Dynamic object attributes

In this example, we will illustrate the use of:

- categories
- category predicates
- dynamic predicates

by defining a category that implements a set of predicates for handling dynamic object attributes.

Defining a category

We want to define a set of predicates to handle dynamic object attributes. We need public predicates to set, get, and delete attributes, and a private dynamic predicate to store the attributes values. Let us name these predicates `set_attribute/2` and `get_attribute/2`, for getting and setting an attribute value, `del_attribute/2` and `del_attributes/2`, for deleting attributes, and `attribute_/2`, for storing the attributes values.

We do not want to encapsulate these predicates in an object. Why? Because they are a set of useful, closely related, predicates that may be used by several, unrelated, objects. If defined at an object level, we would be constrained to use inheritance in order to have the predicates available to other objects. Furthermore, this could force us to use multi-inheritance or to have some kind of generic root object containing all kinds of possible useful predicates.

For this kind of situation, Logtalk enables the programmer to encapsulate the predicates in a *category*, so that they can be used in any object. A category is a Logtalk entity, at the same level as objects and protocols. It can contain predicates directives and/or definitions. Category predicates can be imported by any object, without code duplication and without resorting to inheritance.

When defining category predicates, we need to remember that a category can be imported by more than one object. Because of that, the calls to the built-in methods that handle the private dynamic predicate (like `assertz/1` or `retract/1`) must be made using the *message to self* control structure, `::/1`. This way, we ensure that when we call one of the attribute predicates on an object, the object own definition of `attribute_/2` will be used. The predicates definitions are straightforward:

```
:- category(attributes).

    :- public(set_attribute/2).
    :- mode(set_attribute(+nonvar, +nonvar), one).

    :- public(get_attribute/2).
    :- mode(get_attribute(?nonvar, ?nonvar), zero_or_more).

    :- public(del_attribute/2).
    :- mode(del_attribute(?nonvar, ?nonvar), zero_or_more).

    :- public(del_attributes/2).
    :- mode(del_attributes(@term, @term), one).

    :- private(attribute_/2).
    :- mode(attribute_(?nonvar, ?nonvar), zero_or_more).
    :- dynamic(attribute_/2).

    set_attribute(Attribute, Value):-
        ::retractall(attribute_(Attribute, _)),
        ::assertz(attribute_(Attribute, Value)).

    get_attribute(Attribute, Value):-
        ::attribute_(Attribute, Value).

    del_attribute(Attribute, Value):-
        ::retract(attribute_(Attribute, Value)).

    del_attributes(Attribute, Value):-
        ::retractall(attribute_(Attribute, Value)).

:- end_category.
```

We have two new directives, `category/1` and `end_category/0`, that encapsulate the category code. If needed, we can put the predicates directives inside a protocol that will be implemented by the category:

```
:- category(attributes,
    implements(attributes_protocol)).

    ...

:- end_category.
```

Any protocol can be implemented by either an object, a category, or both.

Importing the category

We reuse a category's predicates by importing them into an object:

```
:- object(person,
    imports(attributes)).

    ...

:- end_object.
```

After compiling and loading this object and our category, we can now try queries like:

```
| ?- person::set_attribute(name, paulo).

yes

| ?- person::set_attribute(gender, male).

yes

| ?- person::get_attribute(Attribute, Value).

Attribute = name, Value = paulo ;
Attribute = gender, Value = male ;
no
```

Summary

- Categories are similar to objects: we just write our predicate directives and definitions bracketed by opening and ending category directives.
- An object reuses a category by importing it. The imported predicates behave as if they have been defined in the object itself.
- When do we use a category instead of an object? Whenever we have a set of closely related predicates that we want to reuse in several, unrelated, objects. Categories can be interpreted as object building components.