Objects

Logtalk objects should be regarded as a way to encapsulate and reuse predicate definitions. Instead of a single clause database containing all your code, Logtalk objects provide separated namespaces or databases allowing the partitioning of code in more manageable parts. Logtalk does not try to bring some sort of new dynamic state change concept to Logic Programming or Prolog.

In Logtalk, the only pre-defined objects are the pseudo-objects `user` and `debugger`, which are described later at the end of this section.

Objects, prototypes, classes, and instances

Objects, prototypes, parents, classes, subclasses, superclasses, metaclasses, and instances are all terms that always designate an object. Different names are used to emphasize the role played by an object in a particular context. We use a term other than object when we want to make the relationship with other objects explicit. There are only three kinds of encapsulation entities in Logtalk: objects, protocols, and categories.

Defining a new object

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 object. 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 an object named `vehicle` and save it in a `vehicle.lgt` source file which will be compiled to a `vehicle.pl` Prolog file.

Object names can be atoms or compound terms (if we are defining parametric objects, see below). Objects, categories and protocols share the same name space: we can not have an object with the same name as a protocol or a category.

Object code (directives and predicates) is textually encapsulated by using two Logtalk directives: `object/1-5` and `end_object/0`. The most simple object will be one that is self-contained, not depending on any other Logtalk entity:

```
:- object(Object).
    ...
:- end_object.
```

If an object implements one or more protocols then the opening directive will be:

```
:- object(Object,
    implements(Protocol)).
    ...
:- end_object.
```

An object can import one or more categories:

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

If an object implements protocols and imports categories then we will write:

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

In object-oriented programming objects are usually organized in hierarchies that enable interface and code sharing by inheritance. In Logtalk, we can construct prototype-based hierarchies by writing:

```
:- object(Prototype,
    extends(Parent)).
    ...
:- end_object.
```

We can also have class-based hierarchies by defining instantiation and specialization relations between objects. To define an object as a class instance we will write:

```
:- object(Object,
    instantiates(Class)).
    ...
:- end_object.
```

A class may specialize another class, its superclass:

```
:- object(Class,
    specializes(Superclass)).
    ...
:- end_object.
```

If we are defining a reflexive system where every class is also an instance, we will probably be using the following pattern:

```
:- object(Class,
    instantiates(Metaclass),
    specializes(Superclass)).
    ...
:- end_object.
```

In short, an object can be a *stand-alone* object or be part of an object hierarchy. The hierarchy can be prototype-based (defined by extending other objects) or class-based (with instantiation and specialization relations). An object may also implement one or more protocols or import one or more categories.

A *stand-alone* object (i.e. an object with no extension, instantiation, or specialization relations with other objects) is always compiled as a prototype, that is, a self-describing object. If we want to use classes and instances, then we will need to specify at least an instantiation or a specialization relation. The best way to do this is to define a set of objects that provide the basis of a reflective system [Cointe 87, Moura 94]. For example:

```
:- object(object,               % default root of the inheritance graph
    instantiates(class)).       % predicates common to all objects
    ...
:- end_object.


:- object(class,                % default metaclass for all classes
    instantiates(class),        % predicates common to all instantiable classes
    specializes(abstract_class)).
    ...
:- end_object.


:- object(abstract_class,       % default metaclass for all abstract classes
    instantiates(class),        % predicates common to all classes
    specializes(object)).
    ...
:- end_object.
```

Note that with these instantiation and specialization relations `object`, `class` and `abstract_class` are, at the same time, classes and instances of some class. In addition, each object inherits its own predicates and the predicates of the other two objects without any inheritance loop problems.

If you do not need a reflective system solution but still want to play with classes and instances then you can simplify the above scheme by making an object an instance of itself or, if you prefer, by making a class its own metaclass. For example:

```
:- object(class,
    instantiates(class)).
    ...
:- end_object.
```

We can have, in the same application, both prototype and class-based hierarchies (and freely exchange messages between all objects). We can not however mix the two types of hierarchies by, e.g., specializing an object that extends another object in this current Logtalk version.

Parametric objects

Parametric objects have a compound term for name instead of an atom. This compound term usually contains free variables that are instantiated when sending a message to the object. The object predicates can then be coded to depend on the variables instantiation values. When an object state is set at object creation and never changed, parameters provide a better solution than using the object's database via asserts. Parametric objects can also be used to attach a set of predicates to terms that share a common functor and arity.

In order to give access to an object parameters, Logtalk provides the `parameter/2` built-in local method:

```
:- object(Functor(Arg1, Arg2, ...)).

    ...

    Predicate :-
        ...,
        parameter(Number, Value),
        ... .
```

Note that we can't use this method with the message sending operators (`::/2`, `::/1` or `^^/1`). An alternative solution is to use the built-in local method `this/1`. For example:

```
:- object(foo(Arg)).

    ...

    bar :-
        ...,
        this(foo(Arg)),
        ... .
```

Both solutions are equally efficient because the runtime cost of the methods `this/1` and `parameter/2` is negligible. The drawback of the second solution is that we must check all calls of `this/1` if we change the object name.

When storing a parametric object in its own source file, the convention is to name the file after the object, with the object arity appended. For instance, when defining an object named `sort(Type)`, we may save it in a `sort_1.lgt` text file. This way it is easy to avoid file name clashes when saving Logtalk entities that have the same functor but different arity.

Compound terms with the same functor and (usually) the same number of arguments as a parametric object identifier may act as *proxies* to a parametric object. Proxies may be stored on the database as Prolog facts and be used to represent different instantiations of a parametric object parameters.

Finding defined objects

We can find, by backtracking, all defined objects by calling the `current_object/1` built-in predicate with an uninstantiated variable:

```
| ?- current_object(Object).
```

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

Creating a new object in runtime

An object can be dynamically created at runtime by using the `create_object/4` built-in predicate:

```
| ?- create_object(Object, Relations, Directives, Clauses).
```

The first argument, the name of the new object (a Prolog atom or compound term), should not match any existing entity name. The remaining three arguments correspond to the relations described in the opening object directive and to the object code contents (directives and clauses).

For instance, the call:

```
| ?- create_object(foo, [extends(bar)], [public(foo/1)], [foo(1), foo(2)]).
```

is equivalent to compiling and loading the object:

```
:- object(foo,
    extends(bar)).

    :- dynamic.

    :- public(foo/1).

    foo(1).
    foo(2).

:- end_object.
```

If we need to create a lot of (dynamic) objects at runtime, then is best to define a metaclass or a prototype with a predicate that will call this built-in predicate to make new objects. This predicate may provide automatic object name generation, name checking, and accept object initialization options.

Abolishing an existing object

Dynamic objects can be abolished using the `abolish_object/1` built-in predicate:

```
| ?- abolish_object(Object).
```

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

Object directives

Object directives are used to set initialization goals and object properties and to document an object dependencies on other Logtalk entities.

Object initialization

We can define a goal to be executed as soon as an object 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 to other object. For example:

```
:- object(foo).

    :- initialization(init).
    :- private(init/0).

    init :-
        ... .

    ...

:- end_object.
```

Or:

```
:- object(assembler).

    :- initialization(control::start).
    ...

:- end_object.
```

The initialization goal can also be a message to *self* in order to call an inherited or imported predicate. For example, assuming that we have a `monitor` category defining a `reset/0` predicate:

```
:- object(profiler,
    imports(monitor)).

    :- initialization(::reset).
    ...

:- end_object.
```

Note, however, that descendant objects do not inherit initialization directives. In this context, *self* denotes the object that contains the directive. Also note that by initialization we do not necessarily mean setting an object dynamic state.

Dynamic objects

As usually happens with Prolog code, an object can be either static or dynamic. An object created during the execution of a program is always dynamic. An object defined in a file can be either dynamic or static. Dynamic objects are declared by using the `dynamic/0` directive in the object 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 objects when these need to be abolished during program execution. Also, note that we can declare and define dynamic predicates in a static object.

Object dependencies

Besides the relations declared in the object opening directive, the predicate definitions contained in the object may imply other dependencies. These 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 may be used by the Logtalk runtime to ensure that all needed entities are loaded when running an application.

Object documentation

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

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

See the documenting Logtalk programs session for details.

Object relationships

Logtalk provides five sets of built-in predicates that enable us to query the system about the possible relationships that an object may have with other entities.

The built-in predicates `instantiates_class/2` and `instantiates_class/3` can be used to query all instantiation relations:

```
| ?- instantiates_class(Instance, Class).
```

or, if we want to know the instantiation scope:

```
| ?- instantiates_class(Instance, Class, Scope).
```

Specialization relations can be found by using either the `specializes_class/2` or the `specializes_class/3` built-in predicates:

```
| ?- specializes_class(Class, Superclass).
```

or, if we want to know the specialization scope:

```
| ?- specializes_class(Class, Superclass, Scope).
```

For prototypes, we can query extension relations with the `extends_object/2` or the `extends_object/3` built-in predicates:

```
| ?- extends_object(Object, Parent).
```

or, if we want to know the extension scope:

```
| ?- extends_object(Object, Parent, Scope).
```

In order to find which objects import which categories we can use the built-in predicates `imports_category/2` or `imports_category/3`:

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

or, if we want to know the importation scope:

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

To find which objects implements which protocols we can use the `implements_protocol/2` or the `implements_protocol/3` built-in predicates:

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

or, if we want to know the implementation scope:

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

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

Object properties

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

```
| ?- object_property(Object, Property).
```

An object may have the property `static`, `dynamic`, or `built_in`. Dynamic objects can be abolished in runtime by calling the `abolish_object/1` built-in predicate.

The pseudo-object user

Logtalk defines a built-in, pseudo-object named `user` that contains all user predicate definitions not encapsulated in a Logtalk entity. These predicates are assumed to be implicitly declared public.

The pseudo-object debugger

Logtalk defines a built-in, pseudo-object named `debugger` which implements the Logtalk built-in debugger (see the section on debugging Logtalk programs).

XHTML + CSS Last updated on: January 29, 2006