ENGINEERING
INGEGNERIA
INFORMATICA

R&D Lab Palermo, Italy
http://res.eng.it

PRACTIcal reasONIng sySTem

PRACTIONIST Framework

Programming guide

Table of contents

1 INTRODUCTION **3**

2 AGENT STRUCTURE **4**

2.1 DELIBERATION PROCESS 5
2.2 MEANS-ENDS REASONING 7

3 PROGRAMMING PRACTIONIST AGENTS **8**

3.1 EXTENDING PERCEPTUAL CAPABILITIES 9
3.2 IMPLEMENTING EFFECTING CAPABILITIES 12
3.3 DEFINING BELIEFS 14
3.4 DEFINING GOALS 18
3.4.1 Relationships among goals *20*
3.5 DEFINING NEW PLANS 22
3.5.1 Abstract class `Plan` *24*
3.5.2 Plans to react to and handle perceptions: `PerceptionPlan` *25*
3.5.3 Plans to handle received messages: `MessageReceivingPlan` *25*
3.5.4 Plans to handle received generic `Objects`*:* `ObjectReceivingPlan` *26*
3.5.5 Plans to handle changes in beliefs: `BeliefUpdatePlan` *26*
3.5.6 Plans to pursue intentions: `GoalPlan` *27*
3.5.6.1 StartPlan 28
3.5.6.2 PerformPlan 29
3.5.6.3 AchievePlan 29
3.5.6.4 AchieveAbsPredicatePlan 30
3.5.6.5 CeasePlan 30
3.5.6.6 CeaseAbsPredicatePlan 31
3.6 USING ACTS TO IMPLEMENT PLAN BODIES 31
3.6.1 Querying and updating belief base *32*
3.6.2 Sending and receiving messages *33*
3.6.3 Waiting for perceptions *34*
3.6.4 Desiring to pursue a goal *35*
3.6.5 Doing actions *36*

1 Introduction

This document presents how to implement Belief-Desire-Intention (BDI) agents by using PRACTIONIST (PRACTIcal reasONIng sySTem), a framework, developed at the Research & Development Laboratory of ENGINEERING Ingegneria Informatica S.p.A.

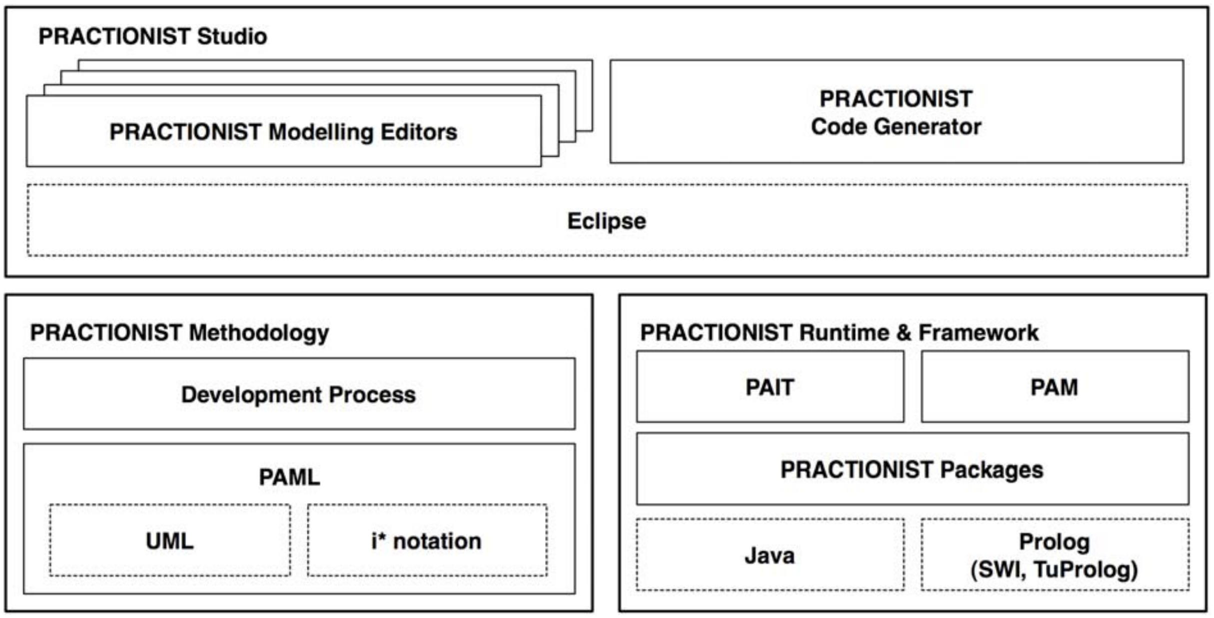

Figure 1 – The PRACTIONIST suite.

PRACTIONIST is a suite of tools including (see Figure 1):

- the **PRACTIONIST Methodology**, consisting of the UML-based PRACTIONIST Agent Modelling Language (PAML) and an iterative and incremental development process;
- the **PRACTIONIST runtime and framework (PRF)**, which defines and supports the execution logic and provides the built-in components according to such a logic to support the development of BDI agents in Java (using JADE) with a Prolog belief base. The PRF also includes the PRACTIONIST Agent Introspection Tool (PAIT), to monitor the intentional components of each agent, and the PRACTIONIST Autonomic Manager (PAM), which enables PRACTIONIST applications to support the self-chop features (self-configuring, self-healing, self-optimizing and self-protecting);
- the **PRACTIONIST Studio**, a design and development tool which supports the methodology.

The framework provides each developed agent with necessary facilities for interfacing with the environment in which it lives: cognitive capacities on the environment and of execution of actions. The agents also own an explicit, though partial and often not completely determined representation of the environment in which they live. Thus the framework allows to develop agents able to reason on the state of world and the state of their execution.

As shown in Figure 2, the PRACTIONIST framework, implemented in Java, is based on a belief base implemented in Prolog. The framework has been designed using some features offered by the JADE platform, which implements FIPA specifications and provides some services essential to execute agents, such as support for communication, interaction protocols and management of the life cycle of agents.

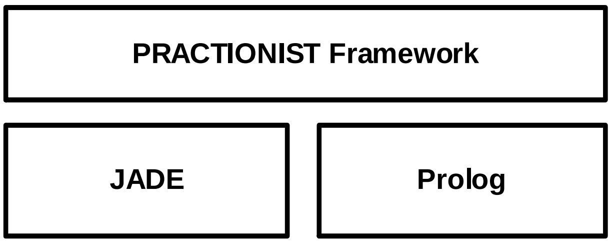

Figure 2 - The PRACTIONIST Framework on top of JADE and Prolog.

It should be noted that the methodology and the PRACTIONIST Studio are not covered by this document, which focuses on the development of individual agents and their components exploiting the features and functionalities offered by the PRACTIONIST framework.

Before, in next section a brief overview of the structure and practical reasoning mechanisms of PRACTIONIST agents is provided. Then, starting from section 3, the specific components of the PRACTIONIST agent will be presented, providing useful guidelines for their design and implementation; besides, several examples of source code will be shown, corresponding to the specific topics covered.

2 Agent structure

In Figure 3 the structure of each PRACTIONIST agent is shown, in terms of its main components. Agents are structured in two main layers: the framework defines the execution logic and provides some built-in services implementing such a logic, while the top layer includes the specific agent components to be defined in order to satisfy specific application requirements.

Therefore, a developer who wants to design a PRACTIONIST agent has to specify:

1. the *goals* the agent could pursue and the relationships among them (*Goal Model*);
2. a set of plans (*Plan Library*) to pursue that goals or react to stimuli coming from environment;
3. a set of *perceptors* to receive such stimuli;
4. the *Actions* the agent could perform and the corresponding *Effectors*;
5. the set of *beliefs* and rules (*Belief Base*) to model information about both its internal state and the external world.

Then the framework provides the required built-in services that define the computational model of PRACTIONIST agents.

This includes the Belief Logic, the Deliberation mechanisms that produce agent intentions, the way the agent makes Means-ends Reasoning to figure out the means (i.e. plans) to achieve its intentions, and the support for the actual execution of such plans.

Moreover, agents are endowed with to dynamically build plans (i.e. Planning).

Finally the management of perceptors and effectors is part of the agent core services infrastructure.

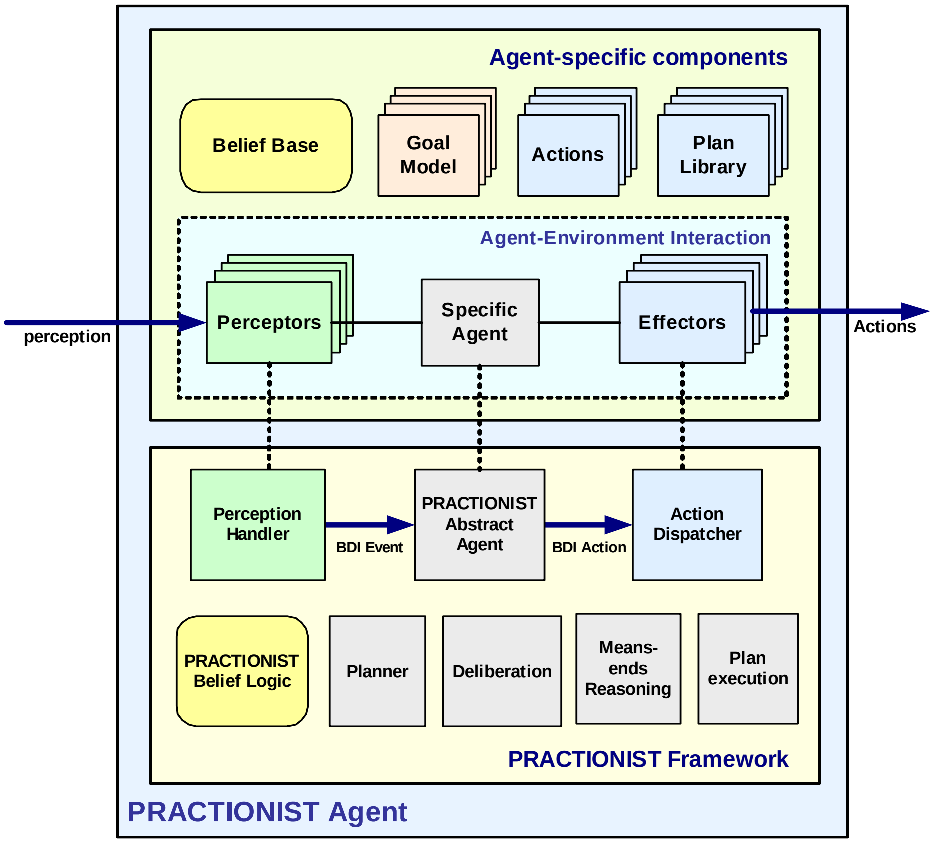

Figure 3 - Structure of a PRACTIONIST agent.

Thus a developer will define and implement the specific components of PRACTIONIST agents, without taking care of the implementation of the general mechanisms that characterize their computational logic, which are already implemented in the modules offered by the framework.

In the next two sections the deliberation process and means-ends reasoning in PRACTIONIST will be explained.

2.1 Deliberation process

The PRACTIONIST framework adopts a goal-oriented approach to develop BDI agents and stresses the separation between the deliberation process and the means-ends reasoning, with the abstraction of goal used to formally define both desires and intentions during the deliberation phase.

Therefore PRACTIONIST agents can be programmed in terms of goals, which then will be related to either desires or intentions according to whether some specific conditions are satisfied or not.

The PRACTIONIST framework supports the explicit representation and implementation of goals with their properties and relations. This provides the ability to reason about them, in terms of believing if goals are impossible, achieved, incompatible with other goals, and so forth. This in turn supports the commitment strategies of agents and their capability to autonomously drop, reconsider, replace or pursue goals. Formally, a goal g is defined as follows:

$$g = \langle \sigma_g, \pi_g, \gamma_g \rangle$$

where σ_g is the success condition, π_g is the possibility condition stating whether g can be achieved or not, and γ_g is the cancel condition stating in which situations the agent should give up to pursue the goal g.

With the aim of defining the goal model of PRACTIONIST agents, we first provide the following definitions regarding some useful relations among goals:

- g_1 is inconsistent with a goal g_2 if and only if when g_1 succeeds, then g_2 fails; a goal g_1 entails a goal g_2 (or equivalently g_2 is entailed by g_1) if and only if when g_1 succeeds, then also g_2 succeeds;
- a goal g_1 is a precondition of a goal g_2 if and only if, to be possible to pursue g_2 , g_1 must succeed;
- a goal g_1 depends on a goal g_2 if g_2 is precondition of g_1 and g_2 must be successful while pursuing g_1 .

When two goals are inconsistent with each other, we can also specify that one is preferred to the other. In PRACTIONIST several goals can be pursued in parallel. Therefore there is no need to prefer some goal to another goal if they are not inconsistent with each other.

The goal model of PRACTIONIST agents contains the set of goals they could pursue and all above-defined relations among such goals (i.e. inconsistence, entailment, precondition, and dependence).

The goal model is used by PRACTIONIST agents when reasoning about goals during their deliberation process. For them, desires and intentions are mental attitudes towards goals, which are in turn considered as descriptions of objectives.

Thus, ”pursuing the goal g” is only a desire if the agent is not yet committed to it, due to some reason. On the other hand, ”pursuing the goal g” becomes an intention when the agent is committed to it and works to achieve it.

Suppose that an agent starts its deliberation process (see Figure 4) and generates a goal $g = \langle \sigma_g, \pi_g, \gamma_g \rangle$ as an option.

Therefore the agent desires to pursue the goal g. But any agent will not be able to achieve all its desires; thus it checks if it believes that the goal g is possible (i.e. if it believes that π_g is true) and not inconsistent with active goals (i.e. those goals that the agent is currently committed to), the desire to pursue g will be promoted to an intention. Otherwise, in case of inconsistence among g and some active goals, the desire to pursue g will become an intention only if g is preferred to all inconsistent active goals, which will in turn be dropped.

Therefore, at the end of the deliberation process, a PRACTIONIST agent could either generate a new intention or remain with an impossible desire; it could drop some existing intentions as well. This ability avoids that agents try to pursue impossible and inconsistent goals (at least from their point of view).

When a desire to pursue g is promoted to an intention, before starting the means-ends reasoning, the agent checks if it believes that the goal g succeeds (that is, if it believes that the success condition σ_g holds) or if the goal g is entailed by some of the current active goals (i.e. some other means is working to achieve a goal that entails the goal g). If either, there is no reason to pursue the goal g and the agent does not need to make any means-ends reasoning to figure out how to pursue it.

Otherwise, before starting the means-ends reasoning, if some declared goals are precondition for g, the agent will first desire to pursue them and then the goal g.

As a default, PRACTIONIST agents adopt a single-minded intention commitment strategy. Thus, it will continue to maintain an intention until it believes that either such an intention has been achieved or it is no longer possible to achieve the intention. Moreover the agent checks if some dependee goal does not succeed. If so, it will desire to pursue it and then continue pursuing the goal g.

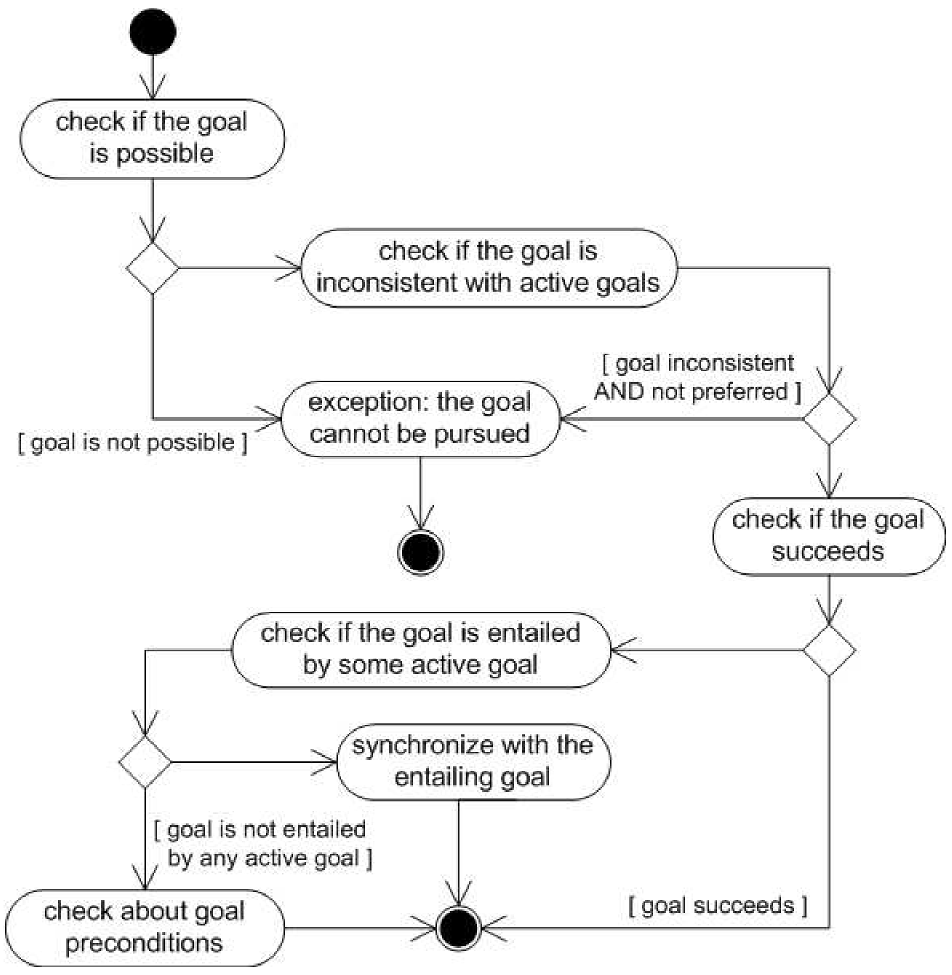
check if the goal is possible
check if the goal is inconsistent with active goals
[goal inconsistent AND not preferred]
exception: the goal cannot be pursued
[goal is not possible]
check if the goal succeeds
check if the goal is entailed by some active goal
synchronize with the entailing goal
[goal is not entailed by any active goal]
check about goal preconditions
[goal succeeds]

Figure 4 - The deliberation process.

2.2 Means-Ends reasoning

In the PRACTIONIST framework plans represent an important container where it is possible to define the actual behaviours of agents. Each agent may own a declared set of plans (the plan library), which specify the activities the agent should undertake in order to achieve its intentions, or handle incoming perceptions, or react to changes of its beliefs.

Several information about plans can be specified (the complete list of such slots is reported in Table 3.2), in order to provide the agents with the capability to dynamically behave when selecting and executing plans. Thus, a plan represents a possible recipe to manage the trigger event, which in turn may be related to a goal, an external event, or an event which notifies a change of the belief set. How to actually handle a certain event is reported within the body, which is an activity that can contain a set of acts, such as desiring to pursue some goal, adding or removing beliefs, sending ACL messages, doing an action and so forth.

Through the perceptors, a PRACTIONIST agent receives perceptions from the environment and transforms them into events, which are put into an event queue. It also contains internal goal events, which are generated when the desire to pursue a given goal is promoted to an intention and some means-ends reasoning is required to figure out how to achieve such an intention. The queue also collects events related to changes in agent's beliefs.

For each event *e* extracted from the queue, the agent performs the following *means-ends reasoning*:

1. it figures out the *practical* plans, which are those plans whose trigger event matches the selected event;
2. among practical plans, the agent detects the *applicable* ones, which are those plan whose context is believed true, and selects one of them (which is called *main plan*);

3. it builds the intended means, containing the main plan and the other alternative practical plans.

Each new intended means is put within a stack according to the following criteria: if the event e refers to an intention to pursue a goal, it is put on top of the stack containing the intended means that has generated the commitment to that intention; otherwise, a new stack is created with the new intended means. Thus, every intended means stack can contain several nested intended means, each able to handle a given event, possibly through several alternative plans. For each stack, the main plan of the topmost intended means is executed. Meanwhile, both success and cancel conditions of the plan are checked, in order to stop the execution (either with success or failure) before its normal completion.

Moreover all intended means stacks are *concurrently* executed, so that each PRACTIONIST agent can perform several non-inconsistent activities in parallel. Indeed, every PRACTIONIST agent will never perform activities or pursue goals incompatible with one another, at least according to its beliefs and goal model.

In order to be able to recover from *plan failures* and try other means to achieve an intention, if an executing main plan fails or is no longer appropriate to achieve the intention, then the agent selects one of applicable *alternative* plans within the same intended means and executes it.

If none of the alternative plans is able to successfully pursue the goal g, the agent takes into consideration the goals that *entail* g. Thus the agent selects one of them and considers it as a desire, processing it in the way described above, from deliberation to means-ends reasoning. If there is no plan to pursue alternative goals, the achievement of the intention has failed, as the agent has not other ways to pursue its intention. Thus, according to agents beliefs, the goal was *possible*, but the agent was no able to pursue it (i.e. there are no plans).

It should be noted that the PRACTIONIST framework provides agents with the ability to dynamically build plans to pursue a given goal, as soon as no plan of the library is able to pursue it. These planning capabilities are based on a backward search algorithm in the states space implemented in Prolog. Thus, the issue of pursuing a goal can be viewed as a planning problem where the initial state is represented by the agent's beliefs and the domain is represented by available actions. The planning component produces a sequence of actions to be performed in order to satisfy the goal. These actions will be part of the body of a dynamically-generated plan, which temporarily becomes an element of the agent's plan library. This plan may be composed by a set of either abstract or concrete actions, according to whether they have some inputs specified or not. In abstract actions, inputs will be initialised with some outputs of previous actions.

As soon as each action is performed, its post-conditions are applied to update the agent's beliefs. The actual effects of the overall plan should be the satisfaction of the initial goal (the ends), as the plan is the right means to pursue it. The plan can fail when action preconditions are not satisfied due to some unexpected changes of beliefs or the execution of some action fails.

It should be noted that since the planning component could be time-consuming, the developer can disable it. Moreover, the developer can set the maximum number of actions for dynamically-generated plans.

3 Programming PRACTIONIST agents

To develop applications with PRACTIONIST, the programmer has to create the class container of all their characteristics and components. To do this he simply has to:

- create a new class (e.g. `PlayerAgent`) extending the class `AbstractAgent` of the package `org.practionist.core`;
- create a constructor with an array of objects as argument and call that of the super class in it; an example of agent construction is shown in Code 3.1. This constructor allows to specify all necessary arguments to start the agent; they will be in the form of string if it is started from the command prompt or of specific java classes if it is executed by another java application. It is worth noting that it is possible to retrieve these arguments by invoking the method `getArgument()` inherited from the `AbstractAgent`;
- implement the `initialize()` method and declare the agent's plan library, belief set, perceptors, effectors, goal model and each potential goal.

In the following sections it will be explained how to declare such components.

At design time it is possible to choose between two classes of agent (see Figure 5): one (i.e. the `AbstractAgent`) is endowed with a reasoning engine on beliefs fully implemented in Java (i.e. tuProlog), the other (i.e. the `AbstractJPLAgent`) is endowed with the underlying reasoning engine of SWI-Prolog.

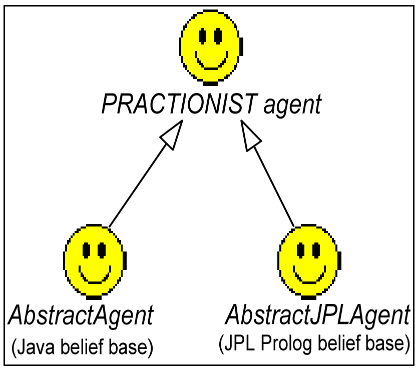

Figure 5 - Types of agent defined in the PRACTIONIST framework.

The choice may depend, for example, on the system architecture and the resources available: as it is implemented in Java, the first option (i.e. the `AbstractAgent`) inherits all advantages of using Java as programming language, such as developing platform-independent applications. Apart from this, the choice does not affect the process of implementation of its components.

Code 3.1 Implementation of a new PRACTIONIST agent

```
package tileworld.player.advancedPlayer;

import org.practionist.belief.*;
import org.practionist.core.*;
...
import org.practionist.goal.*;
...

public class PlayerAgent extends AbstractAgent
{
      public PlayerAgent(Object[] args)
      {
            super(args);
      }

      protected void initialize()
      {
            /** Put here the code to initialize the agent, in terms of initial
             *  belief set, plan library, perceptors, effectors, initial goals.
             */
      }
}
```

3.1 Extending perceptual capabilities

Each PRACTIONIST agent has a perceptual system that allows it to collect the stimuli coming from the environment where it lives. This system is composed of a set of perceptors in charge of detecting the stimuli coming from the environment and turning them into perceptions: in its standard configuration, PRACTIONIST agent is already endowed with two perceptors able to detect messages (i.e. `jade.lang.acl.ACLMessage` objects) and java objects (i.e. `java.lang.Object`). They are respectively `ACLMessagePerceptor` contained in the `org.practionist.perception.message` package and `ObjectPerceptor` contained in the `org.practionist.perception.object` package.

The agent's set of perceptors can be enhanced simply by implementing new perceptors and registering them in `initialize()` method of the agent as shown in Code 3.2.

A perceptor can be defined either by realizing the `org.practionist.perception.Perceptor` interface or specializing the abstract perceptor `org.practionist.perception.AbstractPerceptor` (in this case some of the methods declared in the former are implemented by default). In Figure 6 the structure of the above-mentioned classes and interfaces is shown.

Code 3.2 Registration of a new perceptor

```
protected void initialize()
{
        addPerceptor(new HoleBirthPerceptor());
        ...
}
```

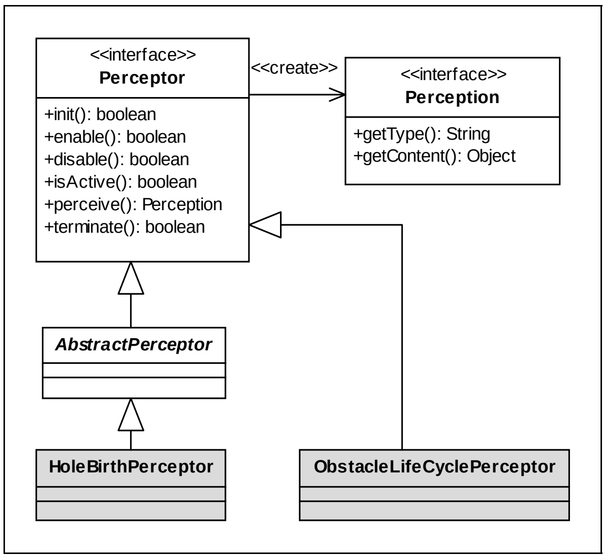

Figure 6 - Example of definition of a preceptor.

The `Perceptor` interface exposes the following methods:

- `init` – to initialize the perceptor;
- `enable` – to trigger the perceptor to detect stimuli;
- `disable` – to deactivate the perceptor to detect stimuli;
- `isActive` – to find out if the perceptor is enabled;
- `perceive` – to find out if there is a new stimulus to catch and produce the related perception;
- `terminate` – to find out if the perceptor can be terminated.

In particular, the `perceive()` method is invoked automatically and cyclically by the agent, while all the other methods can be used by developers at design time to be invoked during normal activities of the agent (i.e. plans). In Code 3.3 an example of perceptor is shown.

Code 3.3 Implementation of a perceptor

```
public class TilePerceptor implements Perceptor
{
	private boolean isActive = false;
	private Grid grid = null;
	public TilePerceptor(Grid grid)
	{
		this.grid = grid;
	}
	public Perception perceive()
	{
		String position = grid.lookForTile();
		if (position != null) return new TilePerception(position);
		return null;
	}
	public boolean disable()
	{
		isActive = false;
		return true;
	}
	public boolean enable()
	{
		isActive = true;
		return true;
	}
	public boolean init()
	{
		return true;
	}
	public boolean isActive()
	{
		return isActive;
	}
	public boolean terminate()
	{
		return true;
	}
}
```

It is also necessary to define the structure of the perception a perceptor will return through the `perceive()` method.

A perception is defined by realizing the `org.practionist.perception.Perception` interface (see Figure 6) which exposes the following methods:

- `getType` – to return the perception type;

- `getContent` – to return the object containing information is based that perception;

In Code 3.4 the definition of a perception is shown.

Code 3.4 Implementation of a perception

```
public class TilePerception implements Perception
{
	public static final String TYPE = "TILE-POSITION";
	public static final String POSITION_NORTH = "NORTH";
	public static final String POSITION_SOUTH = "SOUTH";
	public static final String POSITION_EAST = "EAST";
	public static final String POSITION_WEST = "WEST";
	private String position = "";
	public TilePerception(String position)
	{
		this.position = position;
	}
	public Object getContent()
	{
		return position;
	}
	public String getType()
	{
		return TYPE;
	}
}
```

3.2 Implementing effecting capabilities

Each PRACTIONIST agent has to be endowed with a system of actuators that allows it to deal the environment where it lives. This system is composed of a set of effectors in charge of executing action.

This section describes how to define and use actions and effectors (actuators) in PRACTIONIST.

Actions are components that allow an agent to interact with the environment where it live; they are described by the following elements:

1. *inputs* – they are the objects the action acts over;
2. *output* – they are some kind of direct responses received from the environment;
3. *preconditions* – they are the state of affairs that must be believed true before executing the action;
4. *effects* – (for both successfully and failing action execution), which are the state of affairs that will be believed true or false after executing the action (as long as preconditions are satisfied).

To create an action the programmer has to implement the `BDIAction` interface defined in the `org.practionist.action` package. This interface exposes the following methods:

- `applicable -` to find out if preconditions are satisfied;
- `succeeded -` to find out if the action was completed with success;
- `handleSuccess -` to update the belief set in case of execution with success of the action;

- `handleFailure` - to update the belief set in case of execution with failure of the action;
- `getActor` - to obtain the agent's identification (i.e. `jade.core.AID`) executing the action;
- `getType` - to obtain the action type as a string.

In Code 3.5 the definition of an action is shown.

Code 3.5 Implementation of an action

```
public class PickUpAction implements BDIAction
{
	private boolean succeeded = false;
	public final static String TYPE = "PICKUP";

	/* The action is applicable if the agent is "ready". In order to figure out
	if this state is "ready" a query to the belief set is run.*/
	public boolean applicable(BeliefSet bs)
	{
		AbsPredicate p = AbsPredicateFactory.create("state(value: ready)");
		return bs.bel(p);
	}

	/* If the action is successful, then the agent has the tile; this information
	is maintained through an assertion into the belief base */
	public void handleSuccess(BeliefBase bb)
	{
		bb.add(AbsPredicateFactory.create("hold(obj: tile)"));
	}

	public boolean succeeded()
	{
		return this.succeeded;
	}
	public void handleFailure(BeliefBase bb)
	{
		/* Nothing to do */
	}
	public AID getActor()
	{
		return new AID("PlayerAgent", AID.ISLOCALNAME);
	}
	public String getType()
	{
		return this.TYPE;
	}
	public void setSucceeded(boolean succeeded)
	{
		this.succeeded = succeeded;
	}
}
```

As already said, in PRACTIONIST effectors are used to manage the execution of actions. Indeed, whenever an agent must perform an action, it will select, among its effectors, the ones able to manage it and make one of them execute it.

To create an effector, the programmer has to create a class that implements the `Effector` interface contained in the package `org.practionist.action.` This interface exposes the following methods:

- `triggered` - to find out if the effector is able to manage an action;
- `perform` - to manage the execution of the action; it is worth noting that this method returns the same action it takes as parameter, modified if needed.

Code 3.6 Implementation of an effector

```
public class Taker implements Effector
{
	public Action perform(Action action)
	{
		...
	}

	public boolean triggered(Action action)
	{
		return (action.getType().equals(PickUpAction.TYPE));
	}
}
```

After their creation, it is necessary register the new perceptors in the `initialize()` method of the agent as shown in Code 3.7.

Code 3.7 Registration of an effector

```
protected void initialize()
{
	addEffector(new Taker());
	...
}
```

3.3 Defining beliefs

The PRACTIONIST framework adopts the common approach of modelling agents' beliefs by the doxastic modal logic. Thus, beliefs are expressed through the modal operator $Bel(\alpha, \varphi)$, whose arguments are the agent α (the believer) and what it believes (φ, the fact).

Each fact φ may be believed true or false by a PRACTIONIST agent α, i.e.:

- $Bel(\alpha, \varphi)$ - the agent α believes that φ is true;
- $Bel(\alpha, \neg\varphi)$ - the agent α believes that φ is false;

Moreover, in order to assert that the agent α does not have any belief about φ, we defined the following operator:

- $Ubif(\alpha, \varphi) = \neg Bel(\alpha, \varphi) \wedge \neg Bel(\alpha, \neg\varphi)$

Each fact can either be a closed formula of the classical modal logic or a belief of any agent (e.g. $Bel(\alpha,\ Bel(\beta, \varphi))$ or $Bel(\alpha,\ Ubif(\beta, \varphi))$). In other words, an agent may believe something (e.g. 'it is possible that it is raining in Rome'), or have nested beliefs, that is it may believe that some agent (even itself) believes something (e.g. 'the agent Jim believes that it is raining in Rome').
Finally, in PRACTIONIST it is possible to link an agent's beliefs to others' beliefs or other elements, obtaining new entailed beliefs, through the belief formulas (BFs).
An example of belief formula follows:

$$Bel(tom, Bel(john, raining)) \wedge Bel(tom, trust(who : john)) \Rightarrow Bel(tom, raining)$$

Therefore BFs define relationships among two or more beliefs, allowing to infer new beliefs not explicitly asserted.

In PRACTIONIST the agent's initial belief base is expressed in a Prolog-like language. In Table 3.1 all keywords, useful to define such a belief base, are listed. Therefore, in any moment each PRACTIONIST agent's belief set is composed of the beliefs that have been both directly asserted and inferred by means of BFs and the other built-in theorems (i.e. axioms of the doxastic logic, which are already implemented in PRACTIONIST).

Table 3.1 Belief language: keywords

Keyword	Description
self	It is used to specify that the believer is the agent itself. • `bel(self, <belief>)` • `bel(self, bel(<agent>, bel(self, <fact>)))`
bel_ne	It is used to check if two terms, expressed by either prolog or a belief, are not equals, on the base of the semantics of the ”\=” prolog operator. • `bel_ne(<argument1>, <argument2>)`
bel_eq	It is used to check if two terms, expressed by either prolog or a belief, are equals, on the base of the semantics of the ”==” prolog operator. • `bel_eq(<argument1>, <argument2>)`
bel_gt	It is used to check if argument1 is greater than argument2. • `bel_gt(<argument1>, <argument2>)`
bel_lt	It is used to check if argument1 is lower than argument2. • `bel_lt(<argument1>, < argument2>)`
bel	It is used to declare a belief according to the semantics of the modal operator Bel. • `bel(<believerAgent>, <fact>)` • `bel(<believerAgent >, bel(<believerAgent2>, <fact>))`
ubif	It is used to declare a belief according to the semantics of the modal operator Ubif. • `ubif(<believerAgent>, <fact>)` • `ubif(<believerAgent >, bel(<believerAgent2>, <fact>))`
belief	It is used to declare a belief according to the semantics of the modal operator Bel, where the believer is the agent itself. • `belief(<belief>)`
not	It is used to declare a belief believed false. • `belief(not(<belief>))` • `bel(self, not(<belief>))` • `bel(self, bel(<agent>, not(<belief>)))`
and	It is used to express a set of beliefs collected by means of the “and” logic operator. • `<belief1> and <belief2> and .. and <beliefN>`
or	It is used to express a set of beliefs collected by means of the “or” logic operator. • `<belief1> or <belief2> or .. or <beliefN>`
<=	It is used to define a rule formula on the base of the semantics of the ”:-” prolog operator. • `<head> <= <body>` • `bel(self, <belief>) <= <belief1> or <belief2> and not(<belief3>)`
believes_other	It is used to define a new prolog formula. • `believes_other(self, <prolog formula>, _) :- <prolog code>` • `believes_other(self, check(X, Y), _) :- X == 1 ; Y == 2` So, you can use that formula to define new beliefs or rule formulas: • `bel(<believerAgent>, <belief(X, Y)>) <= <belief1> or check(X, Y)`

To link an agent to its initial belief base, the programmer has to create a file .pl containing initial beliefs and BFs and register this belief base in the `initialize()` method of the agent as shown in Code 3.8

Code 3.8 Registration of the belief base

```
protected void initialize()
{
        addBeliefSet("C:\hello\myBeliefBase.pl");
        ...
}
```

In Code 3.9 an example of declaration of a belief base is shown.

As you can see, in PRACTIONIST a predicate is expressed in the following way:

`predicate-name(r1: v1, … , rn: vn)`

where:

`predicate-name` is the name of the predicate and each slot has a role (`r1, …, rn`) and a value (`v1, …, vn`) and may be retrieved through its role instead of its position.

Code 3.9 Example of declaration of a belief base

```
belief(table(name: table1)).
belief(table(name: table2)).
belief(table(name: table3)).
belief(on(over: block1, under: table1)).
belief(on(over: block3, under: block1)).
belief(on(over: block5, under: block3)).
belief(on(over: block4, under: block2)).
belief(on(over: block2, under: table2)).
belief(clear(obj: block4).
belief(clear(obj: block5).
belief(clear(obj: table3).
belief(fixed(obj: table3)).

belief(goal(on: ['table3', 'block1', 'block2', 'block3', 'block4', block5'])).

bel(self, over(up: B1, down: B2)) <= on(over: B1, under: B2).

bel(self, over(up: B1, down: B2)) <= on(over: B1, under: B3) and
                                     over(up: B3, down: B2).

bel(self, what_block(toMove: Thing, moveTo: Table)) <=
        clear(obj: Table) and
        table(name: Table) and
        bel_ne(Thing, Table) and
        (not(fixed(obj: Table)) or not(bel(self, fixed(obj: Table)))) and
        (not(over(up: Table, down: Thing)) or
         not(bel(self, over(up: Table, down: Thing)))).

bel(self, what_block(toMove: Thing, moveTo: Block)) <=
        clear(obj: Block) and
        bel_ne(Thing, Block) and
        fixing(obj: Block2) and
        (not(fixed(obj: Block)) or not(bel(self, fixed(obj: Block)))) and
        (not(over(up: Block, down: Thing)) or
         not(bel(self, over(up: Block, down: Thing)))) and
        (not(over(up: Block, down: Block2)) or
         not(bel(self, over(up: Block, down: Block2)))).
```

In order to make agents handle the belief base at run time, PRACTIONIST provides two interfaces, both contained in the `org.practionist.belief` package: one is the `BeliefSet` providing all methods to query the belief base; the other is the `BeliefBase` extending the former and providing also the methods to assert and remove beliefs.

Thus, in order to add or remove a belief or simply query the belief base, the programmer has to create the belief and then use the methods of the aforementioned interfaces.

PRACTIONIST also provides two instruments to create beliefs:

- *the factory of predicates:* it is represented by the `AbsPredicateFactory` class contained in the `org.practionist.belief` package which makes available static methods to create a predicate supplying a string (i.e. `java.lang.String)` as parameter;
- *the factory of operators:* it is represented by the `BeliefOperatorFactory` class contained in the `org.practionist.belief` package which makes available static methods to create an operator (Bel or Ubif) supplying a string (i.e. `java.lang.String)`as parameter.

In Code 3.10 an example of updating of belief base is shown.

Code 3.10 How to manage the Belief base

```
/* Create a belief. */
AbsPredicate belief = AbsPredicateFactory.create("ableToOrder(who: self)");

/* Create a belief holding a variable (Var). */
AbsPredicate on = AbsPredicateFactory.create("on(over: block4, under: Var)");

/* Create a belief and assign the value of the clear_obj string to the "object"
role. */
String clear_obj = ...
AbsPredicate clear = AbsPredicateFactory.create("clear(object: %)", clear_obj);

/* Create the belief <this agent believes that <pippo believes that 'block2' is
clear>>. */
Bel bel1 = BeliefOperatorFactory.createBel("pippo", "clear(object: block2)");

/* Create the belief <this agent believes that <pluto believes that <pippo believes
that 'block2' is clear>>>. */
Bel bel2 = BeliefOperatorFactory.createBel("pluto", bel1);

/* Create the belief <this agent believes that <pippo does not any belief about the
fact that the table is clear>> */
Ubif ubif = BeliefOperatorFactory.createUbif("pippo", "clear(object: table)");
Bel bel3 = new Bel(ubif);

BeliefBase bb = getBeliefBase();
bb.add(on);
bb.remove(clear);
bb.bel(bel1);
bb.bif(bel2);
bb.whatAbout(bel3);
Operator o = retrieveUbif(ubif);
```

In addition, PRACTIONIST also provides an useful tool to inspect and verify a belief structure. More accurately, to verify if a belief contains certain information, the programmer has to:

- create a template with information to verify, by using either `AbsPredicateFactory or Belie-fOperatorFactory;`
- create an instance of the `Unifier` class contained in the package `org.practionist.prolog;`
- use one of the methods provided by the Unifier in order to verify if the template unifies with a belief in the belief base according to the Prolog logic of unification).

It is worth noting that the unification process is based on roles of arguments of predicates without considering their position, this is instead taken into consideration in Prolog.

Some examples of use of unification follow:

Code 3.11 Unification process

```
/* Verifies if the noun of the predicate p is ‘on’ and if both roles ‘over’ and ‘un-
der’ are defined in it. */
Unifier unifier = new Unifier();
AbsPredicate template = AbsPredicateFactory.create(“on(over: X, under: Y)”);
if (unifier.unify(template, p)) ...

/* Verifies if the noun of the predicate p is ‘on’, if both roles ‘over’ and ‘under’
are defined in it, and if the role ‘over’ is equal to ‘1’. */
Unifier unifier = new Unifier();
AbsPredicate template = AbsPredicateFactory.create(“on(over: 1, under: Y)”);
if (unifier.unify(template, p)) ...

/* Verifies if the belief represented by ‘bel’ unifies with the belief
  “the agent ‘pippo’ believes that there is another block on the block ‘block1’”. */
Bel bel = ...
Bel template = BeliefOperatorFactory.createBel("pippo", "on(under: block1)”);
if (unifier.unify(template, bel)) ...
```

3.4 Defining goals

In PRACTIONIST the following two families of goals were defined (see Figure 7): **state goals**, which refer to some states of affairs the agent desires/intends to bring about, or cease, or preserve, or avoid; **perform goals**, which are not related to some world states but to some activities the agent desires/intends to perform.
Moreover, PRACTIONIST provides agents with the capability of managing and reasoning about the following state goals:

- `Achieve,` which represents a goal to bring about a state of affair represented by a predicate;
- `Cease` (as opposed to achieve), which represents a goal to stop a state of affair;
- `Maintain,` which represents a goal to observe some world state and continuously re-establish this state when it does not hold;
- `Avoid` (as opposed to maintain), which represents a goal to observe some world state and continuously prevent it;
- `Query,` which aims to obtain some information on a state of the world, expressed by a predicate. The goal will be met if the agent has any belief about it, that is if such a state is believed true or false by the agent.

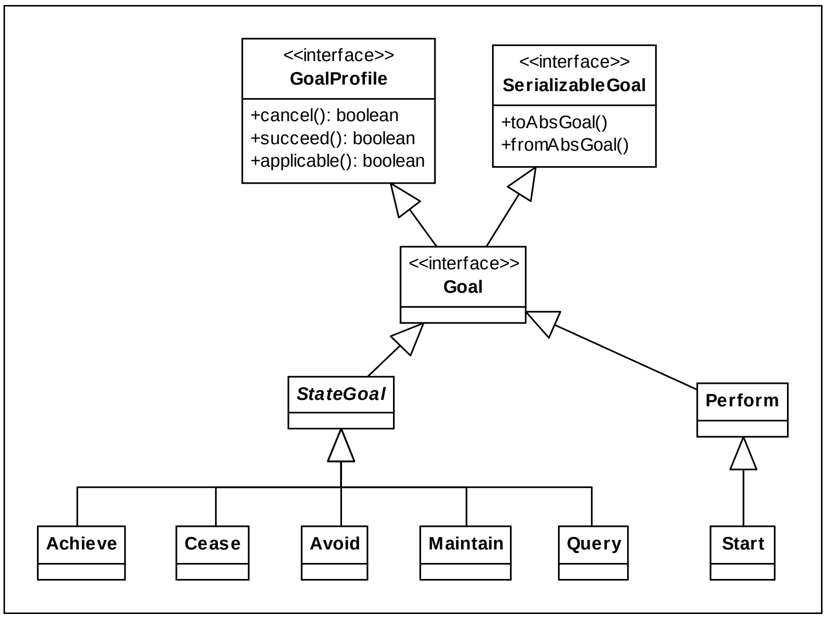

Figure 7 - Goal hierarchy in PRACTIONIST.

The Figure 7 also shows that each goal implements the `GoalProfile` interface to define its behaviour and the `SerializableGoal` interface to make possible to create an instance of such a goal from its abstract representation and vice versa.

More accurately, the `GoalProfile` interface exposes the following methods:

- `applicable -` to find out if the goal is possible according to the agent's beliefs;
- `succeed -` to find out if the goal succeeded according to the agent's beliefs;
- `cancel -` to find out if the goal mustn't be pursued anymore.

As the aforementioned state goals are system goals, their behaviour is standard, that is the methods of the `GoalProfile` and `SerializableGoal` interfaces are already implemented in PRACTIONIST. All the goals agent could pursue have to be registered in the `initialize()` method of the agent as shown in Code 3.12.

Code 3.12 Registration of a goal

```
protected void initialize()
{
 BeliefBase bb= getBeliefBase();
 AbsPredicate on = AbsPredicateFactory.create("on(over: block4, under:  block5)");
 registerGoal(new Achieve(new AbsProposition(on, bb)), "bla bla bla");
 ....
}
```

As already said, a `Perform` goal can be used to make agent desire to execute some activity. PRACTIONIST provides just a perform goal named `Start,` useful to initialize the activities of the agent. You can also specify in its constructor an object of type `jade.content.abs.AbsConcept:` this object identifies a concept used for instance, to define some properties concerning the activities to execute.

Besides, PRACTIONIST allows to customize goals. To implement such goals, the programmer has to create a new class implementing the `Goal` interface. An example of user goal is shown in Code 3.13

Code 3.13 Creating a user goal

```
public class MyGoal implements Goal
{
	private String id;
	private boolean succeed = false;

	public MyGoal(String id)
	{
		this.id = id;
	}

	public boolean applicable()
	{
		return true;
	}

	public boolean succeed()
	{
		return this.succeed;
	}

	public boolean cancel()
	{
		return false;
	}

	public boolean setSucceed()
	{
		this.succeed = true;
	}

	public AbsGoal toAbsGoal()
	{
		AbsGoal abs = new AbsGoal("myGoal");
		abs.set("id", this.id);
		return abs;
	}

	public Goal fromAbsGoal(AbsGoal absGoal)
	{
		if (absGoal.getTypeName.equals("myGoal") &&
		    absGoal.getString("id") != null)
		       return new MyGoal(absGoal.getString("id"));
		return null;
	}
}
```

All the user goals have to be registered in the `initialize()` method of the agent as well. An example is shown in Code 3.14.

Code 3.14 Registration of a user goal

```
protected void initialize()
{
	registerGoal(new MyGoal(), "This goal is to …..");
	....
}
```

3.4.1 Relationships among goals

In addition to the implementation of goal with their properties, the PRACTIONIST framework also supports the explicit representation of relations among goals. This provides the ability to reason about them, in terms of believing if goals are impossible, achieved, incompatible with other goals, and so forth.
Such relations among goals are as follows:

- a goal G_1 **is inconsistent with** a goal G_2 if and only if when G_1 succeeds, then G_2 fails; (it is worth to noting that when two goals are inconsistent with each other, we can also specify that one is preferred to the other.)
- a goal G_1 **entails** a goal G_2 (or equivalently G_2 is entailed by G_1) if and only if when G_1 succeeds, then also G_2 succeeds;
- a goal G_1 **is a precondition of** a goal G_2 if and only if G_1 must succeed in order to be possible to pursue G_2;
- a goal G_1 **depends on** a goal G_2 if G_2 is precondition of G_1 and G_2 must be successful while pursuing G_1.

The class diagram in Figure 8 shows the structure of relations among goals in the PRACTIONIST framework.

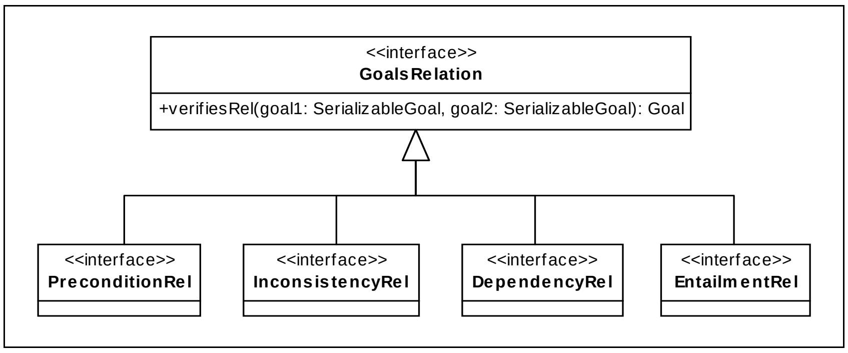

Figure 8 - Class diagram of the relations among goals in PRACTIONIST.

To implement one of the aforementioned relations, the programmer has to create a class that implements the interface corresponding to that relation (see Code 3.15). Such interfaces are in the `org.practionist.goal.model` package. In the example a precondition relation is implemented between two goals: intuitively the filling of a hole can be made only after locating it, so the `FillHole` goal has the `FindHole` goal as precondition.

Code 3.15 Creation of a relation between two goals

```
public class PreconditionFillHoleFindHole implements PreconditionRel
{
	public Goal verifiesRel(SerializableGoal goal1, SerializableGoal goal2)
	{
		if ((goal1 instanceof FillHole) && (goal2 instanceof FindHole))
			return new FindHole();
		return null;
	}
}
```

In addition to the registration of the goals that each agent could try to pursue during his life cycle, the relations among such goals must be registered, so that the agent, more accurately the agent's Goal Model, is able to maintain and verify all the relations among goals.
Some of the above features affect the design activity, while others are exploited by the agent in a transparent way during its life cycle. The goal registration features are directly used by agent developers, who define goals and relations for each agent before inserting them into the goal model during the agent initialization phase (see Code 3.16).

Code 3.16 Registration of relationships among goals

```
protected void initialize()
{
        registerGoal(new ScorePoints(), "This goal is to score agent points");
        registerGoal(new FindTile(), "This goal is to find a tile");

        registerRelation(new EntScorePointsFillHole(), "");
        registerRelation(new PreconditionFillHoleFindHole(), "");
        ....
}
```

3.5 Defining new plans

In the PRACTIONIST framework plans represent an important container where defining the actual behaviours of agents. Each agent may own a declared set of plans (the **plan library**), which specify the activities the agent should undertake in order to achieve its intentions, or handle incoming perceptions, or react to changes of its beliefs.

Several information about plans can be specified (the complete list of such slots is reported in Table 3.2), in order to provide the agents with the capability to dynamically behave when selecting and executing plans. Thus, a plan represents a possible recipe to manage the trigger event, which in turn may be related to a goal, an external event, or an event which notifies a change of the belief set. How to actually handle a certain event is reported within the body, which is an activity that can contain a set of acts, such as desiring to pursue some goal, adding or removing beliefs, sending ACL messages, doing an action and so forth (all the acts will examined in detail later in the document).

Trigger Event	The event (or the set of events) the plan is supposed to handle. It represents the intent of the plan. Whenever trigger event matches the selected event plan is defined *practical*.
Context	A modal logic formula that, when believed true by the agent, makes the plan *applicable*, so that the agent can select it.
Success condition	When the agent believes that this condition holds, the plan ends with success, regardless its execution state.
Cancel condition	When the agent believes that this condition holds, the plan ends with failure, regardless its execution state.
Body	Set of *acts* that are performed during the execution of the plan. The body defines the actual behaviour of the plan.
Invariant	Condition that must remain true during the execution of the plan. As soon as it becomes false (at least according to the agent's point of view), the agent will try to restore it.
Belief updates in case of success	Effects of this plan, in terms of belief updates in case of plan success.
Belief updates in case of failure	Effects of this plan, in terms of belief updates in case of plan failure.

Table 3.2 - Plan structure

After modelling plans it is necessary to register them in the plan library, in order to make them available of the agent; the registration of a plan may be done during the phase of initialization of the agent (by the addPlan method of the agent class) or during the execution of a plan (by the addPlan method of the plan class). Anyway, the programmer has to specify the following properties:

- its class (mandatory);
- an instance of the `PlanDescription` class (contained in the `org.practionist.plan` package) where the elements describing the plan were defined (optional);
- the arguments of the plan constructor (optional);
- a string corresponding to an unambiguous name (mandatory);

In Code 3.17 some examples of plan registration are shown.

Code 3.17 Registration of a plan

```
Object[] args = new Object[] { /*add here arguments*/ };
PlanDescription pd = new ClearBlockPlanDescription();

addPlan(TopLevelPlan.class, "topLevel");
addPlan(RegisterWithManagerPlan.class, args, "registerWithManager");
addPlan(DeregisterFromManagerPlan.class, args, "deregisterFromManager");
addPlan(ClearBlockPlan.class, pd, args, "clearBlock");
....
```

It is also possible to remove a plan (see Code 3.18) by using the `removePlan` method of the plan class passing the plan name as argument.

Code 3.18 Removal of a plan

```
removePlan("topLevel");
removePlan("registerWithManager");
removePlan("deregisterFromManager");
removePlan("clearBlock");
....
```

The class diagram in Figure 9 shows the hierarchy of plans defined in PRACTIONIST:

- `Plan` is the abstract class at the base of the hierarchy of plans defined in the framework and can be used to define any type of behaviour.
- `GoalPlan` can be used to define the behaviour of the agent when it needs to pursue a goal.
- `PerceptionPlan` can be used to define the behaviour of the agent when it needs to react to a perception.
- `MessageReceivingPlan` can be used to define the behaviour of the agent to handle an incoming message.
- `ObjectReceivingPlan` can be used to define the behaviour of the agent to manage an object as input.
- `BeliefUpdatePlan` can be used to define the behaviour of the agent when the value of truth on one of its beliefs change and assumes a specific value.

It is worth noting that such a hierarchy can be enriched by the developer, according to his specific needs.

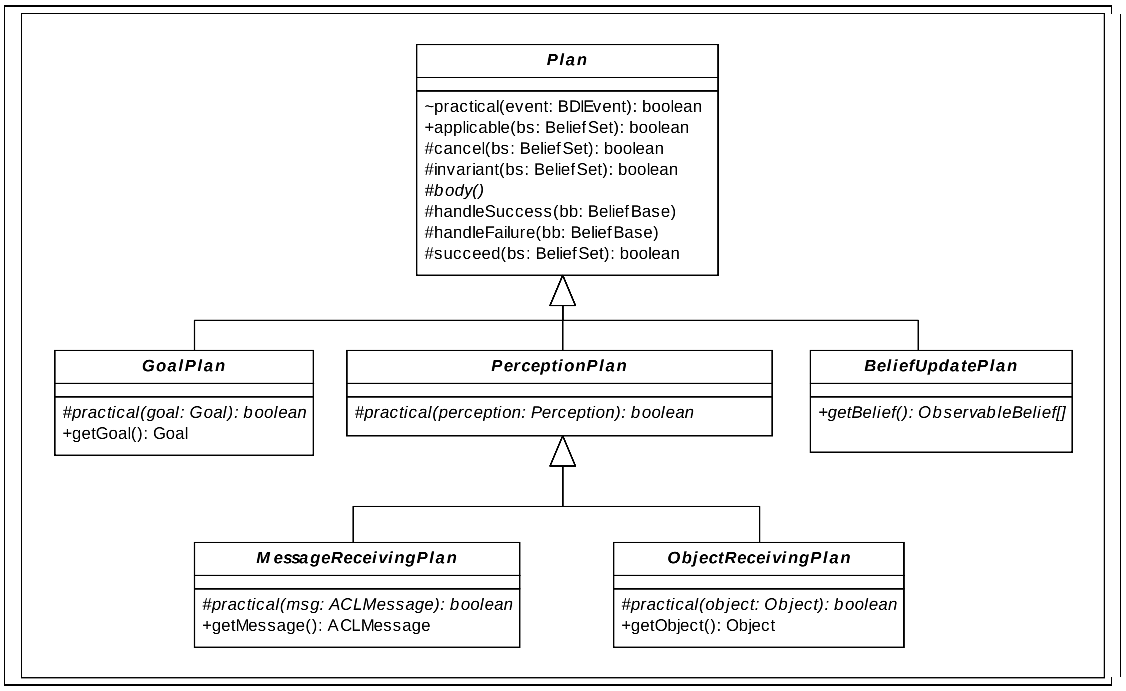
Plan
~practical(event: BDIEvent): boolean
+applicable(bs: BeliefSet): boolean
#cancel(bs: BeliefSet): boolean
#invariant(bs: BeliefSet): boolean
#body()
#handleSuccess(bb: BeliefBase)
#handleFailure(bb: BeliefBase)
#succeed(bs: BeliefSet): boolean
GoalPlan
#practical(goal: Goal): boolean
+getGoal(): Goal
PerceptionPlan
#practical(perception: Perception): boolean
BeliefUpdatePlan
+getBelief(): ObservableBelief[]
MessageReceivingPlan
#practical(msg: ACLMessage): boolean
+getMessage(): ACLMessage
ObjectReceivingPlan
#practical(object: Object): boolean
+getObject(): Object

Figure 9 - Hierarchy of plans in PRACTIONIST.

3.5.1 Abstract class `Plan`

The abstract `org.practionist.plan.Plan` class has the following methods:

- `practical`: it is a package method and defines when the plan is suitable to manage something (i.e. if the event to be managed corresponds to the *Trigger Event* of the plan then this plan turns out to be *practical*). The plan is not practical by default.
- `applicable`: this method may be redefined by the programmer to define when the plan is *applicable* (i.e. if the formula defining the context (*Context*) is believed as true by the agent then this plan turns out to be applicable). In the case that this method is not redefined, the plan is always applicable, by default.
- `cancel`: this method may be redefined by the programmer to define when the plan has to be stopped with failure (i.e. if the formula defined in this method is believed as true by the agent then this plan ends with failure). In the case that this method is not redefined, the plan will never be cancelled, by default.
- `invariant`: this method may be redefined by the programmer to define the condition (*Invariant*) to hold during the execution of the body, (i.e. if the formula defined in this method is believed as true by the agent then the execution of the body will continue). In the case that this method is not redefined, this method always returns true by default.
- `succeed`: this method may be redefined by the programmer to define when the plan ends with success (i.e. if the formula defined in this method is believed as true by the agent then this plan ends with success). In the case that this method is not redefined, the plan will never succeed, by default.
- `handleSuccess`: this method may be redefined by the programmer to update the agent's beliefs in case of success. In the case that this method is not redefined, there is no belief updating.
- `handleFailure`: this method may be redefined by the programmer to update the agent's beliefs in case of failure. In the case that this method is not redefined, there is no belief updating.

To create a new plan, the programmer has to extend the abstract class `Plan` and customize it, by implementing the aforementioned methods, in addition to the body (to define its actual behaviour). But such plan will be never practical: the only way to make it practical is to define this condition in the plan description. To be more precise, an instance of the `PlanDescription` class may be created to declare the elements of a plan, by passing the plan type in the constructor (it is necessary to use one of the three constants defined in the `Plan` class): the methods not redefined will be retrieved in the plan description. Let us suppose that we want to describe a plan able to order blocks according to the order specified in a message sent by another agent; besides let us suppose that it is possible to execute such a plan until the agent believes that he is able to order blocks. In Code 3.19 the corresponding plan is set by means of plan description.

Code 3.19 Example of plan description

```
protected void initialize()
{
 ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
 msg.setOntology(BlockWorldVocabulary.ONTOLOGY_NAME);

 PlanDescription pd = new PlanDescription(Plan.PERCEPTION_PLAN_TYPE);
 pd.setPerception(new ACLMessagePerception(msg));
 pd.addSuccessBeliefAdd(AbsPredicateFactory.create("happy(who: self)"));
 pd.addSuccessBeliefAdd(AbsPredicateFactory.create("ordered(what: blocks)"));
 pd.addFailureBeliefAdd(AbsPredicateFactory.create("ableToOrder(who: self)"));
 pd.setInvariantCondition(AbsPredicateFactory.create("ableToOrder(who: self)"));

 addPlan(TopLevelPlan.class, pd, null, "TopLevelPlan");
 ...
}
```

3.5.2 Plans to react to and handle perceptions: `PerceptionPlan`

The `PerceptionPlan` class can be used to model a plan able to manage a perception detected by one of the perceptors agent has been equipped with. It is contained in the `org.practionist.plan` package.

To create such kind of plan, the programmer has to:

- create a class extending `PerceptionPlan;`
- redefine the `practical` method;
- redefine the `body` method, where the actual activity of the plan is specified;
- redefine all the other methods, if necessary.

3.5.3 Plans to handle received messages: `MessageReceivingPlan`

The `MessageReceivingPlan` class can be used to model a plan able to manage an incoming message. It is contained in the `org.practionist.plan.library` package.

To create such kind of plan, the programmer has to:

- create a class extending `MessageReceivingPlan`;
- redefine the `practical` method;
- redefine the `body` method, where the actual activity of the plan is specified;
- redefine all the other methods, if necessary.

In Code 3.20 an example of implementation of `MessageReceivingPlan` is shown.

Code 3.20 Implementing a MessageReceivingPlan

```
public class ReceivingMsgPlan extends MessageReceivingPlan
{
	protected boolean practical(ACLMessage msg)
	{
		return (msg.getOntology().equals("BlockWorldOntology") &&
                    msg.getPerformative() == ACLMessage.REQUEST);
        }
        ...
}
```

3.5.4 Plans to handle received generic objects: `ObjectReceivingPlan`

The `ObjectReceivingPlan` class can be used to model a plan able to manage an unspecific incoming object (i.e. `java.lang.Object`). It is contained in the `org.practionist.plan.library` package.

To create such kind of plan, the programmer has to:

- create a class extending `ObjectReceivingPlan`;
- redefine the `practical` method;
- redefine the `body` method, where the actual activity of the plan is specified;
- redefine all the other methods, if necessary.

In Code 3.21 an example of implementation of `ObjectReceivingPlan` is shown.

Code 3.21 Implementing an ObjectReceivingPlan

```
public class WSDMEventHandlerPlan extends ObjectReceivingPlan
{
        protected boolean practical(Object object)
        {
                return (object instanceof WSDMEvent);
        }
        ...
}
```

3.5.5 Plans to handle changes in beliefs: `BeliefUpdatePlan`

The `BeliefUpdatePlan` class can be used to model a plan able to manage a *belief base* updating. It is contained in the `org.practionist.plan` package.

To create such kind of plan, the programmer has to:

- create a class extending `BeliefUpdatePlan;`
- implement the `getBelief` method in order to specify the belief updating to manage;
- redefine the `body` method, where the actual activity of the plan is specified;
- redefine all the other methods, if necessary.

In Code 3.22 an example of implementation of `BeliefUpdatePlan` is shown. It is worth noting that in the `getBelief` method the `ObservableBelief` class contained in `org.practionist.belief` was used to specify the belief updating that make the plan practical.

The `ObservableBelief` class has a constructor with two arguments: the belief to observe and the truth value into which it has to switch. Whenever the belief changes and gets such value, the plan turns out to be practical.

The `getBelief` method returns an array of `ObservableBelief` objects and the condition that make the plan practical holds if one of these objects (logical OR) is verified.

In the example of Code 3.22 the plan is practical if one of the following is verified:

- the truth value of the `ordering(item: blocks)` belief changes and switches into true;

- the truth value of the `ableToOrder(who: self)` belief changes (it doesn't matter what value switches into);
- the truth value of the `ready(who: self)` belief changes and switches into false.

Code 3.22 Implementation of a BeliefUpdatePlan

```
public class MyPlan extends BeliefUpdatePlan
{
        public ObservableBelief[] getBelief()
        {
                AbsPredicate p = AbsPredicateFactory.create("ordering(item: blocks)");
                ObservableBelief b1 = new ObservableBelief(p, ObservableBelief.TRUE);
                p = AbsPredicateFactory.create("ableToOrder(who: self)");
                ObservableBelief b2 = new ObservableBelief(p, ObservableBelief.ANY);
                p = AbsPredicateFactory.create("ready(who: self)");
                ObservableBelief b3 = new ObservableBelief(p, ObservableBelief.FALSE);
                return new ObservableBelief[]{b1, b2, b3};
        }
        ...
}
```

3.5.6 Plans to pursue intentions: `GoalPlan`

The `GoalPlan` class can be used to model a plan able to pursue goals. Thus its success condition will correspond to that one of the goal agent would like to pursue: the success condition of the plan will be satisfied when the condition of success of the goal will be satisfied. It is contained in the `org.practionist.plan` package.

To create such kind of plan, the programmer has to:

- create a class extending `GoalPlan`;
- redefine the `practical` method;
- redefine the `body` method, where the actual activity of the plan is specified;
- redefine all the other methods, if necessary.

In an example of implementation of `GoalPlan` is shown.

Code 3.23 Implementing a GoalPlan

```
public class MyGoalPlan extends GoalPlan
{
        protected boolean practical(Goal goal)
        {
                return (goal instanceof MyGoal);
        }
        ...
}
```

The PRACTIONIST framework provides programmers with a set of `GoalPlan`, one for each system goal.

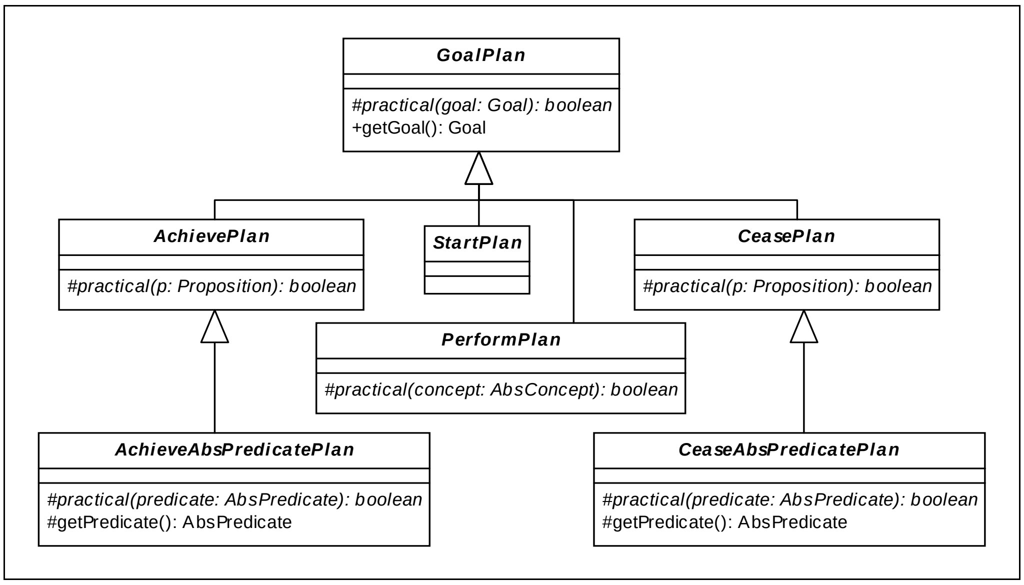
GoalPlan
#practical(goal: Goal): boolean
+getGoal(): Goal
AchievePlan
#practical(p: Proposition): boolean
StartPlan
CeasePlan
#practical(p: Proposition): boolean
PerformPlan
#practical(concept: AbsConcept): boolean
AchieveAbsPredicatePlan
#practical(predicate: AbsPredicate): boolean
#getPredicate(): AbsPredicate
CeaseAbsPredicatePlan
#practical(predicate: AbsPredicate): boolean
#getPredicate(): AbsPredicate

Figure 10 - Hierarchy of GoalPlan in PRACTIONIST.

The Figure 10 shows the hierarchy of such types of `GoalPlan` in PRACTIONIST.

They are contained in the `org.practionist.plan.library` package.

More accurately:

- `StartPlan`: it is an abstract class the programmer may extend to build a plan which manages the agent's intention to pursue a `Start` goal.
- `PerformPlan`: it is an abstract class the programmer may extend to build a plan which manages the agent's intention to pursue a `Perform` goal.
- `AchievePlan`: it is an abstract class the programmer may extend to build a plan which manages the agent's intention to pursue a `Achieve` goal.
- `AchieveAbsPredicatePlan`: it extends the `AchievePlan` class and may be used by the programmer to build a plan which manages the agent's intention to pursue a state represented by the `jade.content.abs.AbsPredicate` abstract descriptor.
- `CeasePlan`: it is an abstract class the programmer may extend to build a plan which manages the agent's intention to pursue a `Cease` goal.
- `CeaseAbsPredicatePlan`: it extends the `CeasePlan` class and may be used by the programmer to build a plan which manages the agent's intention to cease a state represented by the `jade.content.abs.AbsPredicate` abstract descriptor.

In the following paragraphs the aforementioned plan will be analysed in detail.

3.5.6.1 StartPlan

The `StartPlan` class may be extended to create a plan used by the agent to pursue a `Start` goal. A `Start` goal should generally be pursued by the agent as soon as the initialize method of the agent is concluded.

Thus, to implement a `StartPlan`, the programmer has to create a new class extending `org.practionist.plan.library.StartPlan` and implement the `body` method; the other methods are inherited from the `GoalPlan` class and may be redefined to specialize the other elements of the plan.

In Code 3.24 an example of implementation of `StartPlan` is shown.

Code 3.24 Implementing a StartPlan

```
public class StartPlayerPlan extends StartPlan
{
	protected void body() throws PlanExecutionException
	{
		desire(new ScorePoints(), Plan.ASYNC);
		((Start)getGoal()).started();
	}
	...
}
```

3.5.6.2 PerformPlan

The `PerformPlan` class may be extended to create a plan used by the agent to pursue a `Perform` goal.

To implement a `PerformPlan`, the programmer has to create a new class extending `org.practionist.plan.library.PerformPlan` and implement the `practical` and `body` methods. The former defines the plan trigger condition; indeed it expects to be supplied the action to manage by means of the execution of the plan `body` (i.e. `jade.content.abs.AbsConcept`) as parameter.

3.5.6.3 AchievePlan

The `AchievePlan` class may be extended to create a plan used by the agent to pursue an `Achieve` goal.

To implement an `AchievePlan`, the programmer has to create a new class extending `org.practionist.plan.library.AchievePlan` and implement the `practical` and `body` methods. The former defines the plan trigger condition; indeed it expects to be supplied the state to bring about by means of the execution of the plan `body` (i.e. `org.practionist.goal.Proposition`) as parameter.

It is worth noting that in order to figure out if the state indicated into an `Achieve` goal corresponds to that one the plan is able to bring about, it is necessary to use the unification process as shown in Code 3.25 (see also).

Code 3.25 Implementing an AchievePlan

```
public class RegisterWithDFPlan extends AchievePlan
{
	protected boolean practical(Proposition proposition)
	{
		AbsPredicate template = AbsPredicateFactory.
			create("registeredWithDF(agent: VarX, description: VarY)");
		if (proposition instanceof AbsProposition)
		{
			Unifier unifier = new Unifier();
			AbsPredicate p = ((AbsProposition) proposition).getPredicate();
			return (unifier.unify(template, p));
		}
		return false;
	}
	...
}
```

Thus in this example the plan is practical whenever the state to achieve is represented by a proposition having the `registeredWithDF` predicate.

3.5.6.4 AchieveAbsPredicatePlan

The `AchieveAbsPredicatePlan` class may be extended to create a plan used by the agent to pursue a state represented by the `jade.content.abs.AbsPredicate` abstract descriptor.

To implement an `AchieveAbsPredicatePlan`, the programmer has to create a new class extending `org.practionist.plan.library.AchieveAbsPredicatePlan` and implement the `practical` and `body` methods. The former defines the plan trigger condition; indeed it expects to be supplied the state to bring about by means of the execution of the plan `body` (i.e. `jade.content.abs.AbsPredicate`) as parameter.

In Code 3.26 an example of implementation of `AchieveAbsPredicatePlan` is shown. By comparing Code 3.26 and Code 3.25 it is possible to appreciate the advantages of using `AchieveAbsPredicatePlan` instead of `AchievePlan.`

Code 3.26 Implementing an AchieveAbsPredicatePlan

```
public class RegisterWithDFPlan extends AchieveAbsPredicatePlan
{
	protected boolean practical(AbsPredicate predicate)
	{
		Unifier unifier = new Unifier();
		AbsPredicate template = AbsPredicateFactory.
			create("registeredWithDF(agent: VarX, description: VarY)");
		return (unifier.unify(template, predicate));
	}
	...
}
```

3.5.6.5 CeasePlan

The `CeasePlan` class may be extended to create a plan used by the agent to pursue a `Cease` goal.

To implement a `CeasePlan`, the programmer has to create a new class extending `org.practionist.plan.library.CeasePlan` and implement the `practical` and `body` methods. The former defines the plan trigger condition; indeed it expects to be supplied the state to cease by means of the execution of the plan `body` (i.e. `org.practionist.goal.Proposition`) as parameter.

Also in this case, it is worth noting that in order to figure out if the state indicated into a `Cease` goal corresponds to that one the plan is able to cease, it is necessary to use the unification process as shown in Code 3.27.

Code 3.27 Implementing a CeasePlan

```
public class DeregisterFromDFPlan extends CeasePlan
{
	protected boolean practical(Proposition proposition)
	{
		if (proposition instanceof AbsProposition)
		{
			AbsPredicate template = AbsPredicateFactory.create(
				"registeredWithDF(agent: VarX, description: VarY)");
			Unifier unifier = new Unifier();
			AbsPredicate p = ((AbsProposition) proposition).getPredicate();
			return (unifier.unify(template, p));
		}
		return false;
	}
	...
}
```

Thus in this example the plan is practical whenever the state to cease is represented by a proposition having the `registeredWithDF` predicate.

3.5.6.6 CeaseAbsPredicatePlan

The `CeaseAbsPredicatePlan` class may be extended to create a plan used by the agent to cease a state represented by the `jade.content.abs.AbsPredicate` abstract descriptor.

To implement a `CeaseAbsPredicatePlan`, the programmer has to create a new class extending `org.practionist.plan.library.CeaseAbsPredicatePlan` and implement the `practical` and `body` methods. The former defines the plan trigger condition; indeed it expects to be supplied the state to cease by means of the execution of the plan `body` (i.e. `jade.content.abs.AbsPredicate`) as parameter.

In Code 3.28 an example of implementation of `CeaseAbsPredicatePlan` is shown.

Also in this case, by comparing Code 3.28 and Code 3.27 it is possible to appreciate the advantages of using `CeaseAbsPredicatePlan` instead of `CeasePlan.`

Code 3.28 Implementing a CeaseAbsPredicatePlan

```
public class DeregisterFromDFPlan extends CeaseAbsPredicatePlan
{
	protected boolean practical(AbsPredicate predicate)
	{
		Unifier unifier = new Unifier();
		AbsPredicate template = AbsPredicateFactory.
			create("registeredWithDF(agent: VarX, description: VarY)");
		return (unifier.unify(template, predicate));
	}
	...
}
```

3.6 Using acts to implement plan bodies

As already specified, a plan represents a recipe to manage the *Trigger Event*: the real management of the event is charged to the body, which is an activity that can contain a set of *acts*, such as desiring to pursue some goal, adding or removing beliefs, sending ACL messages, doing an action and so forth.

More accurately, to model the behaviour of a plan the programmer has at his disposal the following acts:

- *Querying and updating belief base:* this act is used to assert and remove beliefs or query the agent's *Belief Base.*
- *Sending and receiving messages*: this act is used to provide the agent with the capability to send and receive messages.
- *Waiting for perceptions*: this act is used to block the plan execution until the reception of a certain perception.
- *Desiring to pursue a goal:* this act is used to desire to meet a goal
- *Doing actions:* this act is used to execute an action.

In the following paragraphs the aforementioned acts will be analysed in detail.

3.6.1 Querying and updating belief base

The acts for *Querying and updating belief base* concerns the assert, remove, query and retrieve operations of beliefs. The first two operations involve the updating of the agent's belief base, whereas query and retrieve operations only give information about its beliefs.

To use such acts in the body of a plan, the programmer has to use some methods inherited from the `Plan` abstract class. They are `add` and `remove` which expect to be supplied a belief as parameter, expressed by either a predicate (i.e. `jade.content.abs.AbsPredicate`) or an operator (i.e. `org.practionist.ontology.Operator`). If the parameter is a predicate, the truth value has to be specified (if not specified, the truth value will be true by default); if it is an operator the truth value is into the semantics of the operator itself. Anyway, the belief cannot contain any variable. In Code 3.29 an example of updating of the belief base is shown.

Code 3.29 Examples of updating of beliefs

```
Protected void body() throws PlanExecutionException
{
        /* Creates the abstract descriptor representing the belief, asserts the be-
        lief by using the underlying truth value (true) and finally removes the be-
        lief. */
       AbsPredicate b = AbsPredicateFactory.create("registered(agent: ‘pippo’)");
       add(b);
       remove(b);

        /* Create the abstract descriptor representing the belief, assert the belief
        by using the underlying truth value ‘false’, and finally remove the belief
        (through the operator ‘Not’). */
       b = AbsPredicateFactory.create("registered(agent: ‘pluto’)");
       add(b, false);
       remove(new Not(b));

       /* Create the operator Bel <this agent believes that <agent2 believes...>>,
          assert the operator and finally remove it. */
       Bel bel = new Bel("agent2", b, true);
       add(bel);
       remove(bel));

        /* Create the operator Bel <this agent believes that <agent2 believes false
        ...>>, assert this operator and finally remove it. */
       bel = new Bel("agent2", b, false);
       add(bel);
       remove(bel));

        /* Create the operator Bel <this agent believes that <agent3 believes that
        <agent2 believes that ...>>>, assert the operator and finally remove it. */
       Bel inner_bel = new Bel("agent3", bel);
       add(inner_bel);
       remove(inner_bel));

        /* Create the operator Ubif <this agent does not have any belief about
        <agent2 believes ...>>, assert the operator and finally remove it. */
       Ubif ubif = new Ubif("agent2", b);
       add(ubif);
       remove(ubif);
       ...
}
```

Besides, it is possible to query the belief base in order to check if the agent believes true or false a fact, or if it has or not a belief about a fact, or to obtain information about an agent's belief.

In Code 3.30 an example of query is shown.

Code 3.30 Examples of query on beliefs

```
protected void body() throws PlanExecutionException
{
      /* Create a belief and ask if the agent believes that it is true. */
      AbsPredicate b = AbsPredicateFactory.create("registered(agent: ‘pippo’)");
      bel(b);


      /* Create abelief and ask if the agent believes that it is false. */
      b = AbsPredicateFactory.create("registered(agent: ‘pluto’)");
      bel(b, false);


      /* Create a belief and ask if the agent believes has any belief about it. */
      b = AbsPredicateFactory.create("registered(agent: ‘joe’)");
      bif(b);


      /* Ask what the agent believes about the abovementioned belief. */
      whatAbout(b);


      /* Create an operator Bel and ask if the agent believes that it is true.*/
      Bel bel = new Bel("agent4", b);
      bel(bel);
      ...
}
```

Finally, PRACTIONIST (to be more precise, the `Plan` abstract class) provides methods to retrieve beliefs: given a belief template (where some slots may be variables) it is possible to obtain the first or all beliefs unifying with that template. In Code 3.31 an example of retrieve is shown.

Code 3.31 Examples of retrieve of beliefs

```
protected void body() throws PlanExecutionException
{
      /* Create a template and retrieve the first belief unifying such template. */
      AbsPredicate template = AbsPredicateFactory.create("registered(agent: VAR)");
      AbsPredicate belief = retrieveAbsPredicate(template);

       /* Retrieve the first belief with thruth value ‘false’ unifying such tem-
       plate.*/
      belief = retrieveAbsPredicate(template, false);

      /* Retrieve all the beliefs unifying such template. */
      AbsPredicate[] beliefs = retrieveAllAbsPredicates(template);

      /* Retrieve the first operator Bel unifying such template. */
      Bel bel = new Bel(template);
      Bel b = retrieveBel(bel);

      /* Retrieve all the operators Bel unifying such template. */
      Bel bels = retrieveAllBels(bel);
      ...
}
```

3.6.2 Sending and receiving messages

PRACTIONIST provides *acts* to make agents able to address messages (i.e. `jade.lang.acl.ACLMessage`) to one another and receive messages. All these methods are inherited from the Plan abstract class.

More accurately, they are:

- *send*: to send a message by passing the *ACLMessage* to address as parameter;
- *waitFor*: to lie in wait for an *ACLMessage*, by passing the template of the message to wait for as parameter (as *ACLMessage* or *MessageTemplate*);
- *sendAndReceive*: to send a message and lie in wait for a replay; it is possible to specify the template of the message to wait for (as *ACLMessage* or *MessageTemplate*).

In Code 3.32 an example of send/receive message is shown.

For all these acts, it is also possible to specify the maximum time to be awaited, expressed in milliseconds, and the mode of management of the message: exclusive (`Plan.EXCLUSIVE_HANDLING` constant) or non exclusive (`Plan.NOT_EXCLUSIVE_HANDLING` constant). In the former, the message will be only managed by the plan which waits for it, in the latter the message will be managed also by the plans practical for that message.

Code 3.32 Sending and receiving messages

```
protected void body() throws PlanExecutionException
{
        /* Create an ACLMessage with an ACLMessage.REQUEST performative, ... */
        ACLMessage toSend = new ACLMessage(ACLMessage.REQUEST);
        toSend.setOntology("TileWorldOntology");
        toSend.setConversationId("1234567890");
        ...

        /* Send the message.*/
        send(toSend);

        /* Create a message to be used as a template for receiving of the message.*/
        ACLMessage template = new ACLMessage(ACLMessage.INFORM);
        template.setOntology("TileWorldOntology");
        ACLMessage received = waitFor(template, Plan.EXCLUSIVE_HANDLING, 3000);

        /* Create a MessageTemplate to be used for receiving of the message. */
        MessageTemplate template2 = MessageTemplate.MatchConversationId("012345");
        ACLMessage received2 = waitFor(template2, Plan.NOT_EXCLUSIVE_HANDLING, 2000);

        /* Create a MessageTemplate for receiving a message where the slot identify-
        ing ontology contains the "TileWorldOntology" value or "BlockWorldOntology"
        one. */
        MessageTemplate mt1 = MessageTemplate.MatchOntology("TileWorldOntology");
        MessageTemplate mt2 = MessageTemplate.MatchOntology("BlockWorldOntology");
        MessageTemplate template3 = MessageTemplate.or(mt1, mt2);
        ACLMessage rec3 = sendAndReceive(toSend, template3, Plan.EXCLUSIVE_HANDLING);
        ...
}
```

3.6.3 Waiting for perceptions

To make an agent able to wait for a perception during the execution of a plan, the programmer has to use the `waitFor` method defined in the `org.practionist.plan.Plan` class specifying the `org.practionist.perception.Perception` to wait for. To be more precise, by calling this act, a plan blocks its execution until that perception arrives. Also in this case, it is possible to specify the mode of management (exclusive/non exclusive) .

In Code 3.33 an example of Waiting for perceptions is shown.

Code 3.33 Waiting an event

```
protected void body() throws PlanExecutionException
```

```
{
        /* Create a template to wait an event, and after return the object contained
        in it. */
        Perception toWait = new MyPerception();
        Perception p = waitFor(toWait, Plan.EXCLUSIVE_HANDLING);
        ...
}
```

It is worth noting that it is necessary to implement the equals method of the java Object class in the perception which is expected, as shown in Code 3.34.

Code 3.34 Implementation of a perception

```
public class TilePerception implements Perception
{
        public static final String TYPE = "TILE-POSITION";
        public static final String POSITION_NORTH = "NORTH";
        public static final String POSITION_SOUTH = "SOUTH";
        public static final String POSITION_EAST = "EAST";
        public static final String POSITION_WEST = "WEST";
        private String position = "";

        public TilePerception(String position)
        {
                this.position = position;
        }

        public Object getContent()
        {
                return position;
        }

        public String getType()
        {
                return TYPE;
        }

        public boolean isEqual(Object object)
        {
                if (object instanceof TilePerception)
                {
                        TilePerception p = (TilePerception) object;
                        return getContent().equals(p.getContent());
                }
                return false;
        }
}
```

3.6.4 Desiring to pursue a goal

Another act available to developers is on the desire to pursue a goal. To use this type of act the programmer has to choose one of the following methods (inherited from class `org.practionist.plan.Plan`):

- `desire`: this method can be used to pursue a goal in a synchronous or asynchronous way. To pursue a goal in a synchronous way, the programmer has to pass the goal and the constant `Plan.SYNC` as parameter: in this case plan blocks its execution until the goal succeeds, if the goal fails the plan fails as well. To pursue a goal in a asynchronous way, the programmer has to pass the goal and the constant `Plan.ASYNC:` in this case after calling the desire method, plan continues its execution without taking care of the goal success. If not specified, the desire methods will be executed in a synchronous way by default.
- `desireAny`: this method can be used to pursue at least one of the goals contained in a list. More accurately, the programmer has to specify a java.util.List, containing the goals as argument. Such goals will

simultaneously be desired (in an asynchronous way), until at least one of them succeeds. The success of one of these goals implies the deletion of the remaining goals and the resume of the execution of the plan.

- `desireAll`: this method can be used to pursue all the goals contained in a list. More accurately, the programmer has to specify a java.util.List, containing the goals as argument. Such goals will simultaneously be desired (in an asynchronous way), until everyone succeed. The success of all these goals implies the resume of the execution of the plan. If some goal fails the plan fails as well.

Invoking one of aforementioned methods the PRACTIONIST deliberation process starts (see section 2.1).

In Code 3.35 an example of desiring a goal is shown.

Code 3.35 Desiring a goal

```
protected void body() throws PlanExecutionException
{
	/* Run the desire of a goal in a synchronous way. */
	Goal goal = new MyGoal();
	desire(goal, Plan.SYNC);

	/* Run the desire of a goal in a asynchronous way. */
	Goal goal2 = new MyGoal();
	desire(goal2, Plan.ASYNC);

	/* Create a goal list and pursue them. */
	List allList = new ArrayList();
	allList.add(new Goal1());
	allList.add(new Goal2());
	allList.add(new Goal3());
	desireAll(allList);

	/* Create a goal list and pursue them until one of them will be succeeded. */
	List anyList = new ArrayList();
	anyList.add(new Goal4());
	anyList.add(new Goal5());
	anyList.add(new Goal6());
	desireAny(anyList);
	...
}
```

In the `Plan` abstract class other methods to desire a goal has been provided:

- `achieve`, to desire an `Achieve` goal;
- `cease`, to desire a `Cease` goal;
- `maintain`, to desire a `Maintain` goal;
- `avoid`, to deiste an `Avoid` goal;
- `perform`, to desire a `Perform` goal.

3.6.5 Doing actions

To make an agent able to execute an action (i.e. an object implementing `org.practionist.action.BDIAction`), the programmer has to use the `doAction` method defined in the `org.practionist.plan.Plan` class.

To handle such kind of act, the agent will select the effectors able to execute the action and burden one of them with its execution.

To manage the action expressed by means of the `doAction` act, the PRACTIONIST agent :

1. checks the applicability of the action, and if not it throws an exception;
2. identifies among its effectors those who are able to manage the action, and chooses one of theme to handle the action; the plan is put in a state of waiting until the execution ends or an exception is thrown;
3. checks if the action has been completed successfully or failed: in the first case it executes the `handle-Success` method and returns the same action, otherwise performs the `handleFailure` method and throws an exception. The implementation of these two methods has the purpose of updating the agent's be-lief set according to changes described by the effects of the action.

In Code 3.36 an example of implementation of an action is shown, while Code 3.37 is an example of plan where an action is executed.

Code 3.36 Definition of an action

```
/* Implementing the class which defines the action. */
public class MoveAction implements BDIAction
{
	public final static String TYPE = "MOVE";
	private String thing1 = null, thing2 = null;
	private AbsPredicate prec1=null, prec2=null, prec3=null;
	private AbsPredicate eff1=null, eff2=null;
	private boolean succeeded = false;

	public MoveAction(String thing1, String thing2)
	{
		this.thing1 = thing1;
		this.thing2 = thing2;
		prec1 = AbsPredicateFactory.create("clear(obj: " + thing1 + ")");
		prec2 = AbsPredicateFactory.create("clear(obj: " + thing2 + ")");
		eff1 = AbsPredicateFactory.create("on(under: " + thing2 +", over:
			" + thing1 + ")");
	}

	public boolean applicable(BeliefSet bs)
	{
		AbsPredicate tmp = null;
		tmp = AbsPredicateFactory.create("on(under: Other, over: " + thing1 +
			")");
		prec3 = bs.retrieveAbsPredicate(tmp, true);
		return (bs.bel(prec1) && bs.bel(prec2) && prec3 != null);
	}

	public void handleSuccess(BeliefBase bb)
	{
		String other = prec3.getString("under");
		eff2 = AbsPredicateFactory.create("clear(obj: " + other + ")");
		bb.remove(prec3);
		bb.add(eff2);
		bb.remove(prec2);
		bb.add(eff1);
	}
}
```

Code 3.37 Execution of an action

```
/* Executing an act into the body of a plan. */
protected void body() throws PlanExecutionException
{
	/* Create the action and execute it */
	BDIAction move = new MoveAction("block1","table2");
	doAction(move);
	....
}
```