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

Someone said I need one...

Spoilers

- Objects in a Prolog world
- Logtalk design goals
- Logtalk architecture
- Logtalk versus Prolog and Prolog modules
- Logtalk overview and quick tour
- Some demos (if time allows)
- Logtalk as a portable Prolog application
- Logtalk programming support

A warning...

- I have a **laser** and I’m not afraid to use it ... Beware of asking embarrassing questions!
- But please feel free to interrupt and ask nice ones that make the speaker shine!

A bit of history...

- Project started in January 1998
- First version for registered users in July 1998
- First public beta in October 1998
- First stable version in February 1999

Logtalk in numbers

- Compiler + runtime: ~11000 lines of Prolog code
- 37 major releases, 122 including minor ones
- Current version: Logtalk 2.37.2

minor release number

major release number

second generation of Logtalk

- ~1500 downloads/month (so many earthlings, so little time)

Logtalk distribution

- Full sources (compiler/runtime, Prolog config files, libraries)
- MacOS X (pkg), Linux (rpm), Debian (deb), and Windows (exe) installers
- XHTML and PDF User and Reference Manuals (121 + 144 pages)
- +90 programming examples
- Support for several text editors and syntax highlighters (text services and code publishing)

Objects in Prolog?!?

We're being invaded!

Objects have identifiers and dynamic state!

- It seems Prolog modules are there first:

Identifier!

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

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

Dynamic state!

```
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... but, guess what, Prolog modules are there first:

Dynamically creates module foo if it doesn't exist!

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

Design goals

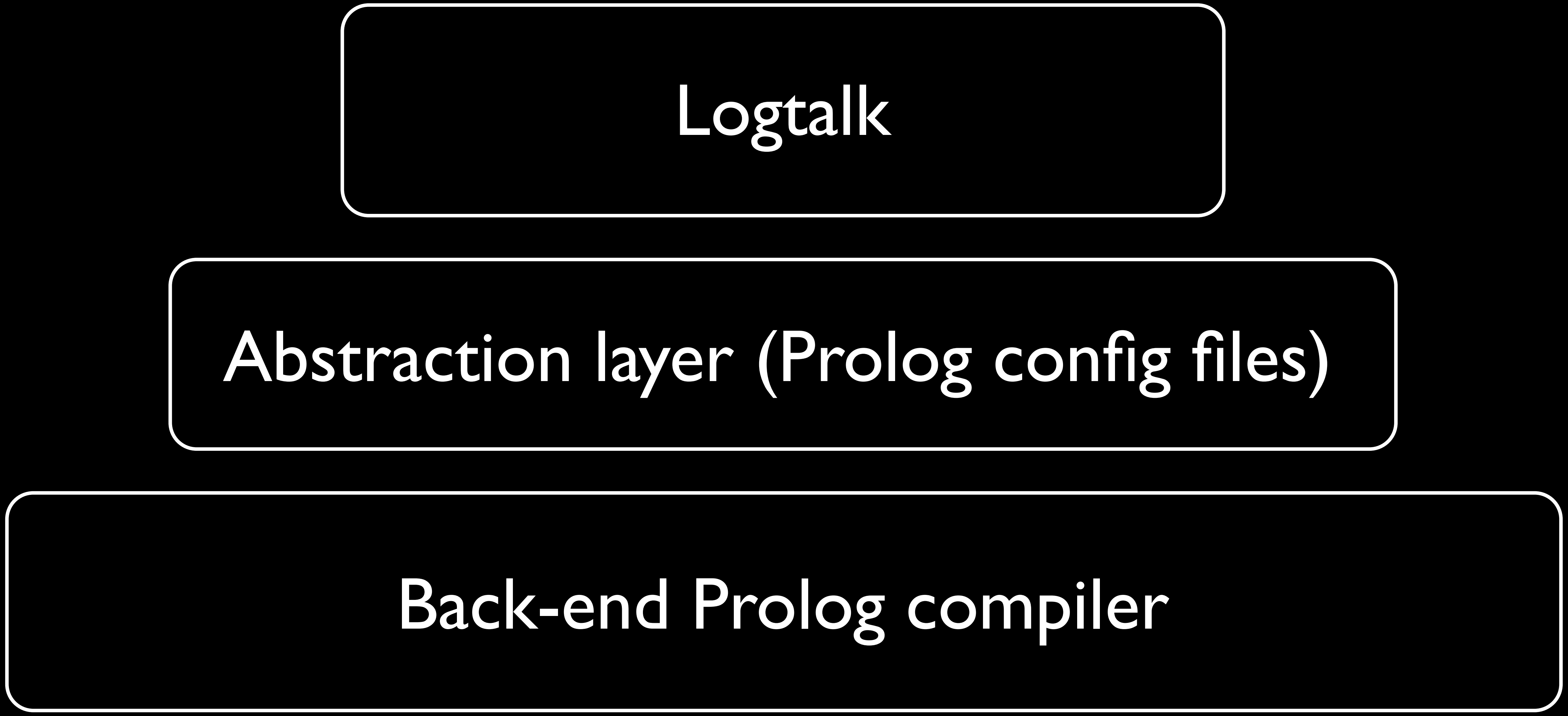
So... which object-oriented features to adopt?

- **Code encapsulation**
  - ➡ objects,
  - ➡ protocols/interfaces
- **Code reuse**
  - ➡ inheritance
  - ➡ composition

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, and events)
- 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

Logtalk Architecture



Supported Prolog compilers

- Used daily in Logtalk development:

  B-Prolog, ECLiPSe, GNU Prolog, SWI-Prolog, XSB, YAP

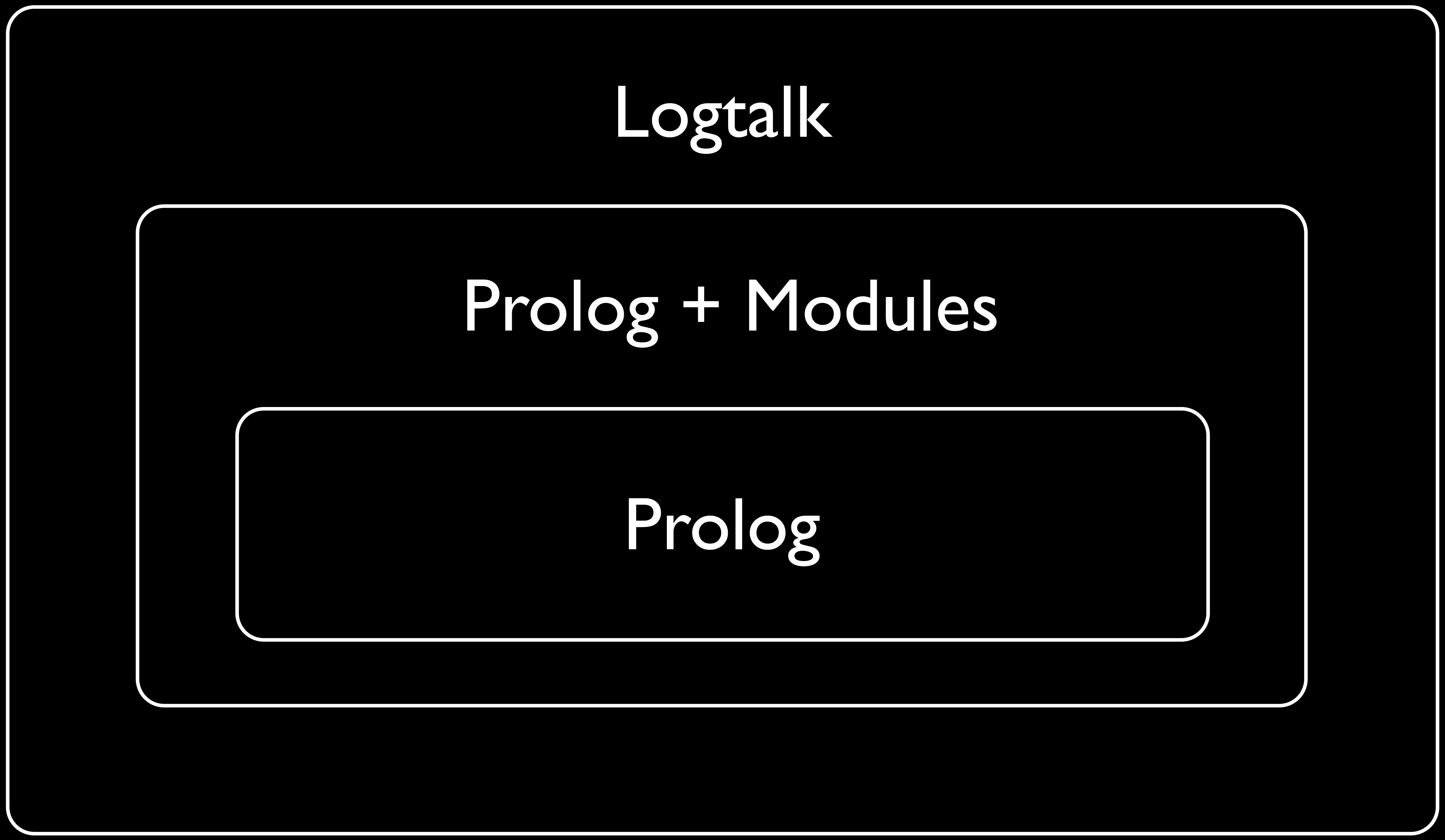
- Also runs without modifications using:

  Ciao Prolog, CxProlog, IF/Prolog, JIProlog, K-Prolog, Open Prolog, Qu Prolog, SICStus Prolog

- Runs after patching using:

  ALS Prolog, Amzi! Prolog, BinProlog, LPA MacProlog, LPA WinProlog, Prolog II+, Quintus Prolog

Logtalk versus Prolog



Compiling Prolog modules as objects

- Supported directives: `module/1-2`, `use_module/2`, `export/1`, `reexport/2`, `meta_predicate/1` (plus `use_module/1` with some back-end Prolog compilers)
- Caveats: module library meta-predicates taking closures are (currently) not supported

Why compile Prolog modules as objects?!?

- Locating potential issues when migrating Prolog code to Logtalk
- Run Prolog module code in Prolog compilers without a module system
- Reuse Prolog module libraries as-is
- The proof is in the pudding
- I have been a bad boy; I must punish myself

Overview

Logtalk for Prolog programmers

- Prolog syntax plus a few operators and directives for a smooth learning curve (no need to bring your climbing gear)
- can use most Prolog implementations as a back-end compiler (and allows you to take advantage of Prolog compiler specific goodies)
- easy porting of Prolog applications (wrap-around!)
- unmatched portability (leaves Prolog modules in the dust ;-)
- private, protected, and public object predicates
- static and dynamic object predicates
- static and dynamic objects

Logtalk for object-oriented programmers

- prototypes and classes (no need to take sides and envy your neighbors)
- parametric objects (think runtime, not compile time templates)
- multiple object hierarchies (as one size doesn't fit all)
- multiple inheritance and multiple instantiation
- private, protected, and public inheritance
- protocols (aka interfaces; can be implemented by both prototypes and classes)
- categories (fine-grained units of code reuse;can be imported by both prototypes and classes)
- static and dynamic binding (with predicate lookup caching)

Other interesting features

- high-level multi-threading programming (with selected back-end Prolog compilers)
- event-driven programming (why break encapsulation?)
- dynamic programming language (how trendy!)
- reflection (both structural and behavioral)
- automatic generation of XML documentation files

A quick tour

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
```

Defining and implementing protocols

```
:- protocol(listp).

  :- public(append/3).
  :- public(member/2).
  ...

:- end_protocol.


:- object(list,
    implements(listp)).

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

:- end_object.
```

Object hierarchies: prototypes

```
:- object(state_space).

  :- public(initial_state/1).
  :- public(next_state/2).
  :- public(goal_state/1).
  ...

:- end_object.


:- object(heuristic_state_space,
     extends(state_space)).

  :- public(heuristic/2).
  ...

:- end_object.
```

Object hierarchies: classes

```
:- object(person,
     instantiates(class),
     specializes(object)).

  :- public(name/1).
  :- public(age/1).
  ...

:- end_object.


:- 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.
```

Using parametric objects

```
| ?- rectangle(3, 4)::width(Width).

Width = 3
yes


| ?- rectangle(3, 4)::perimeter(Perimeter).

Perimeter = 14
yes


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

Area = 12
yes
```

Categories

- Dual concept of protocols
- Fine-grained units of code reuse
- 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
- 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 derived (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

```
% setup employee as a monitor for any message sent to itself:
:- initialization(define_events(before,employee,_,_,employee)).


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


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

  % define a "before" event handler for the complemented object:
  before(This, Message, Sender) :-
     this(This),
     write('Received message '), writeq(Message),
     write(' from '), writeq(Sender), nl.
  ...

:- end_category.
```

Logtalk users profile

- Wise people (bias? what bias?)
- Developing large applications...
- ... or applications where using modules would be awkward (e.g. representing taxonomic knowledge)
- Looking for portability (e.g. using YAP for speed and SWI-Prolog for the development environment)

Demo time!

Logtalk as a portable Prolog application

The good...

- Plain Prolog application (no foreign code)
- Runs in most Prolog compilers
- Free, open source (Artistic License 2.0)
- Portable libraries
- Competitive features (compared with both Prolog modules and OOP languages)
- Competitive performance (close to plain Prolog when using static binding)

... the bad...

- Some features are only available in some Prolog compilers (Unicode, Threads)
- Limited feature set due to the lack of Prolog standardization (are we there yet? NO!)
- Need to write a book about Logtalk programming (and plant a tree and have some...)

... and the ugly!

- Testing new releases across supported Prolog compilers and supported operating-systems (consumes valuable development time)
- Poor support for reflection in too many Prolog compilers (predicate properties, compiler version, environment information, ...)

Programming Support

Programming Support

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)
- Emacs (sh)
- Vim (sh, ai, cc, cf, ei)
- NEdit (sh)

Source Code Publishing

- Pygments (e.g. Trac)
- Source-highlight (HTML, LaTeX, DocBook, ...)
- Highlight (HTML, RTF, LaTeX, ...)
- GeSHi (Wikis)

Ive got you under my skin
Ive got you deep in the heart of me
So deep in my heart, that youre really a part of me
Ive got you under my skin

Ive tried so not to give in
Ive said to myself this affair never will go so well
But why should I try to resist, when baby will I know than well
That Ive got you under my skin

That’s all folks!

Id sacrifice anything come what might
For the sake of having you near
In spite of a warning voice that comes in the night
And repeats, repeats in my ear

Dont you know you fool, you never can win
Use your mentality, wake up to reality
But each time I do, just the thought of you
Makes me stop before I begin
Cause Ive got you under my skin

Please don’t forget to buy the nice t-shirt!