BDI4JADE Tutorial

Matheus Dias (revised by Ingrid Nunes)

November 4, 2014

Contents

1 Introduction

BDI4JADE[1] is an agent platform that implements the (belief-desire-intention) (BDI) [4] architecture. It consists of a BDI layer implemented on top of JADE[2] [1]. BDI4JADE leverages all the features provided by JADE and reuses it as much as possible. Other highlights of our JADE extension, besides providing BDI abstractions and the reasoning cycle, include:

- **Use of Capabilities** — agents aggregate a set of capabilities, which are a collection of beliefs and plans, and allow modularisation of particular agent functionality.
- `PlanBody` **is an Extension of JADE** `Behaviour` — in order to better exploit JADE features, plan bodies are subclasses of JADE behaviours.
- **Java Annotations** — annotations are provided to allow easier configuration of agent components, without compromising its flexibility.
- **Extension Points** — strategies can be easily implemented to extend parts of the reasoning cycle, such as belief revision and plan selection.
- **Listeners and Events** — different events (such as events related to goals and beliefs) can be observed in the platform, allowing listeners to react according to events that occurred.
- **Java Generics for Beliefs** — beliefs can store any kind of information and are associated with a name, and if the value of a belief is retrieved, it must be cast to its specific type, so the use of Java generics allows us to capture incorrect castings at compile time.

As opposed to different BDI platforms that have been proposed, it does not introduce a new programming language nor rely on a domain-specific language (DSL) written in terms of XML files. Because agents are implemented with the constructions of the underlying programming language, Java, we bring two main benefits. First, features of the Java language, such as annotations and reflection, can be exploited for the development of complex applications. Second, it facilitates the integration of existing technologies, e.g. frameworks and libraries, which is essential for the development of large scale enterprise applications, involving multiple concerns such as persistence and transaction management. This also enables a smooth adoption of agent technology.

2 The `HelloWorld` Application

In this section, we explain details of how the `HelloWorld` application is implemented. There are two versions of it. The first, detailed in Section 2.1, is

[1] `http://inf.ufrgs.br/prosoft/bdi4jade`
[2] `http://jade.tilab.com/`

implemented as an agent. The second, described in Section 2.2, is implemented as a capability, which can be associated with any agent. In order to run these, and other provided examples, you should run the `BDI4JADEExamplesApp` class.

2.1 Alternative 1: `HelloWorldAgent`

The `HelloWorld` is an application in which an agent has the goal of saying hello world, and a plan that can achieve it by printing in the main console "Hello World, *name*!," where *name* is given as parameter of the agent goal.

Agents in BDI4JADE are associated with a single or multiple capabilities. The agent that is part of the `HelloWorld` application has a single capability, and is implemented in the class `HelloWorldAgent`, which extends the `SingleCapabilityAgent`. Alternatively, you may create an agent that may be associated with multiple capabilities. In this case, instated of extending `SingleCapabilityAgent`, you must extend `MultipleCapabilityAgent`.

```
34 public class HelloWorldAgent extends SingleCapabilityAgent
```

Goals may be added to agents, which in BDI4JADE are objects that are instance of any Java class that implements the `Goal` interface. In our example, the goal of saying hello world is implemented in the inner class `HelloWorldGoal`, whose enclosing class is the `HelloWorldAgent` class. `HelloWorldGoal` has a single constructor of receives a parameter that indicates the name that should be used in the hello world message. This parameter may be later retrieved by a getter.

```
36     public static class HelloWorldGoal implements Goal {
37         private static final long serialVersionUID = -9039447524062487795L;
38 
39         private String name;
40 
41         public HelloWorldGoal(String name) {
42             this.name = name;
43         }
44 
45         public String getName() {
46             return name;
47         }
48     }
49 
```

In order for an agent to be able to achieve this goal, it must have a plan that achieves it. A plan has some informational details about it, such as a template that specifies which goals it can achieve, and a body. In our example, a plan body is implemented in the `HelloWorldPlanBody` class, which extends the `AbstractPlanBody` class and is also implemented as an inner class of the `HelloWorldAgent` class. The implementation of the `action()` method consists

of printing the hello world message and then setting the end state of the plan as successful (`EndState.SUCCESSFUL`).

```
50      public static class HelloWorldPlanBody extends AbstractPlanBody {
51          private static final long serialVersionUID = -9039447524062487795L;
52
53          public void action() {
54              System.out.println("Hello, "
55                      + ((HelloWorldGoal) getGoal()).getName() + "!");
56              setEndState(EndState.SUCCESSFULL);
57          }
58      }
```

The method `getGoal()` (part of the `PlanBody` interface implemented by the `AbstractPlanBody` class) returns the goal that triggered the execution of the plan, and in this case we know it is an instance of `HelloWorldGoal` class. So, to get the parameter of the goal to print the hello world message, we invoke the `getGoal()` method, cast its return to `HelloWorldGoal`, and get its parameter by invoking the `getName()` method.

Given that we now have a plan body — the `HelloWorldPlanBody` — we need to create a `Plan` and add it to the agent. This is done in the constructor of the `HelloWorldAgent` class. The default constructor of the `SingleCapabilityAgent` class (which is the `HelloWorldAgent` parent class) is implicitly invoked in the `HelloWorldAgent` constructor. This implicitly invoked constructor instantiates a `Capability` and associates it with the agent. This `Capability` can be accessed by the `getCapability()` method. Capabilities are associated with a plan library, accessed by the `getPlanLibrary()` method, which has the method `addPlan()`, which in turn adds a plan to the capability plan library. So in the `HelloWorldAgent` constructor, we add a plan that is an instance of `DefaultPlan` class to the capability plan library. This class has a constructor that receives as a parameters (i) a class of goals that can be achieved by the plan; and (ii) a class that implements `PlanBody`. Therefore, in our case, we instantiate the `DefaultPlan` with the goal class `HelloWorldGoal`, and plan body class `HelloWorldPlanBody`.

```
62      public HelloWorldAgent() {
63          getCapability().getPlanLibrary()
64          .addPlan(new DefaultPlan(HelloWorldGoal.class,HelloWorldPlanBody.class));
65      }
66
```

The `HelloWorldAgent` can be instantiated and executed like any JADE agent. After the `HelloWorldAgent` is started, an instance of the `HelloWorldGoal` must be added to the agent by invoking the method `addGoal()`, in order to the agent try to, and eventually achieve, this goal.

2.2 Alternative 2: `HelloWorldAnnotatedCapability`

The previous section showed how to create an agent, without creating a capability separately. In this section, we detail an alternative implementation of the `HelloWorld` application, which consists of a capability that can be added to agents, taking advantage of the BDI4JADE annotations.[3] The created capability is the `HelloWorldAnnotatedCapability` class, which extends the `Capability` class.

```
public class HelloWorldAnnotatedCapability extends Capability {
```

Simular to the previous version of the `HelloWorld` application, there is an inner class that represents the goal to be achieved — the `HelloWorldGoal`.

```
40      @GoalOwner(capability = HelloWorldAnnotatedCapability.class)
41      public static class HelloWorldGoal implements Goal {
42          private static final long serialVersionUID = -9039447524062487795L;
43
44          private String name;
45          private long time;
46
47          public HelloWorldGoal(String name) {
48              this.name = name;
49          }
50
51          @Parameter(direction = Direction.IN)
52          public String getName() {
53              return name;
54          }
55
56          public void setTime(long time) {
57              this.time = time;
58          }
59
60          @Parameter(direction = Direction.OUT)
61          public long getTime() {
62              return time;
63          }
64
65          @Override
66          public String toString() {
67              return getClass().getSimpleName() + " - name: " + name
68                      + " / time: " + time;
69          }
70      }
```

This goal was implemented with additional features. We used the `@GoalOwner` annotation to indicate to which capability this goal belongs and, given that we do not specify whether this goals is external or internal, it is considered *external* because it is the default value. The consequence of having a goal owner is that this goal can be achieved only by the capability to which it belongs (or part

[3] `http://docs.oracle.com/javase/tutorial/java/annotations/`

or child capabilities). For further information about it, we refer the reader to elsewhere [2, 3].

Besides using the `@GoalOwner` annotation, we used the `@Parameter` annotation to indicate goal parameters. There is one input parameter (`Direction.IN`), which is the *name* to be displayed in the hello world message. It is specified in the goal constructor and can be retrieved by a getter. In addition, there is an output parameter (`Direction.OUT`) named *time*, which will be set with the time when the hello world message is displayed. The time parameter has both a setter, to be used by the plan body that achieves this goal to set its value, and a getter, used to get this value afterwards.

In order to create a plan to achieve the `HelloWorldGoal`, we also need to create a plan body, whose name is `HelloWorldPlanBody`, which also extends the `AbstractPlanBody` class and is also implemented as an inner class of the capability. By using the `@Parameter` annotation, the plan body can automatically get input parameters and set output parameters. Given that there is a setter of *name*, which is annotated as an input parameter, the value obtained by a getter of *name* from the goal that triggered the plan body execution will be set by invoking this setter, before executing the plan body. Similarly, after the plan body execution, the value obtained by the getter of *time* in the plan body will be used to set the value of the output parameter with the same name of the goal that triggered the plan body. Therefore, based on the annotations, BDI4JADE automatically sets input and output parameters.

```
72      public static class HelloWorldPlanBody extends AbstractPlanBody {
73          private static final long serialVersionUID = -9039447524062487795L;
74
75          private String name;
76          private long time;
77
78          public void action() {
79              System.out.println("Hello, " + name + "!");
80              this.time = System.currentTimeMillis();
81              setEndState(EndState.SUCCESSFULL);
82          }
83
84          @Parameter(direction = Direction.OUT)
85          public long getTime() {
86              return time;
87          }
88
89          @Parameter(direction = Direction.IN)
90          public void setName(String name) {
91              this.name = name;
92          }
93      }
94
```

Finally, to add a plan to a capability using annotations, we add an attribute whose type is `Plan`, or any class that implements it, to the capability. This attribute should be annotated with the `@Plan` annotation, as shown below.

A `DefaultPlan` is instantiated to initialise the `plan` attribute. The `DefaultPlan` receives as parameter the `HelloWorldGoal` class and the `HelloWorldPlanBody`

```
97      @bdi4jade.annotation.Plan
98      private Plan plan = new DefaultPlan(HelloWorldGoal.class,
99              HelloWorldPlanBody.class);
100
```

class, indicating that it is a plan that can achieve goals instance of `HelloWorldGoal` and, to do so, the plan body `HelloWorldPlanBody` must be executed.

The `HelloWorldAnnotatedCapability` capability can then be instantiated and added to any agent or associated with other capabilities.

References

[1] Fabio Luigi Bellifemine, Giovanni Claire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, Inc., New York, USA, 2007.

[2] Ingrid Nunes. Capability relationships in BDI agents. In *The 2nd International Workshop on Engineering Multi-Agent Systems (EMAS 2014)*, Paris, 2014.

[3] Ingrid Nunes. Improving the design and modularity of bdi agents with capability relationships. In *EMAS 2014 Post-proceedings (to appear)*, 2014.

[4] A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.