An Introduction to Logtalk

Paulo Moura
Logtalk author and maintainer

https://logtalk.org/

pmoura@logtalk.org

Logtalk

- Declarative object-oriented logic programming language
- Currently implemented as a trans-compiler to Prolog
- Designed to extend and leverage Prolog with a strong emphasis on portability
- Focused in code encapsulation and code reuse mechanisms and improved predicate semantics
- Comprehensive ecosystem

Presentation outline

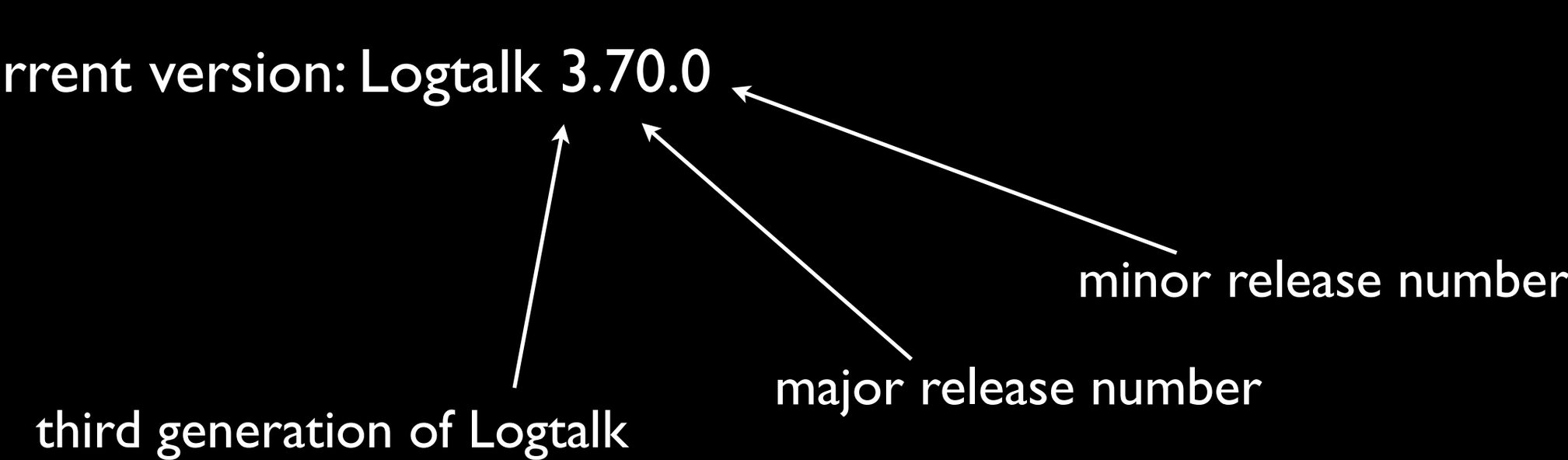
- History and numbers
- Objects in a Prolog world (and why)
- Logtalk design goals
- Logtalk architecture
- Logtalk overview and quick tour
- Logtalk as a portable Prolog application
- Logtalk programming support

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 (2.0) in February 1999
- Third generation (3.0) in January 2015

Logtalk in numbers

- Compiler + runtime: ~19500 lines of Prolog code + ~1500 lines of Logtalk code (excluding layout and comments)
- Third generation: 70 releases (release early, release often mantra)
- Current version: Logtalk 3.70.0



- ~1000 downloads/month (so many earthlings, so little time)

Logtalk distribution

- Full sources (compiler/runtime, Prolog integration support, documentation, libraries, developer tools, examples, ports, utilities, ...)
- macOS (.pkg), Linux (.rpm), Debian (.deb), and Windows (.exe) installers
- Handbook (user manual, reference manual, libraries and tools documentation, FAQ, glossary)
- ~200 programming examples
- Coding support (text editing services and code publishing)

Logtalk users and applications

- Academic users (research; undergraduate, master, and phd thesis)
- Industrial users (both startups and established companies)
- Some industry and users sponsorship
- Wide range of applications using multiple backends
- Applications where software engineering principles are key

Objects in Prolog?!?

We're being invaded!

Objects have identifiers and dynamic state!

- It seems Prolog modules are there first:

Identifier!

Dynamic state!

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

:- 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 are there first:

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

Dynamically creates module foo if it doesn't exist!

Why not stick to modules?

Prolog modules key characteristics:

- Designed as a simple solution to to hide auxiliary predicates
- Based on a predicate prefixing compilation mechanism
- Reuse based on import/export semantics
- Default importing of module exported predicates when loading a module file
- Strongly biased towards implicit predicate qualification

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 or its abbreviation!)
- provide a clean separation between loading and importing (avoiding import conflicts)
- provide portability for libraries and tools

Why not stick to modules?

Prolog modules fail to:

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

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 applications)
- Instant holy wars when discussing modules

Logtalk design goals

Which object-oriented features to adopt or invent?

- **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)

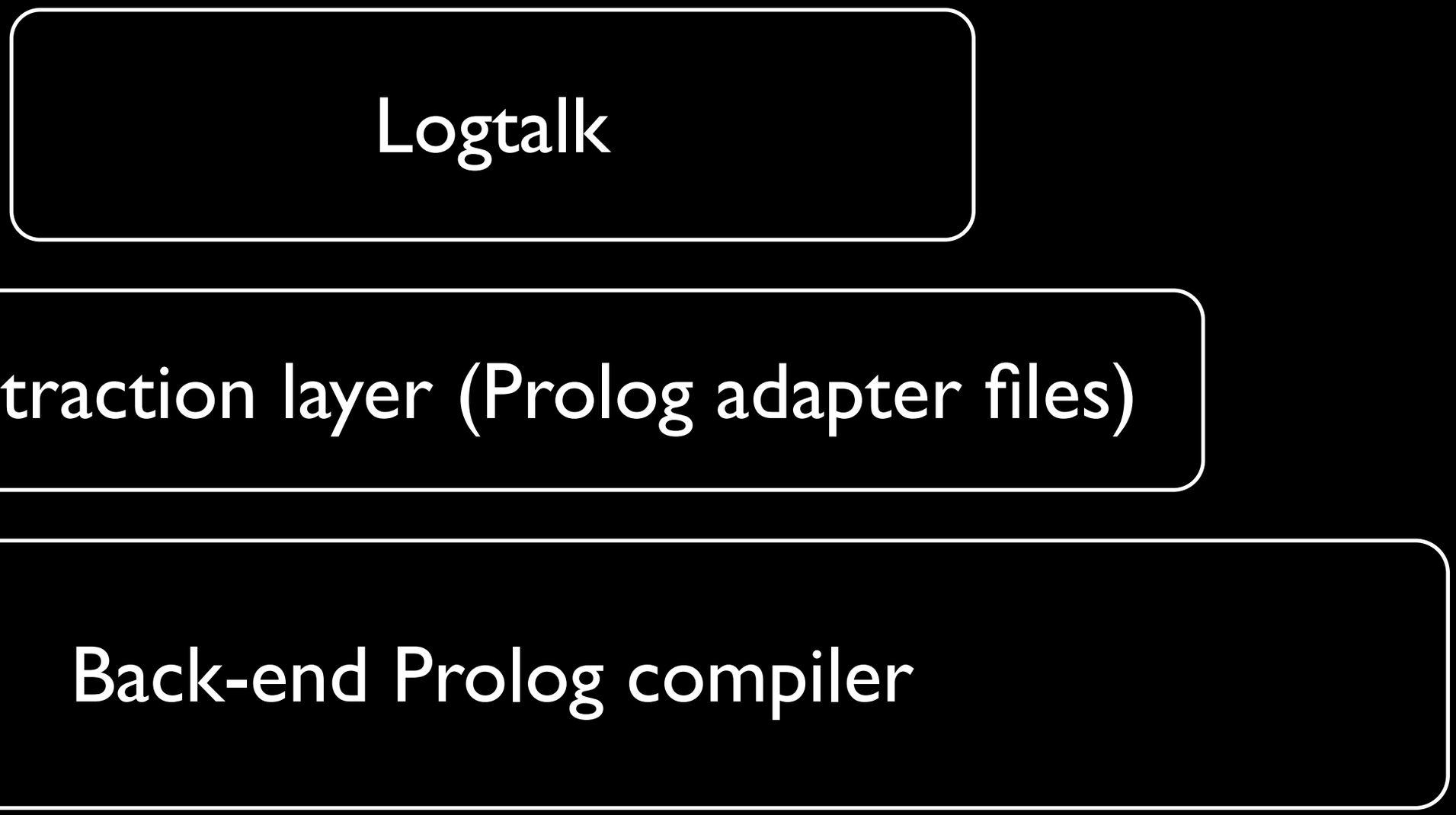
Declarative OOP

- Code *encapsulation* and *reuse patterns*
- Objects as units of encapsulation
- Objects encapsulate predicate declarations and definitions
- Object as *theories*
- Objects, protocols (interfaces), and categories as first-class entities
- Entity relations define reuse patterns and the *roles* played by the participating entities
- *Prototype*, *parent*, *class*, *instance*, *metaclass*, *subclass*, *superclass*, or *ancestor* are just roles that an object can play

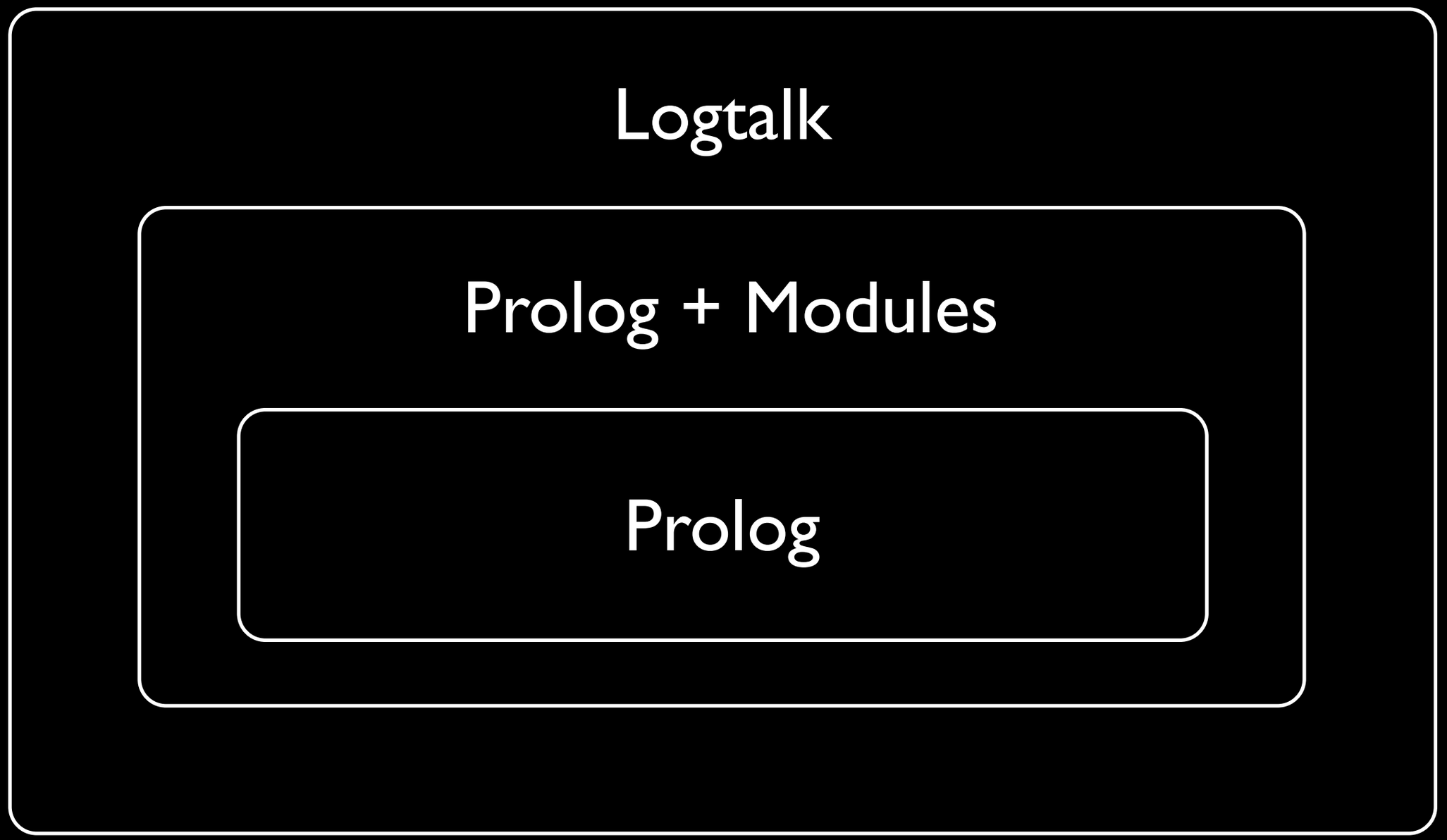
Design goals

- 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 (with "prototype" and "class" being object *roles* and object relations interpreted as *patterns of code reuse*)
- Compatibility with most Prolog compilers and the ISO Prolog Core standard

Logtalk Architecture



Logtalk versus Prolog



Supported Prolog compilers

- Runs out-of-the box using:

  B-Prolog, Ciao Prolog, CxProlog, ECLiPSe, GNU Prolog, JIProlog, XVM, Qu Prolog, Quintus Prolog, SICStus Prolog, SWI-Prolog, Tau Prolog, Trealla Prolog, XSB, YAP

- Older versions also supported:

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

Logtalk

overview

Logtalk for Prolog programmers

- Prolog syntax plus a few operators and directives for a smooth learning curve
- Can use most Prolog implementations as a back-end compiler (allows use of most Prolog compiler specific features and libraries)
- Easy porting of Prolog applications
- Unmatched portability (leaves Prolog modules in the dust)
- Private, protected, and public object predicates (encapsulation is enforced, unlike in most Prolog module systems)
- Static and dynamic object predicates
- Static and dynamic objects

Logtalk for object-oriented programmers

- Prototypes and classes (object *roles*, not distinct entities)
- Parametric objects
- Multiple object hierarchies
- Multiple inheritance and multiple instantiation
- Private, protected, and public inheritance (generalized to protocols and categories)
- 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; can be used for hot patching)
- Static binding and dynamic binding (with predicate lookup caching)

Other interesting features

- Extended support for DCGs
- High-level multi-threading programming
- Event-driven programming
- Reflection (both structural and behavioral)
- Message printing and question asking mechanisms
- Dependable term-expansion mechanism
- Lambda expressions
- Coinduction
- Top of class developer tools

A quick tour on Logtalk programming

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, 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, ...]).
    ...
    width(_Width_).

    height(_Height_).

    area(Area) :-
        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 as parametric object proxies (i.e. 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 hot patch 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
(hot patching)

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

:- end_category.
```

Logtalk as a portable Prolog application

The good...

- Plain portable Prolog implementation (no foreign code)
- Supports most Prolog compilers
- Free, open source (Apache License 2.0)
- Portable libraries (yes, they do exist!)
- Portable tools (ditto!)
- 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 with some backend Prolog compilers (e.g. Unicode, threads)
- Limited feature set due to the lack of Prolog standardization

... and the ugly!

- Testing new releases across all supported Prolog compilers and all supported operating-systems is time consuming (better automation is possible, however)
- Poor support for reflection in too many Prolog compilers (predicate properties, compiler version, environment information, ...)

Programming Support

- Coding
- Code publishing
- Developer tools
- Deployment tools

Coding

- Syntax highlight
- Auto-indentation
- Code completion
- Code folding
- Snippets

Publishing Logtalk Source Code

- Source code repositories
- Web pages (e.g. wikis, blogs)
- Papers

Programming Tools

- Tutor (explains compiler errors and warnings)
- Linter (one of the best; only SICStus Prolog compares)
- Debugger (extended version of the traditional procedure box model with unification and exception ports)
- Unit test framework (best in class)
- Diagrams (library, entity, directory, file, xref, …)
- Documenting tools

Programming Tools

- Ports profiler
- Dead code scanner
- Code metrics
- Make (supports multiple targets)
- Porting
- Version management
- Support for selected backend Prolog tools

Deployment Tools

- Logtalk and Logtalk applications can be precompiled to relocatable Prolog code
- Support for selected backend Prolog compilers

Ive got you under my skin
Ive got you deep in the heart of me
So deep in my heart, that you're 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

Don't 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!