña, Spain.