BDI4JADE Tutorial

Matheus Dias and Ingrid Nunes

November 23, 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

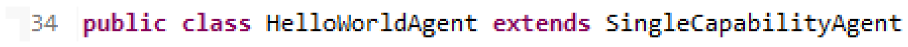
[1]`http://inf.ufrgs.br/prosoft/bdi4jade`
[2]`http://jade.tilab.com/`

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`

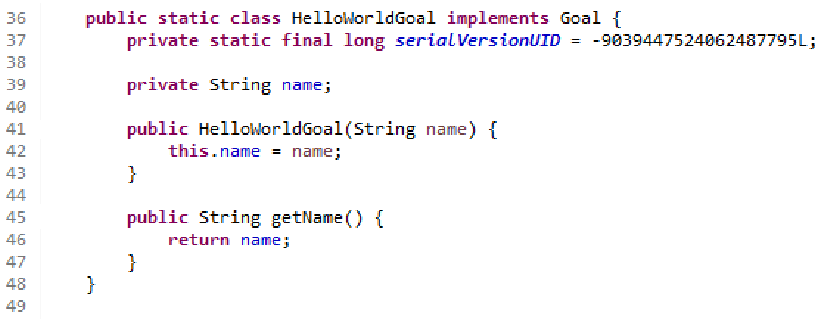
The `HelloWorld` is an application in which an agent has the goal of saying hello world, and a plan that can achieve it by printing in the main console "Hello World, *name*!," where *name* is given as parameter of the agent goal.

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`.



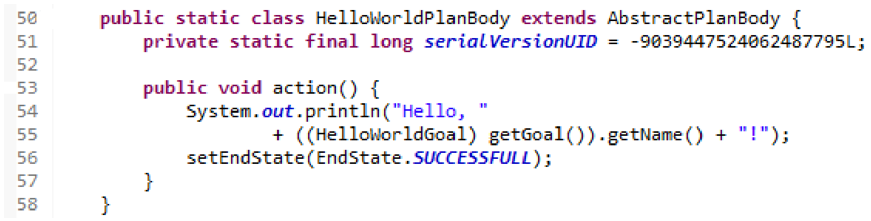
```
public class HelloWorldAgent extends SingleCapabilityAgent
```

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.



```
    public static class HelloWorldGoal implements Goal {
        private static final long serialVersionUID = -9039447524062487795L;

        private String name;

        public HelloWorldGoal(String name) {
            this.name = name;
        }

        public String getName() {
            return name;
        }
    }

```

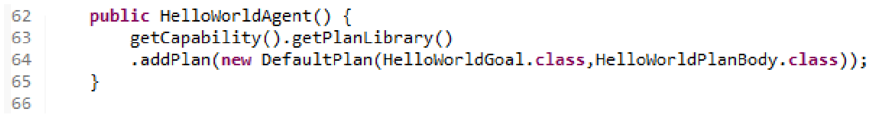
In order for an agent to be able to achieve this goal, it must have a plan that achieves it. A plan has some informational details about it, such as a template that specifies which goals it can achieve, and a body. In our example, a plan body is implemented in the `HelloWorldPlanBody` class, which extends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



```
    public static class HelloWorldPlanBody extends AbstractPlanBody {
        private static final long serialVersionUID = -9039447524062487795L;

        public void action() {
            System.out.println("Hello, "
                    + ((HelloWorldGoal) getGoal()).getName() + "!");
            setEndState(EndState.SUCCESSFULL);
        }
    }
```

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`.

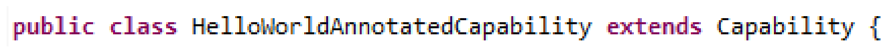


```
    public HelloWorldAgent() {
        getCapability().getPlanLibrary()
        .addPlan(new DefaultPlan(HelloWorldGoal.class,HelloWorldPlanBody.class));
    }

```

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.

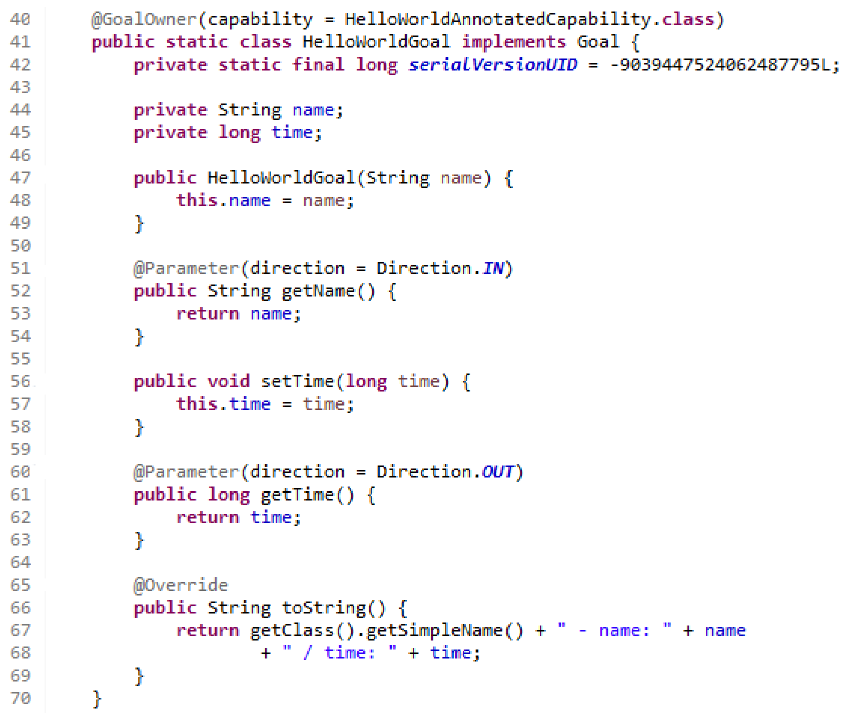
2.2 Alternative 2: HelloWorldAnnotatedCapability

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.



```
    @GoalOwner(capability = HelloWorldAnnotatedCapability.class)
    public static class HelloWorldGoal implements Goal {
        private static final long serialVersionUID = -9039447524062487795L;

        private String name;
        private long time;

        public HelloWorldGoal(String name) {
            this.name = name;
        }

        @Parameter(direction = Direction.IN)
        public String getName() {
            return name;
        }

        public void setTime(long time) {
            this.time = time;
        }

        @Parameter(direction = Direction.OUT)
        public long getTime() {
            return time;
        }

        @Override
        public String toString() {
            return getClass().getSimpleName() + " - name: " + name
                    + " / time: " + time;
        }
    }
```

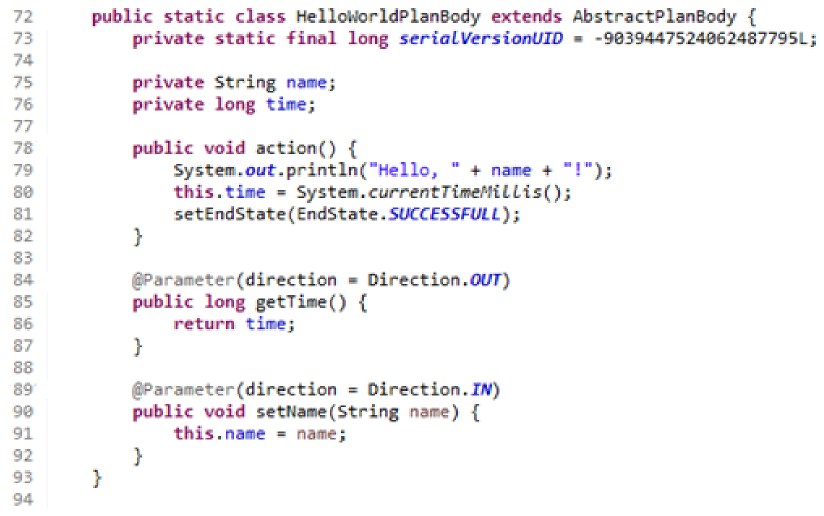
This goal was implemented with additional features. We used the @GoalOwner annotation to indicate to which capability this goal belongs and, given that we do not specify whether this goals is external or internal, it is considered *external* because it is the default value. The consequence of having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

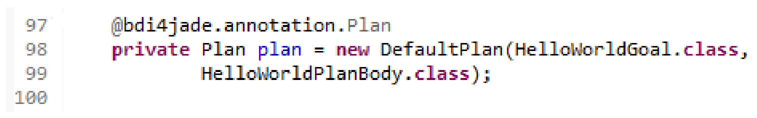
In order to create a plan to achieve the `HelloWorldGoal`, we also need to create a plan body, whose name is `HelloWorldPlanBody`, which also extends the `AbstractPlanBody` class and is also implemented as an inner class of the capability. By using the `@Parameter` annotation, the plan body can automatically 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.



```
    public static class HelloWorldPlanBody extends AbstractPlanBody {
        private static final long serialVersionUID = -9039447524062487795L;

        private String name;
        private long time;

        public void action() {
            System.out.println("Hello, " + name + "!");
            this.time = System.currentTimeMillis();
            setEndState(EndState.SUCCESSFULL);
        }

        @Parameter(direction = Direction.OUT)
        public long getTime() {
            return time;
        }

        @Parameter(direction = Direction.IN)
        public void setName(String name) {
            this.name = name;
        }
    }
```

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`



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

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.

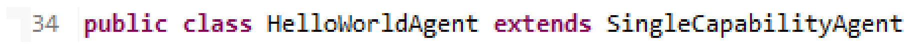
The `HelloWorldAnnotatedCapability` capability can then be instantiated and added to any agent or associated with other capabilities.

3 BDI4JADE Main Components

3.1 BDI Agents

A BDI agent represents an agent that follows the BDI software model. And have a reasoning cycle, responsible for driving the agent behaviour, strategies, and capabilities.

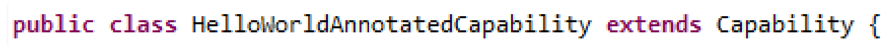
The agent class must extends the capability, and may have a goal class and a planbody class. The goal class and the planbody class can be declared in the capability.



```
public class HelloWorldAgent extends SingleCapabilityAgent
```

3.2 Capabilities

A capability in BDI4JADE is self-contained in the agente the capability consisting of (i) a set of plans, (ii) a fragment of the knowledge base that is manipulated by these plans and (iii) a specification of the interface to the capability.

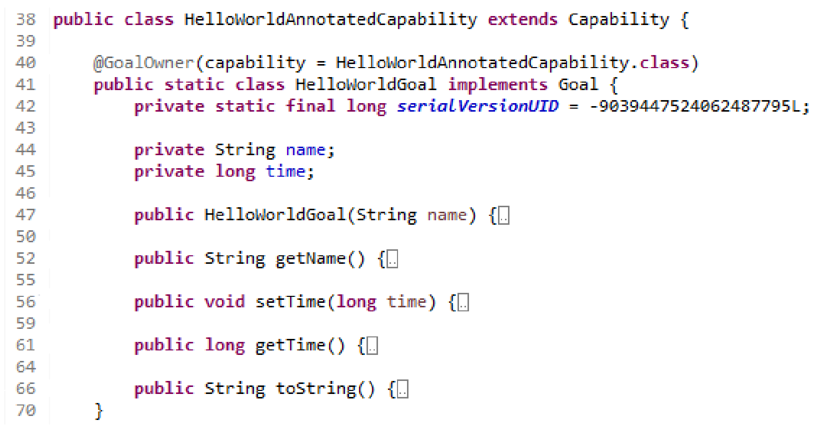


```
public class HelloWorldAnnotatedCapability extends Capability {
```

3.3 Goals

Goals represent the motivational state of the system, and represent desires that the agent wants to achieve.

A goal in BDI4JADE can be any Java object, with the condition that it must implement the Goal interface.the implementation of the goal interface may need the annotation @link GoalOwner in order to especify the capability that own this goal.If is required add a new goal to an agent, the only thing that must be done is to invoke the method void addGoal(Goal goal) of an instance of the BDIAgent.



```
public class HelloWorldAnnotatedCapability extends Capability {

    @GoalOwner(capability = HelloWorldAnnotatedCapability.class)
    public static class HelloWorldGoal implements Goal {
        private static final long serialVersionUID = -9039447524062487795L;

        private String name;
        private long time;

        public HelloWorldGoal(String name) {

        public String getName() {

        public void setTime(long time) {

        public long getTime() {

        public String toString() {
    }
```

3.4 Plans

3.4.1 Plans

The representation of plans in BDI4JADE is not associated with one but with a set of classes. This is because our goal is to reuse JADE as much as possible. In the BDI4JADE platform there is three main kind of plans.

Plans do not state a set of actions to be executed to achieve a goal, but have some information about them.Beside this they define the methods that will be implemented by the sub-classes.

3.4.2 Plan Bodies

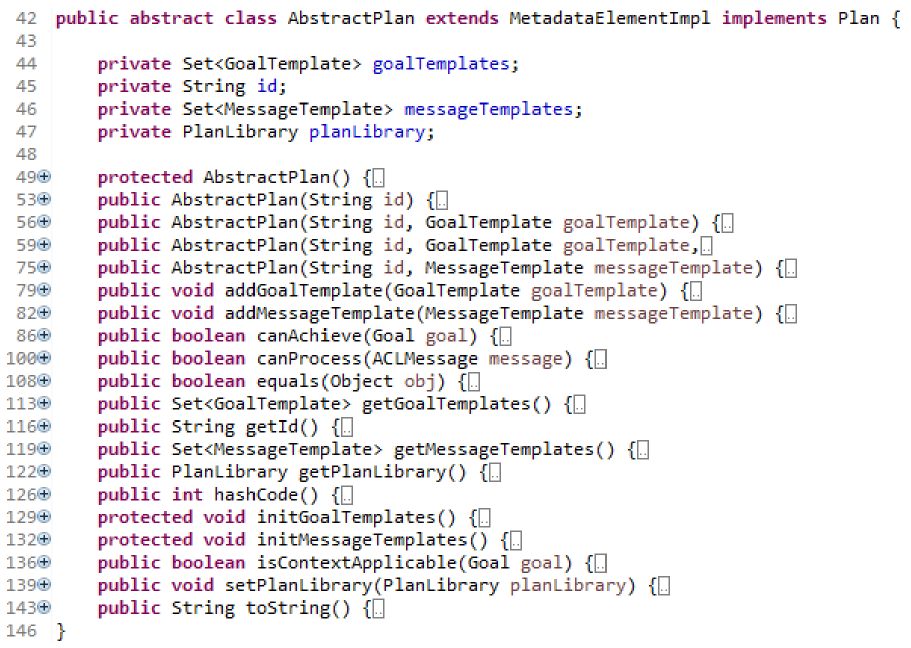
Plan bodies must implement the `PlanBody` interface and extend the JADE `Behaviour` class. Alternatively, a plan body can simply extend the `AbstractPlanBody` class. A plan body must implement the method `action()` that contains actions to achieve goals. To indicate that a plan has completed its execution, the `setEndState(EndState)` must be invoke, with one of two parameters: (i) `EndState.SUCCESSFUL`; or (ii) `EndState.FAILED`. The former indicates that the plan was successfully completed, while the latter indicates that the plan failed. Before the end state of a plan is set, the method `getEndState()` returns `null`, meaning that the plan is still being executed.

The latter is the void init (PlanInstance planinstance), is invoked when the plan body is instantiated. This is used to initialise it, for instance retrieving parameters of the goal to be achieved.

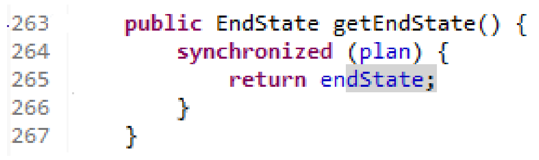
Behaviour behaviour The behaviour being executed to achieve the goal associated with the intention.

Intention The intention whose goal is trying to be achieved.

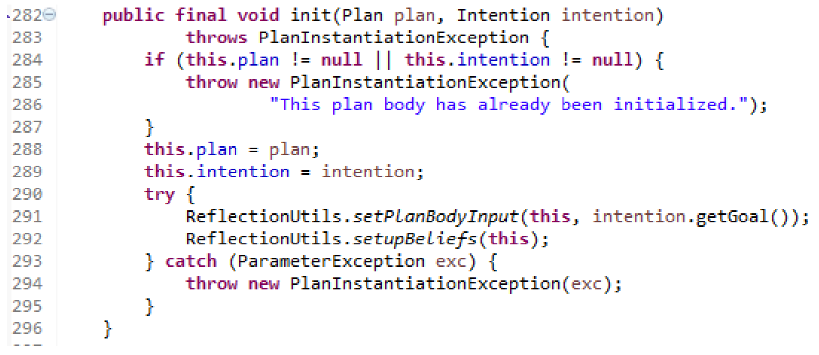
Plan plan The plan that this plan instance is associated with.



```java
public abstract class AbstractPlan extends MetadataElementImpl implements Plan {

    private Set<GoalTemplate> goalTemplates;
    private String id;
    private Set<MessageTemplate> messageTemplates;
    private PlanLibrary planLibrary;

    protected AbstractPlan() {□
    public AbstractPlan(String id) {□
    public AbstractPlan(String id, GoalTemplate goalTemplate) {□
    public AbstractPlan(String id, GoalTemplate goalTemplate,□
    public AbstractPlan(String id, MessageTemplate messageTemplate) {□
    public void addGoalTemplate(GoalTemplate goalTemplate) {□
    public void addMessageTemplate(MessageTemplate messageTemplate) {□
    public boolean canAchieve(Goal goal) {□
    public boolean canProcess(ACLMessage message) {□
    public boolean equals(Object obj) {□
    public Set<GoalTemplate> getGoalTemplates() {□
    public String getId() {□
    public Set<MessageTemplate> getMessageTemplates() {□
    public PlanLibrary getPlanLibrary() {□
    public int hashCode() {□
    protected void initGoalTemplates() {□
    protected void initMessageTemplates() {□
    public boolean isContextApplicable(Goal goal) {□
    public void setPlanLibrary(PlanLibrary planLibrary) {□
    public String toString() {□
}
```



```java
    public EndState getEndState() {
        synchronized (plan) {
            return endState;
        }
    }
```



```java
    public final void init(Plan plan, Intention intention)
            throws PlanInstantiationException {
        if (this.plan != null || this.intention != null) {
            throw new PlanInstantiationException(
                    "This plan body has already been initialized.");
        }
        this.plan = plan;
        this.intention = intention;
        try {
            ReflectionUtils.setPlanBodyInput(this, intention.getGoal());
            ReflectionUtils.setupBeliefs(this);
        } catch (ParameterException exc) {
            throw new PlanInstantiationException(exc);
        }
    }
```

EndState endstate The end state of the plan instance (FAILED or SUCCESSFUL), or null if it is currently being executed.

List⟨Goal⟩ subgoals The subgoals dispatched by this plan. In case the goal of the intention associated with the plan of this plan instance is dropped, all subgoals are also dropped.

List⟨GoalFinishedEvent⟩ goalEventQueue When this plan instance dispatches a goal, it can be notified when the dispatched goal finished.

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)*, pages 56–72, 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.