Categories

Categories provide a way to encapsulate a set of related predicate definitions that do not represent an object and that only make sense when composed with other predicates. Categories may also be used to break a complex object in functional units. A category can be imported by several objects (without code duplication), including objects participating in prototype or class-based hierarchies. This concept of categories shares some ideas with Smalltalk-80 functional categories [Goldberg 83], Flavors mix-ins [Moon 86] (without necessarily implying multi-inheritance) and Objective-C categories [Cox 86]. There are no pre-defined categories in Logtalk.

Defining a new category

We can define a new object in the same way we write Prolog code: by using a text editor. Logtalk source files may contain one or more objects, categories, or protocols. If you prefer to define each entity in its own source file, it is recommended that the file be named after the category. By default, all Logtalk source files use the extension `.lgt` but this is optional and can be set in the configuration files. Compiled source files (by the Logtalk preprocessor) have, by default, a `.pl` extension. Again, this can be set to match the needs of a particular Prolog compiler in the corresponding configuration file. For instance, we may define a category named `documenting` and save it in a `documenting.lgt` source file that will be compiled to a `documenting.pl` Prolog file.

Category names must be atoms. Objects, categories, and protocols share the same name space: we can not have a category with the same name as an object or a protocol.

Category code (directives and predicates) is textually encapsulated by using two Logtalk directives: `category/1-3` and `end_category/0`. The most simple category will be one that is self-contained, not depending on any other Logtalk entity:

```
:- category(Category).
    ...
:- end_category.
```

If a category implements one or more protocols then the opening directive will be:

```
:- category(Category,
    implements(Protocol)).
    ...
:- end_category.
```

A category may be defined as a composition of other categories by writing:

```
:- category(Component,
    imports(Category1, Category2, ...)).
    ...
:- end_category.
```

This feature should only be used when extending a category without breaking its functional cohesion (for example, when a modified version of a category is needed for importing on several unrelated objects). The prefered way of composing several categories is by importing them into an object.

Categories cannot inherit from objects. In addition, categories cannot contain clauses for dynamic predicates. This restriction applies because a category can be imported by several objects and because we cannot use the database handling built-in methods with categories (messages can only be sent to objects). However, categories may contain declarations for dynamic predicates and they can contain predicates which handle dynamic predicates. For example:

```
:- category(attributes).

    :- public(attribute/2).
    :- public(set_attribute/2).
    :- public(del_attribute/2).

    :- private(attribute_/2).
    :- dynamic(attribute_/2).

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

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

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

:- end_category.
```

Each object importing this category will have its own `attribute_/2` private, dynamic predicate. The predicates `attribute/2`, `set_attribute/2`, and `del_attribute/2` always access and modify the dynamic predicate contained in the object receiving the corresponding messages.

Finding defined categories

We can find, by backtracking, all defined categories by using the `current_category/1` Logtalk built-in predicate with an uninstantiated variable:

```
| ?- current_category(Category).
```

This predicate can also be used to test if a category is defined by calling it with a valid category identifier (an atom or a compound term).

Creating a new category in runtime

A category can be dynamically created at runtime by using the `create_category/4` built-in predicate:

```
| ?- create_category(Category, Relations, Directives, Clauses).
```

The first argument, the name of the new category - a Prolog atom - should not match with an existing entity name. The remaining three arguments correspond to the relations described in the opening category directive and to the category code contents (directives and clauses).

For instance, the call:

```
| ?- create_category(ccc,
                     [implements(ppp)],
                     [private(bar/1)],
                     [(foo(X):-bar(X)), bar(1), bar(2)]).
```

is equivalent to compiling and loading the category:

```
:- category(ccc,
    implements(ppp)).

    :- dynamic.

    :- private(bar/1).

    foo(X) :-
        bar(X).

    bar(1).
    bar(2).

:- end_category.
```

If we need to create a lot of (dynamic) categories at runtime, then is best to to define a metaclass or a prototype with a predicate that will call this built-in predicate in order to provide more sophisticated behavior.

Abolishing an existing category

Dynamic categories can be abolished using the `abolish_category/1` built-in predicate:

```
| ?- abolish_category(Category).
```

The argument must be an identifier of a defined dynamic category, otherwise an error will be thrown.

Category directives

Category directives are used to set initialization goals and category properties and to document a category dependencies on other Logtalk entities.

Category initialization

We can define a goal to be executed as soon as a category is (compiled and) loaded to memory with the `initialization/1` directive:

```
:- initialization(Goal).
```

The argument can be any valid Prolog or Logtalk goal, including a message sending call.

Dynamic categories

As usually happens with Prolog code, a category can be either static or dynamic. A category created during the execution of a program is always dynamic. A category defined in a file can be either dynamic or static. Dynamic categories are declared by using the `dynamic/0` directive in the category source code:

```
:- dynamic.
```

The directive must precede any predicate directives or clauses. Please be aware that using dynamic code implies a performance hit when compared to static code. We should only use dynamic categories when these need to be abolished during program execution.

Category dependencies

Besides the relations declared in the category opening directive, the predicate definitions contained in the category may imply other dependencies. This can be documented by using the `calls/1` and the `uses/1` directives.

The `calls/1` directive can be used when a predicate definition sends a message that is declared in a specific protocol:

```
:- calls(Protocol).
```

If a predicate definition sends a message to a specific object, this dependence can be declared with the `uses/1` directive:

```
:- uses(Object).
```

These two directives can be used by the Logtalk runtime to ensure that all needed entities are loaded when running an application.

Category documentation

A category can be documented with arbitrary user-defined information by using the `info/1` directive:

```
:- info(List).
```

See the documenting Logtalk programs session for details.

Category relationships

Logtalk provides two sets of built-in predicates that enable us to query the system about the possible relationships that a category can have with other entities.

The built-in predicates `implements_protocol/2` and `implements_protocol/3` find which categories implements which protocols:

```
| ?- implements_protocol(Category, Protocol).
```

or, if we want to know the implementation scope:

```
| ?- implements_protocol(Category, Protocol, Scope).
```

Note that, if we use an uninstantiated variable for the first argument, we will need to use the `current_category/1` built-in predicate to ensure that the returned entity is a category and not an object.

To find which objects (or categories) import which categories we can use the `imports_category/2` or `imports_category/3` built-in predicates:

```
| ?- imports_category(Object, Category).

| ?- imports_category(Category1, Category2).
```

or, if we want to know the importation scope:

```
| ?- imports_category(Object, Category, Scope).

| ?- imports_category(Category1, Category2, Scope).
```

Note that a category may be imported by several objects (or other categories).

Category properties

We can find the properties of defined categories by calling the built-in predicate `category_property/2`:

```
| ?- category_property(Category, Property).
```

A category may have the property `static`, `dynamic`, or `built_in`. Dynamic categories can be abolished in runtime by calling the `abolish_category/1` built-in predicate.

Importing categories

Any number of objects can import a category. In addition, an object may import any number of categories. The syntax is very simple:

```
:- object(Object,
    imports(Category1, Category2, ...)).
    ...
:- end_object.
```

To make all public predicates imported via a category protected or to make all public and protected predicates private we prefix the category's name with the corresponding keyword:

```
:- object(Object,
    imports(private::Category)).
    ...
:- end_object.
```

or:

```
:- object(Object,
    imports(protected::Category)).
    ...
:- end_object.
```

Omitting the scope keyword is equivalent to writing:

```
:- object(Object,
    imports(public::Category)).
    ...
:- end_object.
```


XHTML + CSS Last updated on: June 29, 2006