JADE TUTORIAL

CREATING ONTOLOGIES BY MEANS OF THE BEAN-ONTOLOGY CLASS

USAGE RESTRICTED ACCORDING TO LICENSE AGREEMENT.

last update: 15 April 2010. JADE 4.0

Authors: Paolo Cancedda, Giovanni Caire

TABLE OF CONTENTS

1 OVERVIEW

The `BeanOntology` class is an extension of the `Ontology` class that allows automatically creating all the schemas of a JADE ontology starting from the Java classes representing the ontological elements (ontological classes). The `BeanOntology` is based on the "Convention over Configuration" principle that is:
- If the ontological classes follow the standard JavaBean convention, the related schemas are created automatically with no effort from the developer.
- If developers need/want to deviate from the convention, they can do that by properly annotating the ontological classes as described in section 3.

When creating an ontology using the `BeanOntology` class developers do not need to care or know about all stuff related to ontological schemas (described in the "Tutorial on Content Languages and Ontologies" available in the JADE web site). They just need to create the ontological classes (beans) representing the concepts, agent-actions and predicates relevant to the addressed domain and add them to the ontology class as exemplified below.

```
public class MyOntology extends BeanOntology {
	private static Ontology theInstance = new MyOntology(ONTOLOGY_NAME);

	public static Ontology getInstance() {
		return theInstance;
	}

	private MyOntology(String name) {
		super(name);

		try {
			// Add class C1
			add(C1.class);
			// Add class C2
			add(C2.class);
			// Add all ontological classes included in a package
			add("com.acme.rocket.ontology");
		}
		catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

As shown in the code snippet above, the add() method that allows registering ontological elements to the ontology is available in two versions.
The first one taking a `Class` object as argument adds the specified class plus any super-class implementing `Concept`, `AgentAction` or `Predicate` as well.
The second one taking a package name as parameter adds all classes implementing `Concept`, `Predicate` or `AgentAction` included in the given package.

2 DIFFERENCES WITH RESPECT TO THE TRADITIONAL APPROACH

For developers with experience in creating JADE ontologies, this section highlights the differences between the `BeanOntology` based approach and the traditional approach.
Such differences are

1. As already described ontological classes are added specifying a class or a package name; the BeanOntology automatically creates the necessary schemas and registers them to the ontology.
2. Slots with cardinality greater than one can be implemented using both java collections (List or Set) and jade.util.leap collections (List or Set) as well as arrays.
3. There is no need/chance to specify the ontology `Introspector` as an ontology defined using the BeanOntology approach always uses its own custom introspector
4. An ontology defined using the BeanOntology approach always needs the `BasicOntology` among its ancestors; when no base ontology is explicitly specified, `BasicOntology` is set internally as the base ontology.

3 CUSTOMIZATIONS

In order for developers to be able to use ontological classes that do not comply with the standard JavaBean convention, the following customizations are supported by means of proper annotations.
1) Specifying an ontological type name different from that of the ontological class

```
@Element(name = "T2")
public class T1 … {
      …
}
```

2) Specifying a slot name different from that indicated by the getter/setter methods of the ontological class

```
@Slot(name = "S2")
public T getS1();
public void setS1(T t);
```

3) Specifying that a slot is mandatory (by default a slot is considered optional)

```
@Slot(mandatory = true)
public T getS ();
public void setS(T t);
```

4) Suppressing the slot corresponding to given getter/setter methods

```
@SuppressSlot
public T getS();
public void setS (T t);
```

5) Indicating the type of the elements of an aggregate slot.

```
@AggregateSlot(type = T)
public List getS();
public void setS(List l)
```

It should be noticed that, when using Java collections with generics or arays the element type is automatically managed by the BeanOntology class. So for instance, in case a class has the following getter and setter methods

```
public java.util.list<T> getS();
public void setS(java.util.List<T> l)
```

the element type for the S aggregate slot is automatically set to T

6) Indicating maximum and minimum cardinality of the elements of an aggregate slot

```
@AggregateSlot(cardMin = n1, cardMax = n2)
public List getS();
public void setS(List l)
```

7) Indicating the type of result of an action

```
@Result(type = T)
public class A implements AgentAction {
      …
}
```

8) Indicating the type of the elements when the result of an action is an aggregate

```
@AggregateResult(type = T)
public class A implements AgentAction {
      …
}
```

9) Indicating the maximum and minimum cardinality of the elements when the result of an action is an aggregate

```
@AggregateResult(type = T, cardMin = n1, cardMax = n2)
public class A implements AgentAction {
      …
}
```

4 THE BEANONTOLOGY IN PRACTICE

This section gives step-by-step guidance to developing an ontology based on the BeanOntology class by means of a practical example. This example refers to the music shop scenario described in the “Tutorial on Content Languages and Ontologies” available in the JADE web site. Using the same scenario allows interested readers in appreciating the simplifications introduced by the BeanOntology approach.

The full sources of the ontology are available in packages `examples.content.eco`, `examples.content.mso`, `examples.content.sfo` included in the jade examples distribution.

4.1 Creating ontological java beans

As usual the first step is creating the java classes (beans) that describe the concepts, agent actions and predicates relevant to the addressed domain. Such classes remain the very same of the original example, except (when needed) for the annotations needed to characterize the slots in terms of optionality, cardinality of aggregate fields and possibly other ontological peculiarities, such as slot/concept names.
The Track, CD and Single classes are provided below as an example.

```
public class Track implements Concept {

	private String name;
	private Integer duration;
	private byte[] pcm;

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	@Slot(mandatory = false)
	public Integer getDuration() {
		return duration;
	}

	public void setDuration(Integer duration) {
		this.duration = duration;
	}

	@Slot(mandatory = false)
	public byte[] getPcm() {
		return pcm;
	}

	public void setPcm(byte[] pcm) {
		this.pcm = pcm;
	}
}
```

The `@Slot` annotations mark `duration` and `pcm` as optional slots.

```
public class CD extends Item {

	private String title;
	protected List tracks;

	public String getTitle() {
		return title;
	}
```

```
	public void setTitle(String t) {
		title = t;
	}

	@AggregateSlot(cardMin = 1)
	public List getTracks() {
		return tracks;
	}

	public void setTracks(List l) {
		tracks = l;
	}
}
```

The annotation `@AggregateSlot` here is used to specify that the aggregate slot `tracks` contains at least a `Track` element.

```
public class Single extends CD {

	@AggregateSlot(cardMin = 2, cardMax = 2)
	public List getTracks() {
		return tracks;
	}

	public void setTracks(List l) {
		tracks = l;
	}
}
```

According to the original example, a Single concept must contain exactly 2 tracks. The annotation `@AggregateSlot` defines this constraint.

4.2 Creating the ontology class

Once the ontological java beans have been created it is possible to create the ontology class and this is where the BeanOntology shows its main benefits.
The simplest approach is to gather all the ontological beans in a package and let the BeanOntology find them. The `ECommerceOntology` is created in this way.

```
private ECommerceOntology() throws BeanOntologyException {
	super(ECOMMERCE_ONTOLOGY_NAME);
	//
	add("examples.content.eco.elements");
}
```

Alternatively it is possible to specify each class needed by the ontology as shown in the `MusicShopOntology` below.

```
private MusicShopOntology() throws BeanOntologyException {
	super(ONTOLOGY_NAME, ECommerceOntology.getInstance());

	add(Track.class);
	add(CD.class);
	add(Single.class);
```

```
}
```

In general it is possible to mix the two approaches and, for instance, add to an ontology all classes of a given package plus other classes not belonging to that package.

4.3 Further customizations

The ontology presented is functionally equivalent to the one defined in the Jade ontology tutorial, but the items defined have different names.
By default, the bean ontology builder uses the *lowerCamelCase* naming convention for slots and the class name for elements.
When necessary, the `BeanOntology` may be instructed to use names chosen by the user. For instance, we can modify the `CreditCard` class in order to match the names of the original ontology:

```
@Element(name="CREDITCARD")
public class CreditCard implements Concept {

	…

	@Slot(name = "expirationdate")
	public Date getExpirationDate() {
		return expirationDate;
	}
```

5 HIERARCHYC AND FLAT SCHEMAS

By default the schemes automatically created by the BeanOntology are defined hierarchically. For instance if `ExpertProgrammer` extends `Programmer` that extends `Person` that on its turn implements `Concept`, when the `ExpertProgrammer` class is added to the ontology three schemas are created: one for `Person`, one for `Programmer` having the person schema as super-schema and one for `ExpertProgrammer` having the programmer schema as super-schema.
Both versions of the `add()` method of the `BeanOntology` class are also available in a form that takes a second argument of type `boolean` and that allows creating flat schemas as exemplified below.

```
public class Person implements Concept {

	private String name;
	private int age;

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}
```

```
	public int getAge() {
		return age;
	}

	public void setAge(int age) {
		this.age = age;
	}
}

public class Programmer extends Person {

	private int bugsProducedPerHour;

	public int getBugsProducedPerHour() {
		return bugsPerHour;
	}

	public void setBugsProducedPerHour(int bugsProducedPerHour) {
		this.bugsProducedPerHour = bugsProducedPerHour;
	}
}

public class ExpertProgrammer extends Programmer {

	private int bugsFixedPerHour;

	public int getBugsFixedPerHour() {
		return bugsFixedPerHour;
	}

	public void setBugsFixedPerHour(int bugsFixedPerHour) {
		this.bugsFixedPerHour = bugsFixedPerHour;
	}
}
```

With reference to the above classes is `ExpertProgrammer` is added to a bean-ontology using the form

```
add(ExpertProgrammer.class, false);
```

a single schema is created with no super-schema and containing four slots: `name`, `age`, `bugsProducedPerHour` and `bugsFixedPerHour`.