Wrapper for Object-Oriented Programming in Erlang

Contact: about (dash) wooper (at) esperide (dot) com

Creation Date: Thursday, February 25, 2008

Lastly Updated: Monday, March 12, 2018

The latest version of this documentation is to be found at the official WOOPER website (`http://wooper.esperide.org`).

The documentation is also mirrored here.

Table of Contents

Overview

WOOPER, which stands for *Wrapper for Object-Oriented Programming in Erlang*, is a free software lightweight layer on top of the Erlang language that provides constructs dedicated to Object-Oriented Programming (OOP).

WOOPER is a rather autonomous part of the Ceylan project.

At least a basic knowledge of Erlang is expected in order to use WOOPER.

Understanding WOOPER in Two Steps

Here is a class definition, and here is an example of code using it. That's it!

Now, let's discuss a bit more in-depth of these subjects.

Motivations & Purpose

Some problems may almost only be tackled efficiently thanks to an object-oriented modelling.

The set of code and conventions proposed here allows to benefit from all the main OOP features (including polymorphism, life cycle management, state management and multiple inheritance) directly from Erlang (which natively does not rely on the OOP paradigm), so that - in the cases where it makes sense - an object-oriented approach at the implementation level can be easily achieved.

The WOOPER Mode of Operation in a Nutshell

The WOOPER OOP concepts translate into Erlang constructs according to the following mapping:

WOOPER base concept	**Corresponding mapping to Erlang**
class definition	module (typically compiled in a `.beam` file)
instance	process
instance reference	process identifier (PID)
new operators	WOOPER-provided functions, making use of user-defined `construct/N` functions (a.k.a. the constructors)
delete operator	WOOPER-provided function, making use of any user-defined `destruct/1` (a.k.a. the destructor)
method definition	module function that respects some conventions
method invocation	sending of an appropriate inter-process message
method look-up	class-specific virtual table taking into account inheritance transparently
instance state	instance-specific datastructure storing its attributes, and kept by the instance-specific WOOPER tail-recursive infinite loop

... continued on next page

WOOPER base concept	Corresponding mapping to Erlang
instance attributes	key/value pairs stored in the instance state
class (static) method	exported module function

In practice, developing a class with WOOPER mostly involves including the wooper.hrl header file and respecting the WOOPER conventions detailed below.

Example

Here is a simple example of how a WOOPER class can be defined and used.

It shows `new/delete` operators, method calling (both request and oneway), and inheritance.

A cat is here a viviparous mammal, as defined below (this is a variation of our more complete class_Cat.erl example):

```
-module(class_Cat).

% Determines what are the mother classes of this class (if any):
-define(wooper_superclasses,[class_Mammal,class_ViviparousBeing]).

% Parameters taken by the constructor ('construct').
% They are here the ones of the Mammal mother class (the viviparous being
% constructor does not need any parameter) plus whisker color.
% These are class-specific data needing to be set in the constructor:
-define(wooper_construct_parameters,Age,Gender,FurColor,WhiskerColor).

% Declaring all variations of WOOPER standard life-cycle operations:
% (this is just a pasted template, with updated arities)
-define( wooper_construct_export, new/4, new_link/4,
   synchronous_new/4, synchronous_new_link/4,
   synchronous_timed_new/4, synchronous_timed_new_link/4,
   remote_new/5, remote_new_link/5, remote_synchronous_new/5,
   remote_synchronous_new_link/5, remote_synchronous_timed_new/5,
   remote_synchronous_timed_new_link/5, construct/5, destruct/1 ).

% Member method declarations:
-define( wooper_method_export,getWhiskerColor/1,setWhiskerColor/2,
   canEat/2 ).

% Static method declarations:
-define( wooper_static_method_export, get_default_whisker_color()/0 ).

% Allows to define WOOPER base variables and methods for that class:
-include("wooper.hrl").
```

```
% Constructs a new Cat.
construct( State, ?wooper_construct_parameters ) ->
   % First the direct mother classes:
   MammalState = class_Mammal:construct( State, Age, Gender, FurColor ),
   ViviparousMammalState = class_ViviparousBeing:construct(MammalState),
   % Then the class-specific attributes; returns an updated state:
   setAttributes( ViviparousMammalState, whisker_color, WhiskerColor ).

destruct(State) ->
   io:format( "Deleting cat ~w! (overridden destructor)~n", [self()] ),
   State.

% Member methods.

% A cat-specific const request:
getWhiskerColor(State)->
   ?wooper_return_state_result( State, ?getAttr(whisker_color) ).

% A (non-const) oneway:
setWhiskerColor(State,NewColor)->
   NewState = setAttribute( State, whisker_color, NewColor ),
   ?wooper_return_state_only( NewState ).

% Overrides any request method defined in the Mammal class:
% (const request)
canEat(State,soup) ->
   ?wooper_return_state_result( State, true );

canEat(State,croquette) ->
   ?wooper_return_state_result( State, true );

canEat(State,meat) ->
   ?wooper_return_state_result( State, true );

canEat(State,_OtherFood) ->
   ?wooper_return_state_result( State, false ).

% Static method:
get_default_whisker_color() ->
   white.
```

Straightforward, isn't it? We will discuss it in-depth, though.

To test this class (provided that `GNU make` and `Erlang 20.0` or more recent are available in one's environment), one can easily install `Ceylan-WOOPER`, which depends on `Ceylan-Myriad`, hence to be installed first:

```
$ git clone https://github.com/Olivier-Boudeville/Ceylan-Myriad.git
$ cd Ceylan-Myriad && make all && cd ..
```

Then, as `Ceylan-Myriad` is known by WOOPER as the `Common` layer:

```
$ ln -s Ceylan-Myriad common
$ git clone https://github.com/Olivier-Boudeville/Ceylan-WOOPER.git
$ cd Ceylan-WOOPER && make all
```

Running the cat-related example just boils down to:

```
$ cd examples && make class_Cat_run
```

In the `examples` directory, the test defined in class_Cat_test.erl should run against the class defined in class_Cat.erl, and no error should be detected:

```
Running unitary test class_Cat_run (second form)
Erlang/OTP 20 [erts-9.0.1] [source] [64-bit] [smp:8:8] [..]
--> Testing module class_Cat_test.
[..]
Deleting cat <0.70.0>! (overridden destructor)
Deleting mammal <0.68.0>! (overridden destructor)
Actual class from destructor: class_Cat.
Deleting mammal <0.70.0>! (overridden destructor)
This cat could be created and be synchronously deleted, as expected.
--> Successful end of test.
(test finished, interpreter halted)
```

That's it!
Now, more in-depth explanations.

Why Adding Object-Oriented Capabilities To Erlang?

Although applying blindly an OOP approach while using languages based on other paradigms (Erlang ones are functional and concurrent; the language is not specifically targeting OOP) is a common mistake, there are some problems that may be deemed inherently "object-oriented", i.e. that cannot be effectively modelled without encapsulated abstractions sharing behaviours.

Examples of this kind of systems are multi-agent simulations. If they often need massive concurrency, robustness, distribution, etc. (Erlang is particularly suitable for that), the various types of agents have also often to largely share states and behaviours, while still being able to be further specialised on a per-type basis.

The example mentioned in this document is an illustration[1] of the interacting lives of numerous animals of various species. Obviously, they have to share behaviours (ex: all ovoviviparous beings may lay eggs, all creatures can live and die, all have an age, etc.), which cannot be mapped easily (read: automatically) to Erlang concepts without adding some generic constructs.

WOOPER, which stands for *Wrapper for OOP in Erlang*, is a lightweight yet effective (performance-wise, but also regarding the overall developing efforts) means of making these constructs available, notably in terms of state management and multiple inheritance.

The same programs could certainly be implemented without such OOP constructs, but at the expense of way too much manually-crafted, specific (per-class) code. This process would be tedious, error-prone, and most often the result could hardly be maintained.

[1] This example is not a *simulation*, it is just a multi-agent system. For real, massive, discrete-time simulations of complex systems in Erlang (using WOOPER), one may refer to Sim-Diasca.

How to Use WOOPER: Detailed Description & Concept Mappings

Classes

Classes & Names A class is a blueprint to create objects, a common scheme describing the state and behaviour of its instances, i.e. the attributes and methods that the created objects for that class all have.

With WOOPER, each class has a unique name, such as `class_Cat`.

To allow for **encapsulation**, a WOOPER class is mapped to an Erlang module, whose name is by convention made from the `class_` prefix followed by the class name, in the so-called CamelCase: all words are spelled in lower-case except their first letter, and there are no separators between words, like in: *ThisIsAnExample*.

For example, a class modeling a cat should translate into an Erlang module named `class_Cat`, thus in a file named `class_Cat.erl`. At the top of this file, the corresponding module would be therefore declared with: `-module(class_Cat).`.

Similarly, a pink flamingo class could be declared as `class_PinkFlamingo`, in `class_PinkFlamingo.erl`, which would include a `-module(class_PinkFlamingo).` declaration.

The class name can be obtained through its `get_classname/0` static method[2] (automatically defined by WOOPER):

```
> class_Cat:get_classname().
class_Cat
```

Note that a static method (i.e. a class method that does not apply to any specific instance) of a class X is nothing more than an Erlang function exported from the corresponding `class_X` module: all exported functions could be seen as static methods.

Inheritance & Superclasses A WOOPER class can inherit from other classes, in which case the state and behaviour defined in the mother classes are readily available to this child class.

Being in a **multiple inheritance** context, a given class can have any number (`[0..n]`) of direct mother classes, which themselves may have mother classes, and so on. This leads to a class hierarchy that forms a graph.

This is declared in WOOPER thanks to the `wooper_superclasses` define. For example, a class with no mother class should specify, once having declared its module:

```
-define(wooper_superclasses,[]).
```

As for our cat, this superb animal could be modelled both as a mammal (itself a specialised creature) and a viviparous being[3]. Hence its direct inheritance could be defined as:

```
-define(wooper_superclasses,[class_Mammal,class_ViviparousBeing]).
```

The superclasses (direct mother classes) of a given class can be known thanks to its `get_superclasses/0` static method:

```
> class_Cat:get_superclasses().
[class_Mammal,class_ViviparousBeing]
```

[2] The `get_classname/0` static method has no real interest of its own, it is defined mostly for explanation purpose.

[3] Neither of them is a subset of the other, these are mostly unrelated concepts, at least in the context of that example! (ex: a platypus is a mammal, but not a viviparous being).

Instances

Instance Mapping With WOOPER, which focuses on multi-agent systems, all **instances** of a class are mapped to Erlang processes (one WOOPER instance is exactly one Erlang process).

They are therefore, in UML parlance, *active objects* (each has its own thread of execution, they may apparently "live" simultaneously[4]).

Instance State Another common OOP need is to rely on **state management** and **encapsulation**: each instance should be stateful, have its state fully private, and be able to inherit automatically the data members defined by its mother classes.

In WOOPER, this is obtained thanks to a per-instance associative table, whose keys are the names of attributes and whose values are the attribute values. This will be detailed in the state management section.

[4]For some uses, such a concurrent feature (with *active* instances) may not be needed, in which case one may deal also with purely *passive* instances (as Erlang terms, not Erlang processes).

To anticipate a bit, instead of using `new/n` (returning the PID of a new process instance looping over its state), one may rely on `construct/n+1` (returning directly to the caller process that corresponding initial state, that can be then stored and interacted upon at will).

Methods

They can be either:

- **member methods**: to be applied to a specific *instance* (of a given class), like in: `MyCat ! declareBirthday`
- or **static methods**: general to a *class*, not targeting specifically an instance, like: `class_Cat:get_default_mew_duration()`

Unless specified otherwise, just mentioning *method* by itself refers to a *member method*. Static methods are discussed into their specific subsection.

Member methods can be publicly called by any process (be it WOOPER-based or not - provided of course it knows the PID of that instance), whether locally or remotely (i.e. on other networked computers, like with RMI or with CORBA, or directly from the same Erlang node), distribution (and parallelism) being seamlessly managed thanks to Erlang.

Member methods (either inherited or defined directly in the class) are mapped to specific Erlang functions, triggered by Erlang messages.

For example, our cat class may define, among others, following member methods (actual arities to be discussed later):

- `canEat`, taking one parameter specifying the type of food, and returning whether the corresponding cat can eat that kind of food; here the implementation should be cat-specific (i.e. specific to cats and also, possibly, specific to this very single cat), whereas the method signature shall be shared by all beings
- `getWhiskersColor`, taking no parameter, returning the color of its whiskers; this is indeed a purely cat-specific method, and different cats may be different whisker colors; as this method, like the previous one, returns a result to the caller, it is a *request* method
- `declareBirthday`, incrementing the age of our cat, not taking any parameter nor returning anything; it will be therefore be implemented as a *oneway* method (i.e. not returning any result to the caller, hence not even needing to know it), whose call is only interesting for its effect on the cat state: here, making it one year older
- `setWhiskerColor`, assigning the specified color to the whiskers of that cat instance, not returning anything (another oneway method, then)

Declaring a birthday is not cat-specific, nor mammal-specific: we can consider it being creature-specific. Cat instances should then inherit this method, preferably indirectly from the `class_Creature` class, in all cases without having to specify anything, since the `wooper_superclasses` define already implies it (implying one time for all that cats *are* creatures). Of course this inherited method may be overridden at will anywhere in the class hierarchy.

We will discuss the *definition* of these methods later, but for the moment let's determine their signatures and declarations, and how we are expected to *call* them.

Method Declaration The cat-specific member (i.e. non-static) methods are to be declared:

- in the `class_Cat` (defined as mentioned in `class_Cat.erl`)
- thanks to the `wooper_method_export` define (which, as expected, automatically exports these member methods)

Their arity should be equal to the number of parameters they should be called with, plus one that is automatically managed by WOOPER and corresponds to the (private) state of that instance.

This `State` variable defined by WOOPER can be somehow compared to the `self` parameter of Python, or to the `this` hidden pointer of C++. That state is automatically kept by WOOPER instances in their main loop, and automatically prepended, as first element, to the parameters of incoming method calls.

In our example, the declarations could therefore result in:

```
-define(wooper_method_export, canEat/2, getWhiskerColor/1,
        setWhiskerColor/2).
```

> **Note**
> In our example, `declareBirthday/1` will be inherited but not overridden (its base implementation being fine for cats as well), so it should not be listed among the `class_Cat` methods.

Some method names are reserved for WOOPER; notably no user method should have its name prefixed with `wooper`.

Method Invocation Let's suppose that the `MyCat` variable designates an instance of `class_Cat`. Then this `MyCat` reference is actually just the PID of the Erlang process hosting this instance.

All member methods (regardless of whether they are defined directly by the actual class or inherited) are to be called from outside this class thanks to a proper Erlang message, sent to the PID of the targeted instance.

When the method is expected to return a result (i.e. when it is a request method), the caller must specify in the corresponding message its own PID, so that the instance knows to whom the result should be sent.

Therefore the `self()` parameter in the call tuples below corresponds to the PID *of the caller*, while `MyCat` is bound to the PID *of the target instance*.

The three methods previously discussed would indeed be called that way:

```
% Calling the canEat request of our cat instance:
MyCat ! {canEat,soup,self()},
receive
    {wooper_result,true} ->
      io:format( "This cat likes soup!!!" );

    {wooper_result,false} ->
      io:format( "This cat does not seem omnivorous." )
```

```
end,

% A parameter-less request:
MyCat ! {getWhiskersColor,[],self()},
receive
    {wooper_result,white} ->
      io:format( "This cat has normal whiskers." );

    {wooper_result,blue} ->
      io:format( "What a weird cat..." )
end,

% A parameter-less oneway:
MyCat ! declareBirthday.
```

Method Name Methods are designated by their name (as an atom), as specified in the `wooper_method_export` define of the class in the inheritance tree that defines them.

The method name is always the first information given in the method call tuple.

Method Parameters All methods are free to change the state of their instance and possibly trigger any side-effect (ex: sending a message, writing a file, etc.).

As detailed below, there are two kinds of methods:

- *requests* methods: they shall return a result to the caller (obviously they need to know it, i.e. the caller has to specify its PID)
- *oneway* methods: no specific result are expected from them (hence no caller PID is to be specified)

Both can take any number of parameters, including none. As always, the **marshalling** of these parameters and, if relevant, of any returned value is performed automatically by Erlang.

Parameters are to be specified in a (possibly empty) list, as second element of the call tuple.

If only a single, non-list, parameter is needed, the list can be omitted, and the parameter can be directly specified: `Alfred ! {setAge,31}.` works just as well as `Alfred ! {setAge,[31]}.`.

Note

This cannot apply if the unique parameter is a list, as this would be ambiguous. For example: `Foods = [meat,soup,croquette], MyCat ! {setFavoriteFoods,Foods}` would result in a call to `setFavoriteFoods/4`, i.e. a call to `setFavoriteFoods(State,meat,soup,croquette)`, whereas the intent of the programmer is probably to call a `setFavoriteFoods/2` method like `setFavoriteFoods(State,Foods) when is_list(Foods) -> [..]`.
The proper call would then be `MyCat ! {setFavoriteFoods,[Foods]}`, i.e. the parameter list should be used, and it would then contain only one element, the food list, whose content would therefore be doubly enclosed.

Two Kinds of Methods

Request Methods A **request** is a method that returns a result to the caller.

For an instance to be able to send an answer to a request triggered by a caller, of course that instance needs to know the caller PID.

Therefore requests have to specify, as the third element of the call tuple, an additional information: the PID to which the answer should be sent, which is almost always the caller (hence the `self()` in the actual calls).

So these three potential information (request name, parameters, reference of the sender - i.e. an atom, usually a list, and a PID) are gathered in a triplet (a 3-tuple) sent as a message: `{request_name,[Arg1,Arg2,..],self()}`.

If only one parameter is to be sent, and if that parameter is not a list, then this can become `{request_name,Arg,self()}`.

For example:

```
MyCat ! {getAge,[],self()}
```

or:

```
Douglas ! {askQuestionWithHint,[{meaning_of,"Life"},{maybe,42}],self()}
```

or:

```
MyCalculator ! {sum,[[1,2,4]],self()}.
```

The actual result `R`, as determined by the method, is sent back as an Erlang message, which is a `{wooper_result,R}` pair, to help the caller pattern-matching the WOOPER messages in its mailbox.

`receive` should then be used by the caller to retrieve the request result, like in the case of this example of a 2D point instance:

```
MyPoint ! {getCoordinates,[],self()},
receive
   {wooper_result,[X,Y]} ->
       [..]
end,
[..]
```

Oneway Methods A **oneway** is a method that does not return a result to the caller.

When calling oneway methods, the caller does not have to specify its PID, as no result is expected to be returned back to it.

If ever the caller sends by mistake its PID nevertheless, a warning is sent back to it, the atom `wooper_method_returns_void`, instead of `{wooper_result,Result}`.

The proper way of calling a oneway method is to send to it an Erlang message that is:

- either a pair, i.e. a 2-element tuple (therefore with no PID specified): `{oneway_name,[Arg1,Arg2,..]}` or `{oneway_name,Arg}` if `Arg` is not a list; for example: `MyPoint ! {setCoordinates,[14,6]}` or `MyCat ! {setAge,5}`
- or, if the oneway does not take any parameter, just the atom `oneway_name`. For example: `MyCat ! declareBirthday`

No return should be expected (the called instance does not even know the PID of the caller), so no receive should be attempted on the caller side, unless wanting to wait until the end of time.

Due to the nature of oneways, if an error occurs instance-side during the call, the caller will never be notified of it.

However, to help the debugging, an error message is then logged (using `error_logger:error_msg`) and the actual error message, the one that would be sent back to the caller if the method was a request, is given to `erlang:exit` instead.

Method Results

Execution Success: `{wooper_result,ActualResult}` If the execution of a method succeeded, and if the method is a request, then `{wooper_result,ActualResult}` will be sent back to the caller (precisely: to the process whose PID was specified in the call triplet).

Otherwise one of the following error messages will be emitted.

Execution Failures When the execution of a method fails, three main error results can be output (as a message for requests, as a log for oneways).

A summary could be:

Error Result	Interpretation	Likely Guilty
`wooper_method_not_found`	No such method exists in the target class.	Caller
`wooper_method_failed`	Method triggered a runtime error (it has a bug).	Called instance
`wooper_method_faulty_return`	Method does not respect the WOOPER return convention.	Called instance

Note

More generally, failure detection may better be done through the use of (Erlang) links, either explicitly set (with `erlang:link/1`) or, preferably (ex: to avoid race conditions), with a linked variation of the `new` operator (ex: `new_link/n`), discussed later in this document.

`wooper_method_not_found`

The corresponding error message is `{wooper_method_not_found, InstancePid, Classname, MethodName, MethodArity, ListOfActualParameters}`.

For example `{wooper_method_not_found, <0.30.0>, class_Cat, layEggs, 2, ...}`.

Note that `MethodArity` includes the implied state parameter (that will be discussed later), i.e. here `layEggs/2` might be defined as `layEggs(State,NumberOfNewEggs) -> [..]`.

This error occurs whenever a called method could not be found in the whole inheritance graph of the target class. It means this method is not implemented, at least not with the deduced arity.

More precisely, when a message `{method_name,[Arg1,Arg2,..,ArgN]...}` (request or oneway) is received, `method_name/N+1` has be to called: WOOPER tries to find `method_name(State,Arg1,..,ArgN)`, and the method name and arity must match.

If no method could be found, the `wooper_method_not_found` atom is returned (if the method is a request, otherwise the error is logged), and the object state will not change, nor the instance will crash, as this error is deemed a caller-side one (i.e. the instance has a priori nothing to do with the error).

`wooper_method_failed`

The corresponding error message is `{wooper_method_failed, InstancePid, Classname, MethodName, MethodArity, ListOfActualParameters, ErrorTerm}`.

For example, `{wooper_method_failed, <0.30.0>, class_Cat, myCrashingMethod, 1, [], {{badmatch,create_bug}, [..]]}`.

If the exit message sent by the method specifies a PID, it is prepended to ErrorTerm.

Such a method error means there is a runtime failure, it is generally deemed a instance-side issue (the caller should not be responsible for it, unless it sent incorrect parameters), thus the instance process logs that error, sends an error term to the caller (if and only if it is a request), and then exits with the same error term.

`wooper_method_faulty_return`

The corresponding error message is `{wooper_method_faulty_return, InstancePid, Classname, MethodName, MethodArity, ListOfActualParameters, ActualReturn}`.

For example, `{wooper_method_faulty_return, <0.30.0>, class_Cat, myFaultyMethod, 1, [], [{{state_holder,..]}`.

This error occurs only when being in debug mode.

The main reason for this to happen is when debug mode is set and when a method implementation did not respect the expected method return convention (neither the `wooper_return_state_result` macro nor the `wooper_return_state_only` one was used in this method clause).

It means the method is not implemented correctly (it has a bug), or that it was not (re)compiled with the proper debug mode, i.e. the one the caller was compiled with.

This is an instance-side failure (the caller has no responsibility for that), thus the instance process logs that error, sends an error term to the caller (if and only if it is a request), and then exits with the same error term.

Caller-Side Error Management

As we can see, errors can be better discriminated if needed, on the caller side. Therefore one could make use of that information, as in:

```
MyPoint ! {getCoordinates,[],self()},
receive
    {wooper_result, [X,Y] } ->
      [..];
    {wooper_method_not_found, Pid, Class, Method, Arity, Params} ->
      [..];
    {wooper_method_failed, Pid, Class, Method, Arity, Params, ErrorTerm} ->
      [..];
    % Error term can be a tuple {Pid,Error} as well, depending on the exit:
    {wooper_method_failed, Pid, Class, Method, Arity, Params, {Pid,Error}}
      [..];
    {wooper_method_faulty_return, Pid, Class, Method, Arity, Params, Unexpe
      [..];
    wooper_method_returns_void ->
      [..];
    OtherError ->
      % Should never happen:
      [..]
end.
```

However defensive development is not really favoured in Erlang, one may let the caller crash on unexpected return instead. Therefore generally one may rely simply on matching the message sent in case of success[5]:

```
MyPoint ! {getCoordinates,[],self()},
receive
    {wooper_result, [X,Y] } ->
      [..]
end,
[..]
```

Method Definition Here we reverse the point of view: instead of **calling** a method, we are in the process of **implementing** a callable one.

A method signature has always for first parameter the state of the instance, for example: `getAge(State) -> [..]`, or `getCoordinate(State,Index) -> [..]`.

For the sake of clarity, this variable should preferably always be named `State`.

A method must always return at least the newer instance state, even if the state did not change.

[5] In which case, should a failure happen, the method call will become blocking.

In this case the initial state parameter is directly returned, as is, like in:

```
getWhiskerColor(State) ->
  ?wooper_return_state_result(State,?getAttr(whisker_color) ).
```

State is unchanged here.

Note that when a method "returns" the state of the instance, it returns it to the (local, process-wise) private WOOPER-based main loop of that instance: in other words, the state variable is *never* exported/sent/visible outside of its process (unless of course a developer writes a specific method for that).

Encapsulation is ensured, as the instance is the only process able to access its own state. On method ending, the instance then just loops again, with its updated state: that new state will be the base one for the next call, and so on.

One should therefore see each WOOPER instance as primarily a process executing a main loop that keeps the current stat of that instance:

- it is waiting idle for any incoming (WOOPER) message
- when such a message is received, based on the actual class of the instance and on the method name specified in the call, the appropriate function defined in the appropriate module is selected by WOOPER, taking into account the inheritance graph (actually a direct per-class mapping, somewhat akin to the C++ virtual table, was already determined at start-up, for better performances)
- then this function is called with the appropriate parameters (those of the call, in addition to the internally kept current state)
- if the method is a request, the specified result is sent back to the caller
- then the instance loops again, on a state possibly updated by this method call

Thus the caller will only receive the **result** of a method, if it is a request. Otherwise, i.e. with oneways, nothing is sent back (nothing can be, anyway).

More precisely, depending on its returning a specific result, the method signature will correspond either to the one of a request or of a oneway, and will use in its body, respectively, either the `wooper_return_state_result` or the `wooper_return_state_only` macro to ensure that a state *and* a result are returned, or just a state.

A good practise is to add a comment to each method definition, and to specify whether it is a request or a oneway, if it is a `const` method, etc. For example, the previous method could be best written as:

```
% Returns the current color of the whiskers of that cat instance.
% (const request)
getWhiskerColor(State) ->
   ?wooper_return_state_result(State, ?getAttr(whisker_color)).
```

Note

When a constructor or a method determines that a fatal error should be raised (for example because it cannot find a required registered process), it should use `throw`, like in: `throw({invalid_value,V})`. Using `exit` is supported but not recommended.

For Requests Requests will use `?wooper_return_state_result(NewState,Result)`: the new state will be kept by the instance, whereas the result will be sent to the caller. Hence `wooper_return_state_result` means that the method returns a state **and** a result.

For example a const request will return an unchanged state, and thus will be just useful for its result (and possible side-effects):

```
getAge(State) ->
   ?wooper_return_state_result(State,?getAttr(age)).
```

All methods are of course given the parameters specified at their call.
For example, we can declare:

```
giveBirth(State,NumberOfMaleChildren,NumberOfFemaleChildren) ->
   [..]
```

And then we may call it, in the case of a cat having 2 male kitten and 3 female ones, with:

```
MyCat ! {giveBirth,[2,3],self()}.
```

Requests can access to one more information than oneways: the PID of the caller that sent the request. As WOOPER takes care automatically of sending back the result to the caller, having the request know explicitly the caller is usually not useful, thus the caller PID does not appear explicitly in request signatures, among the actual parameters.

However WOOPER keeps track of this information, which remains available to requests.

The caller PID can indeed be retrieved from a request body by using the `getSender` macro, which is automatically managed by WOOPER:

```
giveBirth(State,NumberOfMaleChildren,NumberOfFemaleChildren) ->
  CallerPID = ?getSender(),
  [..]
```

Thus a request has natively access to its caller PID, i.e. with no need to specify it in the parameters as well as in the third element of the call tuple; so, instead of having to define:

```
MyCat ! {giveBirth,[2,3,self()],self()}
```

one can rely on only:

```
MyCat ! {giveBirth,[2,3],self()}
```

while still letting the possibility for the called request (here `giveBirth/3`, for a state and two parameters) to access the caller PID thanks to the `getSender` macro, and maybe store it for a later use or do anything appropriate with it.

Note that having to handle explicitly the caller PID is rather uncommon, as WOOPER takes care automatically of the sending of the result back to the caller.

The `getSender` macro should only be used for requests, as of course the sender PID has no meaning in the case of oneways.

If that macro is called nevertheless from a oneway, then it returns the atom `undefined`.

For Oneways Oneway will rely on the `?wooper_return_state_only(NewState)` macro: the instance state will be updated, but no result will be returned to the caller, which is not even known.

For example:

```
setAge(State,NewAge) ->
  ?wooper_return_state_only( setAttribute(State,age,NewAge) ).
```

This oneway can be called that way:

```
MyCat ! {setAge,4}.
% No result to expect.
```

Oneways may also be `const`, i.e. leave the state unchanged, only being called for side-effects, for example:

```
displayAge(State) ->
  io:format("My age is ~B~n.",[ ?getAttr(age) ]),
  ?wooper_return_state_only(State).
```

Usefulness Of These Two Return Macros The definition of the `wooper_return_state_result` and `wooper_return_state_only` macros is actually quite simple; they are just here to structure the method implementations (helping the method developer not mixing updated states and results), and to help ensuring, in debug mode, that methods return well-formed information: an atom is then prepended to the returned tuple and WOOPER matches it during post-invocation, before handling the return, for an increased safety.

For example, in debug mode, `?wooper_return_state_result(AState,AResult)` will simply translate into `{wooper_result,AState,AResult}`, and when the execution of the method is over, the WOOPER main loop of this instance will attempt to match the method returned value with that triplet.

Similarly, `?wooper_return_state_only(AState)` will translate into `{wooper_result,AState}`.

If not in debug mode, then the `wooper_result` atom will not even be added in the returned tuples; for example `?wooper_return_state_result(AState,AResult)` will just be `{AState,AResult}`.

Performances should increase a bit, at the expense of a less safe checking of the values returned by methods.

The two `wooper_return_state_*` macros have been introduced so that the unwary developer does not forget that his requests are not arbitrary functions, that they should not only return a result but also a state, and that the order is always: first the state, then the result, not the other way round.

Type Specifications Although doing so is optional, WOOPER strongly recommends declaring type specifications as well (and provides suitable constructs for that), like in:

```
% Returns the current color of the whiskers of that cat instance.
% (const request)
-spec getWhiskerColor(wooper:state()) -> request_return(color()).
getWhiskerColor(State) ->
    ?wooper_return_state_result(State, ?getAttr(whisker_color)).
```

(of course the developer is responsible for the definition of the `color()` type here)

Similarly, the aforementioned `declareBirthday/1` oneway could be defined as:

```
% Declares the birthday of this creature: increments its age.
% (oneway)
-spec declareBirthday(wooper:state()) -> oneway_return().
declareBirthday(State) ->
   AgedState = setAttribute(State, age, ?getAttr(age)+1),
   ?wooper_return_state_ony(AgedState).
```

Self-Invocation: Calling a Method From The Instance Itself When implementing a method of a class, one may want to call other methods **of that same class** (have they been overridden or not).

For example, when developing a `declareBirthday/1` oneway of `class_Mammal` (which, among other things, is expected to increment the mammal age), you may want to perform a call to the `setAge/2` oneway (possibly introduced by an ancestor class like `class_Creature`, or possibly overridden directly in `class_Mammal`) on the current instance.

One could refer to this method respectively as a function exported by that ancestor (ex: called as `class_Creature:setAge(...)`) or that is local to the current module (a `setAge(...)` call designating then `class_Mammal:setAge/2`).

However, in the future, child classes of `class_Mammal` may be introduced (ex: `class_Cat`), and they might define their own version of `setAge/2`.

Instead of hardcoding which version of that method shall be called (like in the two previous cases, which establish statically the intended version to call), a developer may desire, if not expect, that, for a cat or for any specialised version thereof, `declareBirthday/1` calls automatically the "right" `setAge/2` method (i.e. the lastly overridden one in the inheritance graph). Possibly any `class_Cat:setAge/2` - not the version of `class_Creature` or `class_Mammal`.

Such an inheritance-aware call could be easily triggered asynchronously: a classical message-based method call directly addressed by an instance to itself could be used, like in `self()!{setAge,10}`, and (thanks to WOOPER) this would lead to executing the "right" version of that method.

If this approach may be useful when not directly needing from the method the result of the call and/or not needing to have it executed at once, in the general case one wants to have that possibly overridden method be executed *directly*, synchronously, and to obtain immediately the corresponding updated state and, if relevant, the associated output result.

Then one should call the WOOPER-defined `executeRequest/{2,3}` or `executeOneway/{2,3}` functions (or any variation thereof), depending on the type of the method to call.

These two helper functions behave quite similarly to the actual method calls that are based on the operator `!`, except that no target instance has to be specified (since it is by definition a call made by an instance to itself) and that no message exchange at all is involved: the method look-up is just performed through the inheritance hierarchy, the correct method is called with the specified parameters and the result is then directly returned.

More precisely, **executeRequest** is `executeRequest/2` or `executeRequest/3`, its parameters being the current state, the name of the request method, and, if needed,

the parameters of the called request, either as a list or as a standalone one.

`executeRequest` returns a pair made of the new state and of the result.

For example, for a request taking more than one parameter, or one list parameter:

```
{NewState,Result} = executeRequest(CurrentState, myRequestName,
                                   ["hello", 42])
```

For a request taking exactly one, non-list, parameter:

```
{NewState,NewCounter} = executeRequest(CurrentState,
                                       addToCurrentCounter, 78)
```

For a request taking no parameter:

```
{NewState,Sentence} = executeRequest(CurrentState, getLastSentence)
```

Regarding now **executeOneway**, it is either `executeOneway/2` or `executeOneway/3`, depending on whether the oneway takes parameters. If yes, they can be specified as a list (if there are more than one) or, as always, as a standalone non-list parameter.

`executeOneway` returns the new state.

For example, a oneway taking more than one parameter, or one list parameter:

```
NewState = executeOneway(CurrentState,say,[ "hello", 42 ])
```

For a oneway taking exactly one (non-list) parameter:

```
NewState = executeOneway(CurrentState,setAge,78)
```

For a oneway taking no parameter:

```
NewState = executeOneway(CurrentState,declareBirthday)
```

> **Note**
> As discussed previously, there are caller-side errors that are not expected to crash the instance. If such a call is performed directly from that instance (i.e. with one of the `execute*` constructs), then two errors will be output: the first, non-fatal for the instance, due to the method call, then the second, fatal for the instance, due to the failure of the `execute*` call. This is the expected behaviour, as here the instance plays both roles, the caller and the callee.

Finally, one can specify **explicitly** the class (of course belonging to the inheritance graph of that class) defining the version of the method that one wants to execute, by-passing the inheritance-aware overriding system.

For example, a method needing to call `setAge/2` from its body would be expected to use something like: `AgeState = executeOneway(State,setAge,NewAge)`.

If `class_Cat` overrode `setAge/2`, any cat instance would then call the overridden `class_Cat:setAge` method instead of the original `class_Creature:setAge`.

What if our specific method of `class_Cat` wanted, for any reason, to call the `class_Creature` version, now shadowed by an overridden version of it? In this case a `execute*With` function should be used.

These functions, which are `executeRequestWith/{3,4}` and `executeOnewayWith/{3,4}`, behave exactly as the previous `execute*` functions, except that they take an additional parameter (to be specified just after the state) that is the name of the mother class (direct or not) having defined the version of the method that we want to execute.

> **Note**
> This mother class does not have to have specifically defined or overridden that method: this method will just be called in the context of that class, as if it was an instance of the mother class rather than one of the actual child class.

In our example, we should thus use simply:

```
AgeState = executeOnewayWith(State,class_Creature,setAge,NewAge)
```

in order to call the `class_Creature` version of the `setAge/2` oneway.

Static Methods Static methods, as opposed to member methods, do not target specifically an instance, they are defined at the class level.

They thus do not operate on PID, they are just to be called thanks to their module name, exactly as any exported standard function.

Static methods are to be listed by the class developer thanks to the `wooper_static_method_export` define, like in:

```
-define( wooper_static_method_export, get_default_whisker_color/0,
         determine_croquette_appeal/1, foo_bar/1 ).
```

The static methods are automatically exported by WOOPER, so that they can be readily called from any context, as in:

```
PossibleColor = class_Cat:get_default_whisker_color(),
[..]
```

State Management

Principles We are discussing here about how an instance is to manage its inner state.

Its state is only directly accessible from inside the instance, i.e. from the body of its methods, whether they are inherited or not: the state of an instance is **private** (local to its process), and the outside can *only* access it through the methods defined by its class.

The state of an instance (corresponding to the one that is given by WOOPER as first parameter of all its methods, thanks to a variable conventionally named `State`) is simply defined as a **set of attributes**.

Each attribute is designated by a name, defined as an atom, and is associated to a mutable value, which can be any Erlang term.

The current state of an instance can be thought as a list of `{attribute_name,attribute_value}` pairs, like in:

```
[ {color,black}, {fur_color,sand}, {age,13}, {name,"Tortilla"} ].
```

State Implementation Details The conceptual attribute list is actually an associative table[6] (ultimately relying on the `map` datatype now; previously on our `hashtable` module), selected for genericity, dynamicity and efficiency reasons.

The hash value of a key (like the `age` key) is computed, to be used as an index in order to find the corresponding value (in the previous example, `13`) in the relevant bucket of the table.

The point is that this kind of look-up is performed in constant time on average, regardless of how many key/value pairs are stored in the table, whereas most dynamic data structures, like plain lists, would have look-up runtime costs that would increase with the number of pairs they contain, thus being possibly most often slower than their hashtable-based counterparts.

[6] A not so conclusive experiment relied on class-specific records being defined. This approach raises issues, for example at construction and destruction time where parent classes have to deal with record types different from their own. Moreover there is no guarantee that creating/destructing longer tuples is significantly more efficient than, say, updating a map (yet the memory footprint shall be lower).

Managing The State Of An Instance A set of functions allows to operate on these state variables, notably to read and write the attributes that they contain.

As seen in the various examples, method implementations will access (read/write) attributes stored in the instance state, whose original version (i.e. the state of the instance at the method beginning) is always specified as their first parameter, conventionally named `State`.

This current state can be then modified in the method, and a final state (usually an updated version of the initial one) will be returned locally to WOOPER, thanks to the final statement in the method, one of the two `wooper_return_state_*` macros.

Then the code (automatically instantiated by the WOOPER header in the class implementation) will loop again for this instance with this updated state, waiting for the next method call, which will possibly change again the state (and trigger side-effects), and so on.

One may refer to wooper.hrl for the actual definition of most of these WOOPER constructs.

Modifying State The `setAttribute/3` Function

Setting an attribute (creating and/or modifying it) should be done with the `setAttribute/3` function:

```
NewState = setAttribute(AState,AttributeName,NewAttributeValue)
```

For example, `AgeState = setAttribute(State,age,3)` will return a new state, bound to `AgeState`, exact copy of `State` (with all the attribute pairs equal) but for the `age` attribute, whose value will be set to 3.

Therefore, during the execution of a method, any number of states can be defined (ex: `State`, `InitialisedState`, `AgeState`, etc.) before all, but the one that is returned, are garbage-collected.

Note that the corresponding state duplication remains efficient both in terms of processing and memory, as the different underlying state structures (ex: `State` and `AgeState`) actually **share** all their terms except the one modified - thanks to the immutability of Erlang variables which allows to reference rather than copy, be these datastructures tables, records, or anything else.

In various cases, notably in constructors, one needs to define a series of attributes in a row, but chaining `setAttribute/3` calls with intermediate states that have each to be named is not really convenient.

A better solution is to use the `setAttributes/2` function (note the plural) to set a list of attribute name/attribute value pairs in a row.

For example:

```
ConstructedState = setAttributes(MyState,[{age,3},
                                          {whisker_color,white}])
```

will return a new state, exact copy of `MyState` but for the listed attributes, set to their respective specified value.

The `removeAttribute/2` Function

> **Note**
>
> The `removeAttribute/2` function is now deprecated and should not be used anymore.

This function was used in order to fully remove an attribute entry (i.e. the whole key/value pair).

This function is deprecated now, as we prefer defining all attributes once for all, at construction time, and never add or remove them dynamically: the good practise is just to operate on their value, which can by example be set to `undefined`, without having to deal with the fact that, depending on the context, a given attribute may or may not be defined.

For example `NewState = removeAttribute(State,an_attribute)` could be used, for a resulting state having no key corresponding to `an_attribute`.

Neither the `setAttribute*` variants nor `removeAttribute/2` can fail, regardless of the attribute being already existing or not.

Reading State The `hasAttribute/2` Function

> **Note**
> The `hasAttribute/2` function is now deprecated and should not be used anymore.

To test whether an attribute is defined, one could use the `hasAttribute/2` function: `hasAttribute(AState,AttributeName)`, which returns either `true` or `false`, and cannot fail.

For example, `true = hasAttribute(State,whisker_color)` matches if and only if the attribute `whisker_color` is defined in state `State`.

Note that generally it is a bad practice to define attributes outside of the constructor of an instance, as the availability of an attribute could then depend on the actual state, which is an eventuality generally difficult to manage reliably.

A better approach is instead to define all possible attributes directly from the constructor. They would then be assigned to their initial value and, if none is appropriate, they should be set to the atom `undefined` (instead of not being defined at all).

The `getAttribute/2` Function

Getting the value of an attribute is to be done with the `getAttribute/2` function:

```
AttributeValue = getAttribute(AState,AttributeName)
```

For example, `MyColor = getAttribute(State,whisker_color)` returns the value of the attribute `whisker_color` from state `State`.

The requested attribute may not exist in the specified state. In this case, a runtime error is issued.

Requesting a non-existing attribute triggers a bad match. In the previous example, should the attribute `whisker_color` not have been defined, `getAttribute/2` would return:

```
{key_not_found,whisker_color}
```

The `getAttr/2` Macro

Quite often, when having to retrieve the value of an attribute from a state variable, that variable will be named `State`, notably when using directly the original state specified in the method declaration.

Indeed, when a method needs a specific value, generally either this value was already available in the state it began with (then we can read it from `State`), or is computed in the course of the method, in which case that value is most often already bound to a variable, which can then be re-used directly rather than be fetched from a state.

In this case, the `getAttr/2` macro can be used: `?getAttr(whisker_color)` expands (literally) as `getAttribute(State,whisker_color)`, and is a tad shorter.

This is implemented as a macro so that the user remains aware that an implicit variable named `State` is then used.

The less usual cases where a value must be read from a state variable that is *not* the initial `State` one occur mostly when wanting to read a value from the updated state returned by a `execute*` function call. In this case the `getAttribute/2` function should be used.

Read-Modify-Write Operations Some additional helper functions are provided for the most common operations, to keep the syntax as lightweight as possible.

The `addToAttribute/3` Function

When having a numerical attribute, `addToAttribute/3` adds the specified number to the attribute.

To be used like in:

```
NewState = addToAttribute(State,AttributeName,Value)
```

For example:

```
MyState = addToAttribute(FirstState,a_numerical_attribute,6)
```

In `MyState`, the value of attribute `a_numerical_attribute` is increased of 6, compared to the one in `FirstState`.

Calling `addToAttribute/3` on a non-existing attribute will trigger a runtime error (`{key_not_found,AttributeName}`).

If the attribute exists, but no addition can be performed on it (i.e. if it is meaningless for the type of the current value), a `badarith` runtime error will be issued.

The `subtractFromAttribute/3` Function

When having a numerical attribute, `subtractFromAttribute/3` subtracts the specified number from the attribute.

To be used like in:

```
NewState = subtractFromAttribute(State,AttributeName,Value)
```

For example:

```
MyState = subtractFromAttribute(FirstState,a_numerical_attribute,7)
```

In `MyState`, the value of attribute `a_numerical_attribute` is decreased of 7, compared to the one in `FirstState`.

Calling `subtractFromAttribute/3` on a non-existing attribute will trigger a runtime error (`{key_not_found,AttributeName}`). If the attribute exists, but no subtraction can be performed on it (meaningless for the type of the current value), a `badarith` runtime error will be issued.

The `toggleAttribute/2` Function

Flips the value of the specified (supposedly boolean) attribute: when having a boolean attribute, whose value is either `true` or `false`, sets the opposite logical value to the current one.

To be used like in:

```
NewState = toggleAttribute(State,BooleanAttributeName)
```

For example:

```
NewState = toggleAttribute(State,a_boolean_attribute)
```

Calling `toggleAttribute/2` on a non-existing attribute will trigger a runtime error (`{key_not_found,AttributeName}`). If the attribute exists, but has not a boolean value, a `badarith` runtime error will be issued.

The `appendToAttribute/3` Function

The corresponding signature is `NewState = appendToAttribute(State,AttributeName,Element` when having a list attribute, appends specified element to the attribute list, in first position.

For example, if `a_list_attribute` was already set to `[see_you,goodbye]` in `State`, then after `NewState = appendToAttribute(State,a_list_attribute,hello)`, the `a_list_attribute` attribute defined in `NewState` will be equal to `[hello,see_you,goodbye]`.

Calling `appendToAttribute/3` on a non-existing attribute will trigger a compile-time error. If the attribute exists, but is not a list, an ill-formed list will be created (ex: `[8|false]` when appending 8 to `false`, which is not a list).

With the hashtable-based version of WOOPER:

- if the target attribute does not exist, will trigger `{{badmatch,undefined},[{hashtable,appendToEr`
- if it exists but is not already a list, it will not crash but will create an ill-formed list (ex: `[8|false]` when appending 8 to `false`, which is not a list).

The `deleteFromAttribute/3` Function

The corresponding signature is `NewState = deleteFromAttribute(State,AttributeName,Eleme` when having a list attribute, deletes first match of specified element from the attribute list.

For example: `NewState = deleteFromAttribute(State,a_list_attribute,hello)`, with the value corresponding to the `a_list_attribute` attribute in `State` variable being `[goodbye,hello,cheers,hello,see_you]` should return a state whose `a_list_attribute` attribute would be equal to `[goodbye,cheers,hello,see_you]`, all other attributes being unchanged.

If no element in the list matches the specified one, no error will be triggered and the list will be kept as is.

Calling `deleteFromAttribute/3` on a non-existing attribute will trigger a compile-time error. If the attribute exists, but is not a list, a run-time error will be issued.

With the hashtable-based version of WOOPER:

- if the target attribute does not exist, will trigger `{{badmatch,undefined},[{hashtable,deleteFrom`
- if it exists but is not already a list, it will trigger `{function_clause,[{lists,delete,[..,..]},{ha`

The `popFromAttribute/2` Function

The corresponding signature is `{NewState,Head} = popFromAttribute(State,AttributeName)`: when having an attribute of type list, this function removes the head from the list and returns a pair made of the updated state (same state except that the corresponding list attribute has lost its head, it is equal to the list tail now) and of that head.

For example: `{NewState,Head} = popFromAttribute(State,a_list_attribute).` If the value of the attribute `a_list_attribute` was `[5,8,3]`, its new value (in `NewState`) will be `[8,3]` and `Head` will be bound to `5`.

The `addKeyValueToAttribute/4` Function

The corresponding signature is `NewState = addKeyValueToAttribute(State,AttributeName,Ke` when having an attribute whose value is a table, adds specified key/value pair to that table attribute.

For example: `TableState = setAttribute(State,my_table,table:new()),` `NewState = addKeyValueToAttribute(TableState,my_table,my_key,my_value)` will result in having the attribute `my_table` in state variable `TableState` being a table with only one entry, whose key is `my_key` and whose value is `my_value`.

Multiple Inheritance & Polymorphism

The General Case Both multiple inheritance and polymorphism are automatically managed by WOOPER: even if our cat class does not define a `getAge` method, it can nevertheless readily be called on a cat instance, as it is inherited from its mother classes (here from `class_Creature`, an indirect mother class).

Therefore all creature instances can be handled the same, regardless of their actual classes:

```
% Inherited methods work exactly the same as methods defined
% directly in the class:
MyCat ! {getAge,[],self()},
receive
  {wooper_result,Age} ->
    io:format( "This is a ~B year old cat.", [Age] )
end,

% Polymorphism is immediate:
% (class_Platypus inheriting too from class_Mammal,
% hence from class_Creature).
MyPetList = [MyCat,MyPlatypus],
foreach(
  fun(AnyCreature) ->
    AnyCreature ! {getAge,[],self()},
    receive
      {wooper_result,Age} ->
        io:format("This is a ~B year old creature.",[Age])
  end,
  MyPetList).
```

Running this code should output something like:

```
This is a 4 year old creature.
This is a 9 year old creature.
```

The point here is that the implementer does not have to know what are the actual classes of the instances that are interacted with, provided that they share a common ancestor: polymorphism allows to handle them transparently.

The Special Case of Diamond-Shaped Inheritance In the case of a diamond-shaped inheritance, as the method table is constructed in the order specified in the declaration of the superclasses, like in:

```
get_superclasses() ->
  [class_X,class_Y,...]).
```

and as child classes override mother ones, when an incoming WOOPER message arrives the selected **method** should be the one defined in the last inheritance branch of the last child (if any), otherwise the one defined in the next to last branch of the last child, etc.

Generally speaking, overriding in that case the relevant methods that were initially defined in the child class at the base of the diamond, in order that they perform explicitly a direct call to the wanted module, is by far the most reasonable solution, in terms of clarity and maintainability, compared to guessing which version of the method in the inheritance graph should be called.

Regarding the instance state, the **attributes** are set by the constructors, and the developer can select in which order the direct mother classes should be constructed.

However, in such an inheritance scheme, the constructor of the class that sits at the top of a given diamond will be called multiple times.

Any side-effect that it would induce would then occur as many times as this class is a common ancestor of the actual class; it may be advisable to create idempotent constructors in that case.

> **Note**
>
> More generally speaking, diamond-shaped inheritance is seldom necessary. More often than not, it is the consequence of a less-than-ideal OOP design, and should be avoided anyway.

Life-Cycle

Basically, creation and destruction of instances are managed respectively thanks to the `new`/`new_link` and the `delete` operators (all these operators are WOOPER-reserved function names, for all arities), like in:

```
MyCat = class_Cat:new(Age,Gender,FurColor,WhiskerColor),
MyCat ! delete.
```

Instance Creation: `new`/`new_link` And `construct`

Role of a `new` /`construct` Pair Whereas the purpose of `new` / `new_link` is to create a working instance on the user's behalf, the role of `construct` is to initialise an instance of that class while being able to be chained for inheritance, as explained later.

Such an initialisation is of course part of the instance creation: all calls to any of the"new" operators result in an underlying call to the (single) corresponding `construct` operator.

For example, both creations stemming from `MyCat = class_Cat:new(A,B,C,D)` and `MyCat = class_Cat:new_link(A,B,C,D)` will rely on `class_Cat:construct/5` to set up a proper initial state for the `MyCat` instance; the same `class_Cat:construct(State,A,B,C,D)` will be called for all creation cases.

The `new_link` operator behaves exactly as the `new` operator, except that it creates an instance that is Erlang-linked with the process that called that operator, exactly like `spawn_link` behaves compared to `spawn`[7].

The `new` and `new_link` operators are automatically defined by WOOPER, but they rely on the class-specific user-defined `construct` operator (only WOOPER is expected to make use of it). This `construct` operator is the one that must be implemented by the class developer (the machinery related to `new` being managed by WOOPER).

Currently a single `construct` operator can be defined, i.e. a single arity is supported[8], possibly with multiple clauses that, as usual, are selected based on pattern-matching.

For example:

```
% Selection based on pattern-matching:
MyFirstDog  = Class_Dog:new(create_from_weight,4.4),
MySecondDog = Class_Dog:new(create_from_colors,[sand,white]).
```

The Various Ways of Creating An Instance As shown with the `new_link` operator, even for a given set of construction parameters, many variations of `new` can be of use: linked or not, synchronous or not, with a time-out or not, on current node or on a user-specified one, etc.

For a class whose instances can be constructed from `N` actual parameters, the following construction operators, detailed in the next section, are built-in:

[7] For example it induces no race condition between linking and termination in the case of a very short-lived spawned process.

[8] Even if generally workarounds can easily be devised (for example by tagging construction parameters with atom so that a single arity can federate all cases), this limitation is planned to be removed.

- if instance is to be created on the **local** node:
  - non-blocking creation: `new/N` and `new_link/N`
  - blocking creation: `synchronous_new/N` and `synchronous_new_link/N`
  - blocking creation with time-out: `synchronous_timed_new/N` and `synchronous_timed_new_link/N`
- if instance is to be created on any specified **remote** node:
  - non-blocking creation: `remote_new/N+1` and `remote_new_link/N+1`
  - blocking creation: `remote_synchronous_new/N+1` and `remote_synchronous_new_link/N`
  - blocking creation with time-out: `remote_synchronous_timed_new/N+1` and `remote_synchronous_timed_new_link/N+1`

> **Note**
> All `remote_*` variations require one more parameter (to be specified first), since the remote node on which the instance should be created has of course to be specified.

All supported `new` variations are detailed below.

Asynchronous new

This corresponds to the plain `new`, `new_link` operators etc. discussed earlier, relying internally on the usual `spawn*` primitives . These basic operators are **asynchronous** (non-blocking): they trigger the creation of a new instance, and return immediately, without waiting for it to complete, and the execution of the calling process continues while (hopefully, i.e. with no guarantee) the instance is being created and executed.

Synchronous new

As mentioned, with the previous asynchronous forms, the caller has no way of knowing when the spawned instance is up and running (if it ever happens).

Thus two counterpart operators, `synchronous_new/synchronous_new_link` are also available.

They behave like `new/new_link` except they will return only when (and if) the created instance is up and running: they are blocking, synchronous, operators.

For example, after `MyMammal = class_Mammal:synchronous_new(...)`, one knows that the `MyMammal` instance is fully created and waiting for incoming messages.

The implementation of these synchronous operations relies on a message (precisely: `{spawn_successful,InstancePid}`) being automatically sent by the created instance to the WOOPER code on the caller side, so that the `synchronous_new` operator will return to the user code only once successfully constructed and ready to handle messages.

Timed Synchronous new

Note that, should the instance creation fail, the caller of a synchronous new would then be blocked for ever, as the awaited message would actually never be sent by the failed new instance.

This is why the `synchronous_timed_new*` operators have been introduced: if the caller-side time-out (whose default duration is 5 seconds) expires while waiting for the created instance to answer, then they will throw an appropriate exception, among:

- `{synchronous_time_out,InstanceModule}`
- `{synchronous_linked_time_out,InstanceModule}`
- `{remote_synchronous_time_out,Node,InstanceModule}`
- `{remote_synchronous_linked_time_out,Node,InstanceModule}`
- `{synchronous_time_out,InstanceModule}`
- `{synchronous_linked_time_out,InstanceModule}`
- `{remote_synchronous_time_out,Node,InstanceModule}`
- `{remote_synchronous_linked_time_out,Node,InstanceModule}`

Then the caller may or may not catch this exception.

Remote new

Exactly like a process might be spawned on another Erlang node, a WOOPER instance can be created on any user-specified available Erlang node.

To do so, the `remote_*new*` variations shall be used. They behave exactly like their local counterparts, except that they take an additional information, as first parameter: the node on which the specified instance must be created.

For example:

```
MyCat = class_Cat:remote_new(TargetNode, Age, Gender,
                             FurColor, WhiskerColor).
```

Of course:

- the remote node must be already existing
- the current node must be able to connect to it (shared cookie)
- all modules that the instance will make use of must be available on the remote node, including the ones of all relevant classes (i.e. the class of the instance but also its whole class hierarchy)

All variations of the `new` operator are always defined automatically by WOOPER: nothing special is to be done for them, provided of course that the constructor they all rely on has been defined.

Some Examples Of Instance Creation Knowing that a cat can be created out of four parameters (Age, Gender, FurColor, WhiskerColor), various cat instances could be created thanks to:

```
% Local asynchronous creation:
MyFirstCat = class_Cat:new(1,male,brown,white),

% The same, but a crash of this cat will crash the current process too:
MySecondCat = class_Cat:new_link(2,female,black,white),

% This cat will be created on OtherNode, and the call will return only
% once it is up and running or once the creation failed. As moreover the
% cat instance is linked to the instance process, it may crash this
% calling process:
MyThirdCat = class_Cat:remote_synchronous_timed_new_link(OtherNode,3,
  male,grey,black),
[..]
```

Definition of the `construct` Operator Each class must define its `construct/N` operator, whose role is to fully initialise, based on the specified construction parameters, the state of new instances in compliance with the class inheritance - regardless of the `new` variation being used.

In the context of class inheritance, the `construct` operators are expected to be chained: they must be designed to be called by the ones of their child classes, and in turn they must call themselves the constructors of their direct mother classes, if any.

Hence they always take the current state of the instance being created as a starting base, and returns it once updated, first from the direct mother classes, then by this class itself.

For example, let's suppose `class_Cat` inherits directly both from `class_Mammal` and from `class_ViviparousBeing`, has only one attribute (`whisker_color`) of its own, and that a new cat is to be created out of three pieces of information:

```
[..]
get_superclasses() ->
  [class_Mammal,class_ViviparousBeing].

[..]
get_attributes() ->
  [whisker_color].

% Constructs a new Cat.
construct(State,Gender,FurColor,WhiskerColor) ->
  % First the (chained) direct mother classes:
  MammalState = class_Mammal:construct(State,_Age=0,Gender,FurColor),
  ViviparousMammalState = class_ViviparousBeing:construct(MammalState),
  % Then the class-specific attributes:
  setAttribute(ViviparousMammalState,whisker_color,WhiskerColor).
```

The fact that the `Mammal` class itself inherits from the `Creature` class does not have to appear here: it is to be managed directly by `class_Mammal:construct` (at any given inheritance level, only direct mother classes must be taken into account).

One should ensure that, in constructors, the successive states are always built from the last updated one, unlike:

```
% WRONG, the age update is lost:
construct(State,Age,Gender) ->
  AgeState = setAttribute(State,age,Age),
  % AgeState should be used here, not State:
  setAttribute(State,gender,Gender),
```

This would be correct:

```
% RIGHT but a bit clumsy:
construct(State,Age,Gender) ->
  AgeState = setAttribute(State,age,Age),
  setAttribute(AgeState,gender,Gender).
```

Recommended form:

```
% BEST:
construct(State,Age,Gender) ->
  setAttributes( State, [ {age,Age}, {gender,Gender} ]).
```

Note

There is no strict relationship between construction parameters and instance attributes, neither in terms of cardinality, type or value.
For examples, attributes could be set to default values, a point could be created from an angle and a distance but its actual state may consist on two cartesian coordinates instead, etc.
Therefore both have to be defined by the class developer, and, in the general case, attributes cannot be inferred from construction parameters.

Finally, a class can define multiple clauses for its constructor: the proper one will be called based on the pattern-matching performed on these parameters.

Instance Deletion

Automatic Chaining Of Destructors We saw that, when implementing a constructor (`construct/N`), like in all other OOP approaches the constructors of the direct mother classes have to be explicitly called, so that they can be given the proper parameters, as determined by the class developer.

Conversely, with WOOPER, when defining a destructor for a class (`destruct/1`), one only has to specify what are the *specific* operations and state changes (if any) that are required so that an instance of that class is deleted: the proper calling of the destructors of mother classes across the inheritance graph is automatically taken in charge by WOOPER.

Once the user-specified actions have been processed by the destructor (ex: releasing a resource, unsubscribing from a registry, deleting other instances, closing properly a file, etc.), it is expected to return an updated state, which will be given to the destructors of the instance superclasses.

WOOPER will automatically make use of any user-defined destructor, otherwise the default one will be used, doing nothing (i.e. returning the exact same state that it was given).

Asynchronous Destruction: using `destruct/1` More precisely, either the class implementer does not define at all a `destruct/1` operator (and therefore uses the default do-nothing destructor), or it defines it explicitly, like in:

```
destruct(State) ->
  io:format("An instance of class ~w is being deleted now!",[?MODULE]),
  % Quite often the destructor does not need to modify the state of
  % the instance:
  State.
```

In both cases (default or user-defined destructor), when the instance will be deleted (ex: `MyInstance ! delete` is issued), WOOPER will take care of:

- calling any destructor defined for that class
- then calling the ones of the direct mother classes, which will in turn call the ones of their mother classes, and so on

Note that the destructors for direct mother classes will be called in the reverse order of the one according to the constructors ought to have been called: if a class `class_X` declares `class_A` and `class_B` as mother classes (in that order), then in the `class_X:construct` definition the implementer is expected to call `class_A:construct` and then `class_B:construct`, whereas on deletion the WOOPER-enforced order of execution will be: `class_X:delete`, then `class_B:delete`, then `class_A:delete`, for the sake of symmetry.

Synchronous Destruction: using `synchronous_delete/1` WOOPER automatically defines as well a way of deleting *synchronously* a given instance: a caller can request a synchronous (blocking) deletion of that instance so that, once notified of the deletion, it knows for sure the instance does not exist anymore, like in:

```
InstanceToDelete ! {synchronous_delete,self()},
% Then the caller can block as long as the deletion did not occur:
receive
  {deleted,InstanceToDelete} ->
      doSomething()
end.
```

The class implementer does not have to do anything to support this feature, as the synchronous deletion is automatically built by WOOPER on top of the usual asynchronous one (both thus rely on `destruct/1`).

Miscellaneous Technical Points

`delete_any_instance_referenced_in/2`

When an attribute contains either a single instance reference (i.e. the PID of the corresponding process) or a list of instance references, this WOOPER-defined helper function will automatically delete (asynchronously) these instances, and will return an updated state in which this attribute is set to `undefined`.

This function is especially useful in destructors.

For example, if `State` contains:

- an attribute named `my_pid` whose value is the PID of an instance
- and also an attribute named `my_list_of_pid` containing a list of PID instances

and if the deleted instance took ownership of these instances, then:

```
delete(State) ->
 TempState = wooper:delete_any_instance_referenced_in(State,my_pid),
 wooper:delete_any_instance_referenced_in(TempState,my_list_of_pid).
```

will automatically delete all these instances (if any) and return an updated state.

Then the destructors of the mother classes can be chained by WOOPER.

See also the various other helpers defined in `wooper.erl`.

`EXIT` Messages

A class instance may receive EXIT messages from other processes.

A given class can process these EXIT notifications:

- either by defining and exporting the `onWOOPERExitReceived/3` oneway
- or by inheriting it

For example:

```
onWOOPERExitReceived(State,Pid,ExitType) ->
  io:format("MyClass EXIT handler ignored signal '~p'"
            " from ~w.~n", [ExitType,Pid]),
  ?wooper_return_state_only(State).
```

may result in an output like:

```
MyClass EXIT handler ignored signal 'normal' from <0.40.0>.
```

If no class-specific EXIT handler is available, the default WOOPER one will be used.

It will just notify the signal to the user, by displaying a message like:

```
WOOPER default EXIT handler for instance <0.36.0> of class class_Cat
  ignored signal 'normal' from <0.40.0>.
```

Monitors

Quite similarly to `EXIT` messages, monitors and `nodeup` / `nodedown` messages are also managed by WOOPER.

Type Specifications

We strongly promote at least the definition of types and function specifications, if not a very regular use of Dialyzer.

Albeit seldom mentioned here, WOOPER defines its own related type constructs in order to apply static typing at its level as well, like in:

```
-spec construct( wooper:state(), age(), gender() ) -> wooper:state().
-spec destruct( wooper:state() ) -> wooper:state().
-spec setAge( wooper:state(), age() ) -> oneway_return().
-spec canEat( wooper:state(), food() ) -> request_return( boolean() ).
```

Please refer to the test examples to better understand their actual use.

Guidelines

All WOOPER classes must include wooper.hrl:

```
-include("wooper.hrl").
```

To help declaring the right defines in the right order, using the WOOPER template is recommended.

One may also have a look at the full test examples, as a source of inspiration.

Source Editors

We use `Emacs` but of course any editor will be fine.

For Nedit users, a WOOPER-aware nedit.rc configuration file for syntax highlighting (on black backgrounds), inspired from Daniel Solaz's Erlang Nedit mode, is available.

Similarity With Other Languages

Finally, WOOPER is in some ways adding features quite similar to the ones available with other languages, including Python (simple multiple inheritance, implied `self/State` parameter, attribute dictionaries/associative tables, etc.; with less syntactic sugar available though) while still offering the major strengths of Erlang (concurrency, distribution, functional paradigm) and not hurting too much the overall performances (mainly thanks to the prebuilt attribute and method tables).

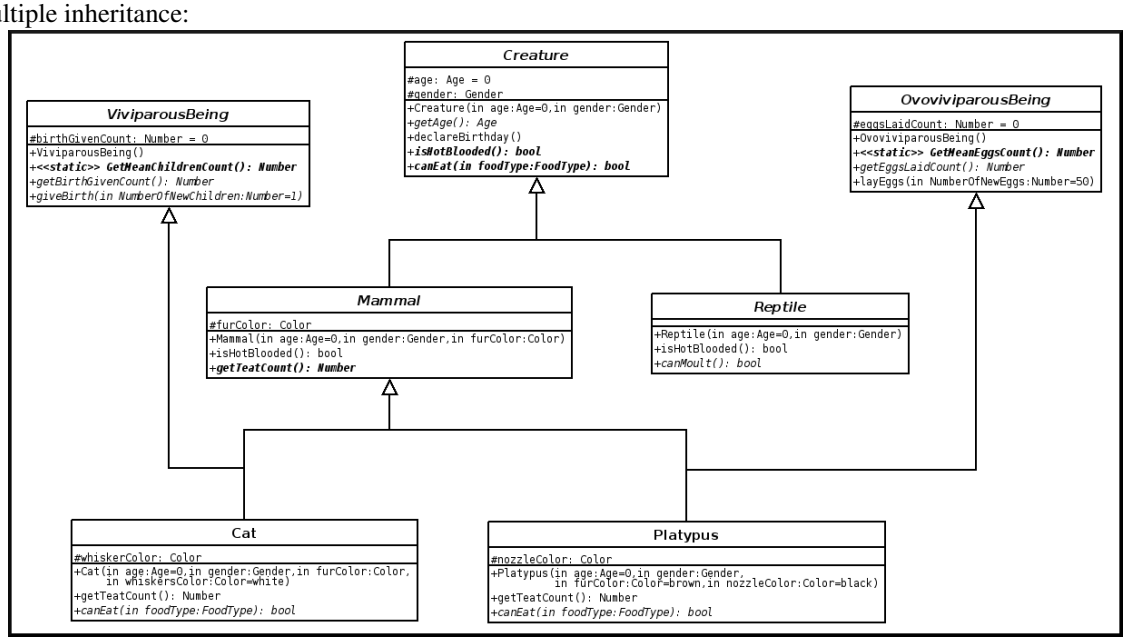
Actually the main implementation shortcomings that remain are:

- some syntactical elements are still too cumbersome (ex: the `wooper_construct_export` declaration, which moreover hinders from being able to declare constructors with various arities)
- the per-instance memory footprint could be reduced by sharing the "virtual table" of a given class between all its instances

Both of these limitations are to be removed over time thanks to metaprogramming (based on parse transforms).

WOOPER Example

We defined a small set of classes in order to serve as an example and demonstrate multiple inheritance:



Class implementations

- class_Creature.erl
- class_ViviparousBeing.erl
- class_OvoviviparousBeing.erl
- class_Mammal.erl
- class_Reptile.erl
- class_Cat.erl
- class_Platypus.erl

Tests

- class_Creature_test.erl
- class_ViviparousBeing_test.erl
- class_OvoviviparousBeing_test.erl
- class_Mammal_test.erl
- class_Reptile_test.erl
- class_Cat_test.erl
- class_Platypus_test.erl

To run a test (ex: `class_Cat_test.erl`), when WOOPER has already been compiled, one just has to enter: `make class_Cat_run`.

Good Practises

When using WOOPER, the following conventions are deemed useful to respect.

No warning should be tolerated in code using WOOPER, as we never found useless notifications.

All attributes of an instance should better be defined from the constructor, instead of being dynamically added during the life of the instance; otherwise the methods would have to deal with some attributes that may, or may not, be defined; if no proper value exists for an attribute at the creation of an instance, then its value should just be set to the atom `undefined`.

When a function or a method is defined in a WOOPER file, it should of course be commented, and, even if the information can be guessed from context and body, in the last line of the comments the type of the function should be specified (ex: `oneway`, `request`, `helper function`, etc.) possibly with qualifiers (ex: `const`), like in:

```
% Sets the current color.
% (oneway)
setColor(State,NewColor) ->
  [..]
```

or:

```
% Gets the current color.
% (const request)
getColor(State) ->
  [..]
```

Helper functions and static methods (which, from an Erlang point of view, are just exported functions) should be named like C functions (ex: `compute_sum`) rather than being written in CamelCase (ex: no helper function should be named `computeSum`), to avoid mixing up these different kinds of code.

To further separate helper functions from instance methods, an helper function taking a `State` parameter should better place it at the end of its parameter list rather than in first position (ex: `compute_sum(X,Y,State)` rather than `compute_sum(State,X,Y)`).

In a method body, the various state variables being introduced should be properly named, i.e. their name should start with a self-documenting prefix followed by the `State` suffix, like in: `SeededState = setAttribute(State,seed,{1,7,11})`.

Some more general (mostly unrelated) Erlang conventions that we like:

- when more than one parameter is specified in a fonction signature, parameter names can be surrounded by spaces (ex: `f( Color )`, or `g( Age, Height )`)
- functions should be separated by (at least) three newlines, whereas clauses for a given function should be separated exactly by one newline
- to auto-document parameters, a "mute" variable is preferably to be used: for example, instead of `f( Color, true )` use `f( Color, _Dither=true )`; however note that these mute variables are still bound and thus pattern-matched: for example, if multiple `_Dither` mute variables are bound in the same scope to different values, a bad match will be triggered at runtime.

Troubleshooting

General Case

Compilation Warnings A basic rule of thumb in all languages is to enable all warnings and eradicate them before even trying to test a program.

This is still more valid when using WOOPER, whose proper use should never result in any warning being issued by the compiler.

Notably warnings about unused variables are precious in order to catch mistakes when state variables are not being properly taken care of (ex: when a state is defined but never re-used later).

Runtime Errors Most errors while using WOOPER should result in relatively clear messages (ex: `wooper_method_failed` or `wooper_method_faulty_return`), associated with all relevant run-time information that was available to WOOPER.

Another way of overcoming WOOPER issues is to activate the debug mode for all WOOPER-enabled compiled modules (ex: uncomment `-define(wooper_debug,).` in `wooper.hrl`), and recompile your classes.

The debug mode tries to perform extensive checking on all WOOPER entry points, from incoming messages to the user class itself, catching mistakes from the class developer as well as from the class user.

For example, the validity of states returned by a constructor, by each method and by the destructor is checked, as the one of states specified to the `execute*` constructs.

If it is not enough to clear things up, an additional step can be to add, on a per-class basis (ex: in `class_Cat.erl`), before the WOOPER include, `-define(wooper_log_wanted,)`..

Then all incoming method calls will be traced, for easier debugging. It is seldom necessary to go till this level of detail.

As there are a few common WOOPER gotchas though, the main ones are listed below.

Mismatches In Method Call Oneway Versus Request Calls

One of these gotchas - experienced even by the WOOPER author - is to define a two-parameter oneway, whose second parameter is a PID, and to call this method wrongly as a request, instead of as a oneway.

For example, let's suppose the `class_Dog` class defines the oneway method `startBarkingAt/3` as:

```
startBarkingAt(State,Duration,ListenerPID) -> ...
```

The correct approach to call this **oneway** would be:

```
MyDogPid ! {startBarkingAt,[MyDuration,self()]}
```

An absent-minded developer could have written instead:

```
MyDogPid ! {startBarkingAt,MyDuration,self()}
```

This would have called a `request` method `startBarkingAt/2` (which could have been for example `startBarkingAt(State,TerminationOffset) -> ...`, the PID being interpreted by WOOPER as the request sender PID), a method that most probably does not even exist.

This would result in a bit obscure error message like `Error in process <0.43.0> on node 'XXXX' with exit value: {badarg,[{class_Dog,wooper_main_loop,1}]}`.

List Parameter Incorrectly Specified In Call

As explained in the Method Parameters section, if a method takes only one parameter and if this parameter is a list, then in a call this parameter cannot be specified as a standalone one: a parameter list with only one element, this parameter, should be used instead.

Error With Exit Value: `{undef,[{map_hashtable,new,[..]}..` You most probably forgot to build the `common` directory (a.k.a. `Ceylan-Myriad`) that contains, among other modules, the `map_hashtable.erl` source file.

Check that you have a `map_hashtable.beam` file indeed, and that it can be found from the paths specified to the virtual machine. Note that the WOOPER code designates this module as the `table` one (ex: `table:new()`), for a better substituability (this is obtained thanks to a parse-transform provided by `Ceylan-Myriad`)
.

Current Stable Version & Download

Using Stable Release Archive

Currently no source archive is specifically distributed, please refer to the following section.

Using Cutting-Edge GIT

We try to ensure that the main line (in the `master` branch) always stays functional. Evolutions are to be take place in feature branches.

This layer, `Ceylan-WOOPER`, relies (only) on:

- Erlang, version 20.2 or higher
- the `Ceylan-Myriad` base layer

We prefer using GNU/Linux, sticking to the latest stable release of Erlang, and building it from sources, thanks to GNU `make`.

For that we devised the install-erlang.sh script; a simple use of it is:

```
$ ./install-erlang.sh --doc-install --generate-plt
```

One may execute `./install-erlang.sh --help` for more details about how to configure it, notably in order to enable all modules of interest (`crypto`, `wx`, etc.) even if they are optional in the context of WOOPER.

As a result, once a proper Erlang version is available, the Ceylan-Myriad repository should be cloned and built, before doing the same with the Ceylan-WOOPER repository, like in:

```
$ git clone https://github.com/Olivier-Boudeville/Ceylan-Myriad
$ cd Ceylan-Myriad && make all && cd ..
# WOOPER knows this package as the 'Common' layer:
$ ln -s Ceylan-Myriad common
$ git clone https://github.com/Olivier-Boudeville/Ceylan-WOOPER
$ cd Ceylan-WOOPER && make all
```

Version History & Changes

As mentioned previously, in the future a version of WOOPER making heavy use of parse-transforms will be distributed.

Version 2.0 [cutting-edge, not available]

Not released yet (work-in-progress).

Version 1.0 [current stable]

Countless improvements have been integrated in the course of the use of WOOPER, which has been now been stable for years.

Since 2016 we switched back to a "rolling release", not defining specific versions.

The main change since the 0.4 version is the use of the newly-introduced `map` Erlang datatype, resulting in the `hashtable` module being replaced by the `map_hashtable`. They obey to the same API and the `table` pseudo-type abstracts out the actual choice in that matter (it is transparently parse-transformed into the currently-retained datatype).

Version 0.4

It is mainly a BFO (*Bug Fixes Only*) version, as functional coverage is pretty complete already.

Main changes are:

- debug mode enhanced a lot: many checkings are made at all fronteers between WOOPER and either the user code (messages) or the class code (constructors, methods, destructor, execute requests); user-friendly explicit error messages are displayed instead of raw errors in most cases; `is_record` used to better detect when an expected state is not properly returned
- `wooper_result` not appended any more to method returns in debug mode
- release mode tested and fixed
- `exit` replaced by `throw`, use of newer and better `try/catch` instead of mere `catch`
- destructor chained calls properly fixed this time
- `delete_any_instance_referenced_in/2` added, `wooper_return_state_*` macros simplified, `remote_*` bug fixed

Version 0.3

Released on Wednesday, March 25, 2009.

Main changes are:

- destructors are automatically chained as appropriate, and they can be overridden at will
- incoming EXIT messages are caught by a default WOOPER handler which can be overridden on a per-class basis by the user-specified `onWOOPERExitReceived/3` method

- direct method invocation supported, thanks to the `executeRequest` and `executeOneway` constructs, and `wooper_result` no more appended to the result tuple
- synchronous spawn operations added or improved: `synchronous_new/synchronous_new_link` and al; corresponding template updated
- state management enriched: `popFromAttribute` added
- all new variations on remote nodes improved or added
- major update of the documentation

Version 0.2

Released on Friday, December 21, 2007. Still fully functional!
Main changes are:

- the sender PID is made available to requests in the instance state variable (see `request_sender` member, used automatically by the `getSender` macro)
- runtime errors better identified and notified
- macros for attribute management added, existing ones more robust and faster
- fixed a potential race condition when two callers request nearly at the same time the WOOPER class manager (previous mechanism worked, class manager was a singleton indeed, but second caller was not notified)
- improved build (Emakefile generated), comments, error output
- test template added
- documentation updated

Version 0.1

Released on Sunday, July 22, 2007. Already fully functional!

WOOPER Inner Workings

Each instance runs a main loop (`wooper_main_loop/1`, defined in wooper.hrl) that keeps its internal state and, through a blocking `receive`, serves the methods as specified by incoming messages, quite similarly to a classical server that loops on an updated state, like in:

```
my_server(State) ->
  receive
    {command,{M,P}} ->
      NewState = execute_command(State,M,P),
      my_server(NewState)
  end.
```

In each instance, WOOPER manages the tail-recursive infinite surrounding loop, `State` corresponding to the (private) state of the instance, and `execute_command(State,M,P)` corresponding to the WOOPER logic that triggers the user-defined method `M` with the current state (`State`) and the specified parameters (`P`), and that may return a result.

The per-instance kept state is twofold, in the sense that it contains two associative tables, one to route method calls and one to store the instance attributes, as explained below.

Method Virtual Table

General Principle This associative table allows, for a given class, to determine which module implements actually each supported method.

For example, all instances of `class_Cat` have to know that their `getWhiskerColor/1` method is defined directly in that class, as opposed to their `setAge/2` method whose actual implementation is to be found, say, in `class_Mammal`, should this class have overridden it from `class_Creature`.

As performing a method look-up through the entire inheritance graph at each call would waste resources, the look-up is precomputed for each class.

Indeed a per-class table is built at runtime, on the first creation of an instance of this class, and stored by the unique (singleton) WOOPER class manager that shares it to all the class instances.

This manager is itself spawned the first time it is needed, and stays ready for all instances of various classes being created (it uses a table to associate to each class its specific virtual table).

This per-class method table has for keys the known method names (atoms) for this class, associated to the values being the most specialised module, in the inheritance graph, that defines that method.

Hence each instance has a reference to a shared table that allows for a direct method look-up.

As the table is built only once and is theoritically shared by all instances[9], it adds very little overhead, space-wise and time-wise. Thanks to the table, method look-up is expected to be quite efficient too (constant-time).

[9] Provided that Erlang does not copy these shared immutable structures, which unfortunately does not seem to be currently the case with the vanilla virtual machine. In a later version of WOOPER, the per-class table will be precompiled and shared as a module, thus fully removing that per-instance overhead.

Attribute Table

This is another associative table, this time necessarily per-instance.

Keys are attribute names of that instance, values are the corresponding attribute values.

It allows a simple, seamless yet efficient access to all data members, including inherited ones.

Issues & Planned Enhancements

- test the impact of using HiPE by default
- integrate automatic persistent storage of instance states, for example in Mnesia databases
- add a mode to support *passive* instances, i.e. pure data-structures not hosted by a specific process; should be as easy as introducing a `new_passive` operator, returning actually the initial state as it is
- integrate specific constructs for code reflection
- check that a class specified in `execute*With` is indeed a (direct or not) mother class of this one, at least in debug mode
- check that referenced attributes are legit (existing, not reserved, etc.)
- support qualifier-based declarations of methods and attributes (`public`, `protected`, `private`, `final`, `const`, etc.)
- even when pasting a template, having to declare all the new-related operators (ex: `new_link/N`) is a bit tedious; an appropriate parse transform could do the trick and automate this declaration
- ensure that all instances of a given class *reference* the same table dedicated to the method look-ups, and do not have each their own private *copy* of it (mere referencing is expected to result from single-assignment); storing a per-class direct method mapping could also be done with prebuilt modules: `class_Cat` would rely on an automatically generated `class_Cat_mt` (for "method table") module, which would just be used in order to convert a method name in the name of the module that should be called in the context of that class, inheritance-wise; or, preferably, this information could be added directly to `class_Cat`

Licence

WOOPER is licensed by its author (Olivier Boudeville) under a disjunctive tri-license giving you the choice of one of the three following sets of free software/open source licensing terms:

- Mozilla Public License (MPL), version 1.1 or later (very close to the former Erlang Public License, except aspects regarding Ericsson and/or the Swedish law)
- GNU General Public License (GPL), version 3.0 or later
- GNU Lesser General Public License (LGPL), version 3.0 or later

This allows the use of the WOOPER code in as wide a variety of software projects as possible, while still maintaining copyleft on this code.

Being triple-licensed means that someone (the licensee) who modifies and/or distributes it can choose which of the available sets of licence terms he is operating under.

We hope that enhancements will be back-contributed (ex: thanks to merge requests), so that everyone will be able to benefit from them.

Sources, Inspirations & Alternate Solutions

- **Concurrent Programming in Erlang**, Joe Armstrong, Robert Virding, Claes Wikström et Mike Williams. Chapter 18, page 299: Object-oriented Programming. This book describes a simple way of implementing multiple inheritance, without virtual table, at the expense of a (probably slow) systematic method look-up (at each method call). No specific state management is supported

- Chris Rathman's approach to life cycle management and polymorphism. Inheritance not supported

- As Burkhard Neppert suggested, an alternative way of implementing OOP here could be to use Erlang behaviours. This is the way OTP handles generic functionalities that can be specialised (e.g. `gen_server`). One approach could be to map each object-oriented base class to an Erlang **behaviour**. See some guidelines about defining your own behaviours and making them cascade

- As mentioned by Niclas Eklund, despite relying on quite different operating modes, WOOPER and Orber, an Erlang implementation of a **CORBA ORB** (*Object Request Broker*) offer similar OOP features, as CORBA IDL implies an object-oriented approach (see their OMG IDL to Erlang Mapping)

WOOPER and Orber are rather different beasts, though: WOOPER is very lightweight (less than 2300 lines, including blank lines and numerous comments), does not involve a specific (IDL) compiler generating several stub/skeleton Erlang files, nor depends on OTP or on Mnesia, whereas Orber offers a full CORBA implementation, including IDL language mapping, CosNaming, IIOP, Interface Repository, etc.

Since Orber respects the OMG standard, integrating a new language (C/C++, Java, Smalltalk, Ada, Lisp, Python etc.) should be rather easy. On the other hand, if a full-blown CORBA-compliant middleware is not needed, if simplicity and ease of understanding is a key point, then WOOPER could be preferred. If unsure, give a try to both!

See also another IDL-based approach (otherwise not connected to CORBA), the Generic Server Back-end (wrapper around `gen_server`).

The WOOPER name is also a tribute to the vastly underrated Wargames movie (remember the WOPR, the NORAD central computer?) that the author enjoyed a lot. It is as well a second-order tribute to the *Double Whopper King Size*, which is a great hamburger indeed[10].

[10] Provided of course one is fine with eating other animals (this is another topic).

Support

Bugs, questions, remarks, patches, requests for enhancements, etc. are to be sent to the project interface, or directly at the mail address mentioned at the beginning of this longer document.

Please React!

If you have information more detailed or more recent than those presented in this document, if you noticed errors, neglects or points insufficiently discussed, drop us a line! (for that, follow the Support guidelines).

Ending Word

Have fun with WOOPER!