

---

# From Plain Prolog to Logtalk Objects: Effective Code Encapsulation and Reuse

---

Paulo Moura

Dep. of Computer Science, Univ. of Beira Interior, Portugal

Center for Research in Advanced Computing Systems

INESC Porto, Portugal

# Objects in Prolog?!?

---

# Objects have identifiers and dynamic state!

- It seems Prolog modules were there first:

Identifier!

```
:- module(broadcast, [...]).
```

Dynamic state!

```
:- dynamic(listener/4).
```

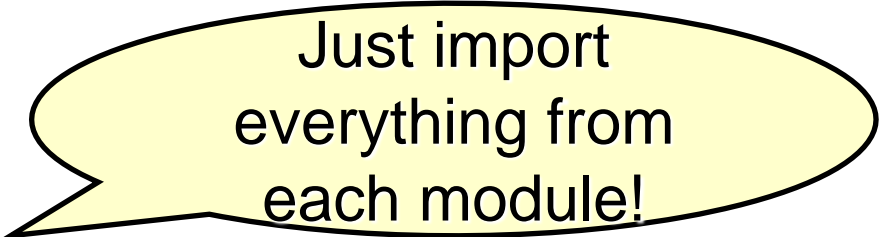
...

```
assert_listener(Templ, Listener, Module, TheGoal) :-  
    asserta(listener(Templ, Listener, Module, TheGoal)).  
retract_listener(Templ, Listener, Module, TheGoal) :-  
    retractall(listener(Templ, Listener, Module, TheGoal)).
```

# Objects inherit lots of stuff I don't need!

- Objects are not necessarily tied to hierarchies.
- But how about typical Prolog module code?

```
:- module(aggregate, [...]).  
:- use_module(library(ordsets)).  
:- use_module(library(pairs)).  
:- use_module(library(error)).  
:- use_module(library(lists)).  
:- use_module(library(apply))....
```



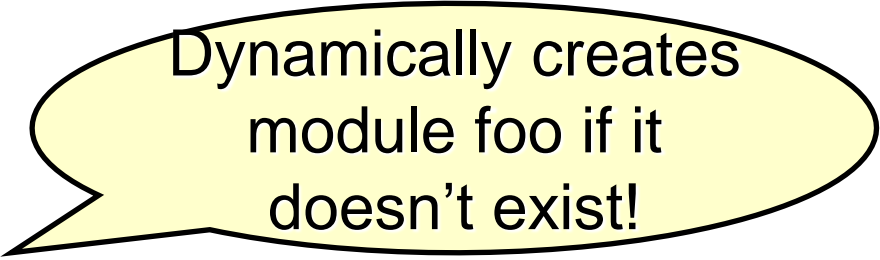
Just import  
everything from  
each module!

# Objects are dynamically created!

---

- Only if really necessary.
- Objects can be (and often are) static, simply loaded from source files...
- But, guess what, Prolog modules were there first:

```
?- foo:assertz(bar).  
yes
```



Dynamically creates  
module foo if it  
doesn't exist!

# Why not stick to modules?!?

---

Prolog modules fail to:

- enforce encapsulation
  - ◆ in most implementations, you can call any module predicate using explicit qualification
- implement predicate namespaces
  - ◆ due to the import semantics and current practice, module predicate names are often prefixed... with the module name!
- provide a clean separation between loading and importing
  - ◆ `ensure_loaded/1` impersonating `use_module/1` while it should be equivalent to `use_module(..., [])`

# Why not stick to modules?!?

---

Prolog modules fail to:

- support separating interface from implementation
  - ◆ the ISO Prolog standard proposes a solution that only allows a single implementation for interface!
- provide a standard mechanism for predicate import conflicts
  - ◆ being fixed, however, in recent versions of some Prolog compilers
- provide the same semantics for both implicit and explicit qualified calls to meta-predicates

# Why not simply improve module systems?!?

---

- No one wants to break backward compatibility
- An ISO Prolog standard that ignores current practice, tries to do better, and fails
- Improvements perceived as alien to Prolog traditions (not to mention the reinvention of the wheel)
- Good enough mentality (also few users working on large apps)
- Instant holy wars when discussing modules



# Logtalk design goals

---

# So... which object-oriented features to adopt?

---

- **Code encapsulation**

- objects (including parametric objects)
- protocols (aka interfaces; separate interface from implementation)
- categories (fine-grained units of code reuse)

- **Code reuse**

- message sending (decoupling between messages and methods)
- inheritance (taxonomic knowledge is pervasive)
- composition (mix-and-match)

# Design goals

---

- Extend Prolog with code encapsulation and reuse features
  - ◆ based on an interpretation of object-oriented concepts in the context of logic programming
- Multi-paradigm language
  - ◆ integrating predicates, objects, events, and threads
- Support for both prototypes and classes
  - ◆ object relations interpreted as patterns of code reuse
- Compatibility with most Prolog compilers and the ISO Prolog Core standard

# A quick tour on Logtalk programming

---

# Objects

## Defining objects

```
:- object(list).
```

```
:- public(append/3).
```

```
append([], L, L).
```

```
append([H| T], L, [H| T2]) :-  
    append(T, L, T2).
```

```
:- public(member/2).
```

```
member(H, [H| _]).
```

```
member(H, [_| T]) :-  
    member(H, T).
```

```
:- end_object.
```

## Sending messages

```
?- list::append(L1, L2, [1, 2, 3]).
```

```
L1 = [], L2 = [1, 2, 3];
```

```
L1 = [1], L2 = [2, 3];
```

```
L2 = [1, 2], L2 = [3];
```

```
L3 = [1, 2, 3], L2 = []
```

```
yes
```

```
?- list::member(X, [a, b, c]).
```

```
X = a;
```

```
X = b;
```

```
X = c
```

```
yes
```

# Protocols (aka "Interfaces")

## Defining protocols

```
:- protocol(listp).  
  
    :- public(append/3).  
    :- public(member/2).  
    ...  
  
:- end_protocol.
```

## Implementing protocols

```
:- object(list,  
          implements(listp)).  
  
    append([], L, L).  
    append([H| T], L, [H| T2]) :-  
        append(T, L, T2).  
  
    member(H, [H| _]).  
    member(H, [_| T]) :-  
        member(H, T).  
  
:- end_object.
```

# Object hierarchies: Prototypes

## A self-contained object

```
:- object(state_space).  
  
    :- public(initial_state/1).  
    :- public(next_state/2).  
    :- public(goal_state/1).  
    ...  
  
:- end_object.
```

## An object defined by extension

```
:- object( heuristic_state_space,  
          extends(state_space)).  
  
    :- public(heuristic/2).  
    ...  
  
:- end_object.
```

# Object hierarchies: Classes

## A class

```
:- object(person,  
          instantiates(class),  
          specializes(object)).  
  
:- public(name/1).  
:- public(age/1).  
...  
  
:- end_object.
```

## An instance

```
:- object( paulo,  
          instantiates(person)).  
  
name('Paulo Moura').  
age(41).  
...  
  
:- end_object.
```



# Parametric objects

```
:- object(rectangle(_Width, _Height)).
```

```
    :- public([width /1, height/1, area/1, perimeter/1]).
```

```
    ...
```

```
width(Width) :-  
    parameter(1, Width).
```

```
height(Height) :-  
    parameter(2, Height).
```

```
area(Area) :-  
    ::width(Width),  
    ::height(Height),  
    Area is Width*Height.
```

```
    ...
```

```
:- end_object.
```

Parameters are  
**logical variables**,  
shared by all object  
predicates.

# Using parametric objects

```
?- rectangle(3, 4)::area(Area).
```

```
Area = 12
```

```
yes
```

```
% Prolog facts are possible instantiations of a parametric object identifier
```

```
rectangle(1, 2).
```

```
rectangle(2, 3).
```

```
rectangle(3, 4).
```

```
?- findall(Area, {rectangle(_, _)}::area(Area), Areas).
```

```
Areas = [2, 6, 12]
```

```
yes
```

# Categories

---

- Dual concept of protocols (functional cohesion)
- Fine-grained units of code reuse (that don't make sense as stand-alone entities)
- Can contain both interface and implementation
- Can be (virtually) imported by any object (classes, instances, or prototypes)
- Can extend existing objects (as in Objective-C)
- Provide runtime transparency (for descendant objects)
- Can declare and use dynamic predicates (each importing object will have its own set of clauses; enables a category to define and manage object state)

# Categories

---

- Can be extended (as with protocols, try to not break functional cohesion!)
- Compilation units, independently compiled from importing objects or implemented protocols (enabling incremental compilation)
- Allows an object to be updated by simply updating the imported categories, without any need to recompile it or to access its source code
- Can be dynamically created and abolished at runtime (just like objects or protocols)

# Defining and importing categories

```
:- category(engine).  
  
:- public(capacity/1).  
:- public(cylinders/1).  
:- public(horsepower_rpm/2).  
...  
  
:- end_category.
```

```
:- object(car,  
    imports(engine)).  
  
...  
  
:- end_object.
```

# Complementing existing objects

---

```
:- object(employee).
```

```
...
```

```
:- end_object.
```

```
:- category(logging  
complements(employee)).
```

```
...
```

```
:- end_category.
```

# Event-driven programming

---

- Allows minimization of object coupling
- Provides a mechanism for building reflexive applications
- Provides a mechanism for easily defining method (predicate) pre- and post-conditions
- Implemented by the language runtime at the message sending mechanism level

# Events

---

- An event corresponds to sending a message
- Described by (Event, Object, Message, Sender)
- before events and after events
- Independence between the two types of events
- All events are automatically generated by the message sending mechanism
- The events watched at any moment can be dynamically changed at runtime



# Monitors

---

- Monitors are objects automatically notified whenever registered events occur
- Any object can act as a monitor
- Define event handlers (before/3 and after/3)
- Unlimited number of monitors for each event
- The monitor status of an object can be dynamically changed in runtime
- The events handlers never affect the term that represents the monitored message

# Monitor semantics

---

- All before event handlers must succeed, so that the message processing can start
- All after event handlers must succeed so that the message itself succeeds; failure of any handler forces backtracking over the message execution (handler failure never leads to backtracking over the preceding handlers)

# Defining events and monitors

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

```
% Setup employee as a monitor for  
% every message sent to itself:
```

```
:- initialization(  
    define_events(  
        before,  
        employee,  
        —,  
        —,  
        employee  
    )  
).
```

```
% Define a "before" event handler for  
% the complemented object:
```

```
:- category(logging,  
    implements(monitoring),  
    % ← event handler protocol  
    complements(employee)).
```

```
before(This, Message, Sender) :-  
    this(This),  
    write('Received message '),  
    writeq(Message),  
    write(' from '),  
    writeq(Sender),  
    nl.
```

```
...  
:- end_category.
```

# Logtalk as a portable Prolog application

---

# Logtalk Architecture

---

Logtalk

Abstraction layer (Prolog config files)

Back-end Prolog compiler

# Supported Prolog compilers

---

## Runs out-of-the box using

- B-Prolog
- CxProlog
- ECLiPSe
- GNU Prolog
- Qu Prolog
- SICStus Prolog
- SWI-Prolog
- XSB
- YAP

## Supported older versions

- Ciao
- IF/Prolog
- JIProlog
- K-Prolog
- Open Prolog
- ALS Prolog
- Amzi! Prolog
- BinProlog
- LPA MacProlog, LPA WinProlog
- Prolog II+
- Quintus Prolog

# The good...

---

- Free, open source
  - ◆ Artistic License 2.0
- Plain Prolog implementation
  - ◆ no foreign code
- Supports most Prolog compilers
  - ◆ Not much left that's not on the previous list
- Portable libraries
  - ◆ Yes, they do exist!
- Competitive features
  - ◆ compared with both Prolog modules and OOP languages
- Competitive performance
  - ◆ close to plain Prolog when using static binding

# ... the bad...

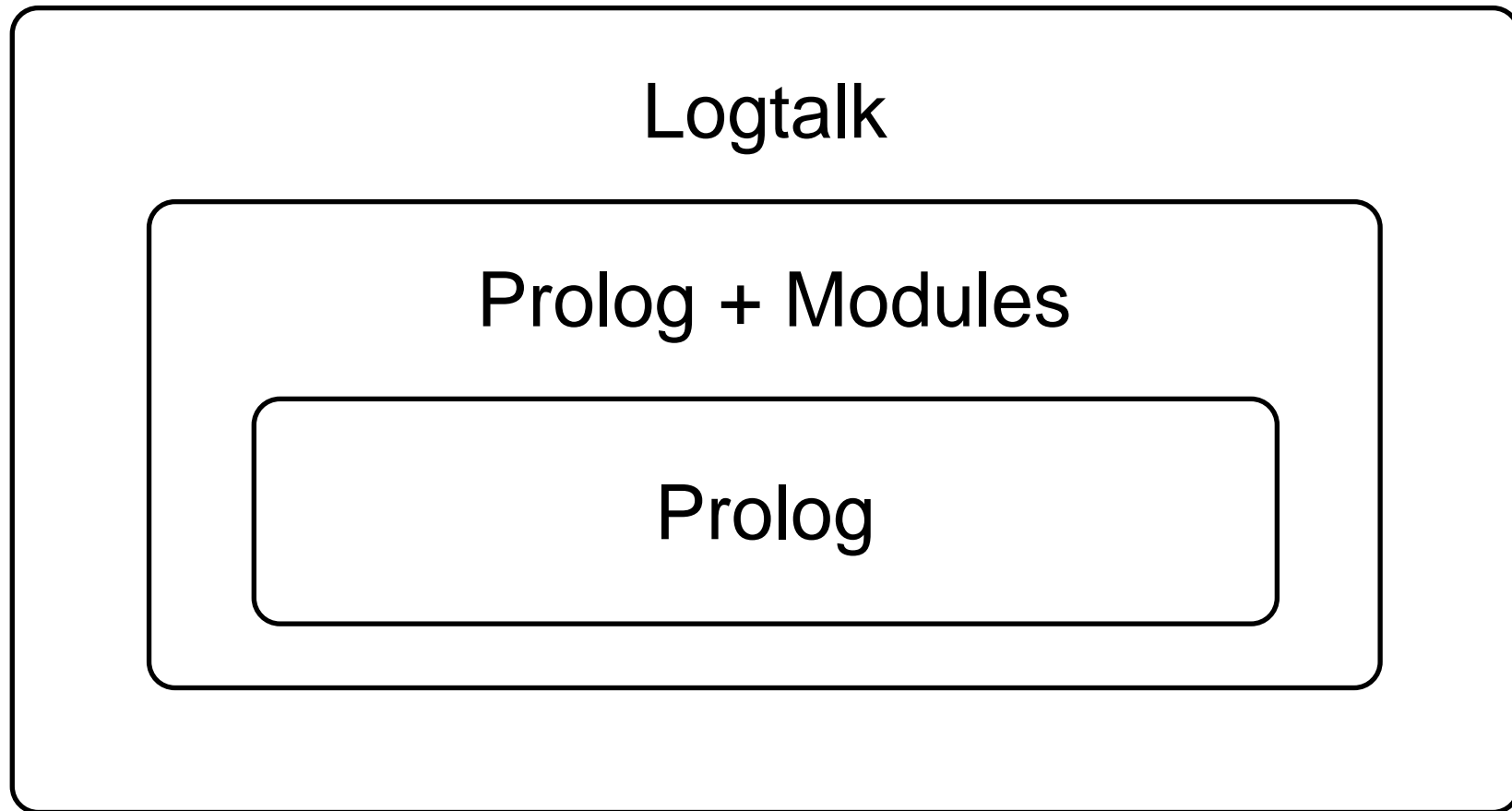
---

- Limited feature set due to the lack of Prolog standardization
- Some features are only available in some Prolog compilers
  - ◆ e.g Unicode, threads
- No book about Logtalk programming yet



# Logtalk versus Prolog

---



# Compiling Prolog modules as Logtalk objects

---

- Supported directives
  - ◆ module/1-2
  - ◆ use\_module/2
  - ◆ export/1
  - ◆ reexport/2
  - ◆ meta\_predicate/1
  - ◆ use\_module/1 (with most back-end Prolog compilers)
- Caveats
  - ◆ module library meta-predicates taking closures are (currently) not supported

# Why compile Prolog modules as Logtalk objects?!?

---

- Run module code in Prolog compilers without a module system
  - ◆ ironic, I know
- Reuse module libraries as-is
  - ◆ lots of good stuff out there
- Locating potential issues when migrating Prolog code to Logtalk
  - ◆ ...
- The proof is in the pudding
  - ◆ ...

# Programming Support



# Writing Logtalk code

---

## Text services

syntax highlight (sh), auto-indentation (ai), code completion (cc), code folding (cf), code snippets (cs), entity index (ei)

- TextMate (sh, ai, cc, cf, cs, ei)
- SubEthaEdit (sh, cc, ei)
- jEdit (sh, ai, cc, cf, cs)
- Kate (sh, cf)
- Gedit (sh, cs)
- Emacs (sh)
- Vim (sh, ai, cc, cf, ei)
- NEdit (sh)

# Publishing Logtalk Source Code

---

- Pygments (e.g. Trac)
- Source-highlight (HTML, LaTeX, DocBook, ...)
- SHJS (e.g. web pages)
- Highlight (HTML, RTF, LaTeX, ...)
- GeSHi (e.g. wikis, blogs)
- SyntaxHighlighter (e.g. web pages)

# Programming Tools

---

- Built-in debugger (extended version of the traditional procedure box model with unification and exception ports)
- Unit test framework
- Entity diagram generator library
- Documenting tools
- Supports the built-in profilers and graphical tracers of selected Prolog compilers

# Logtalk in Eclipse



Full Eclipse integration on top the PDT and SWI-Prolog

- Editor
- Outline
- Search
- Console
- Graphical debugger

See <http://sewiki.iai.uni-bonn.de/research/pdt/docs/v2.1/start>

- Most functionalities of the PDT work for Logtalk too