Wrapper for Object-Oriented Programming in Erlang

Organisation: Copyright (C) 2008-2024 Olivier Boudeville

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

Creation date: Sunday, August 17, 2008

Lastly updated: Tuesday, February 13, 2024

Version: 2.0.28

Status: Stable

Dedication: Users and maintainers of the `WOOPER` layer.

Abstract: The role of the `WOOPER` layer is to provide free software object-oriented facilities to the Erlang language.

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

This 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 (yet it uses Myriad and is used by Traces). WOOPER can be readily built and run on most Unices (including of course GNU/Linux) and on Windows.

The project repository is located here.

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

Seasoned WOOPER users may directly bookmark this always useful cheat sheet.

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 these subjects a bit more in-depth.

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, passive or active instances, 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)
active instance	process
active instance reference	process identifier (PID)
passive instance	opaque term

... continued on next page

WOOPER base concept	Corresponding mapping to Erlang
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)
member method definition	module function that respects some conventions (request/oneway/static method)
member 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
instance attributes	key/value pairs stored in the instance state
class (static) method	module function that respects some conventions

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).

% Optional:
-define(class_description,"Models a domestic cat").

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

% Declaration of class-specific attributes:
% (optional, yet recommended for clarity)
-define(class_attributes,[
        {meow_style,style(),const,"the kind of meow to expect"},
        {whisker_color,"the color of this cat's whiskers"}]).

% Allows to define WOOPER base variables and methods for that class:
```

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

% No need to export constructors, destructor or methods.
% Type specifications remain optional (yet are recommended).

% Constructs a new Cat.
construct(State,Age,Gender,FurColor,WhiskerColor) ->
    % 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:
    setAttribute(ViviparousMammalState,whisker_color,WhiskerColor).

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

% Member methods.

% A cat-specific request, supposing the developer missed the fact
% that it is a const one (no problem):
getWhiskerColor(State)->
    wooper:return_state_result(State,?getAttr(whisker_color)).

% A (non-const) oneway, with a spec:
-spec setWhiskerColor(wooper:state(),foo:color()) -> oneway_return().
setWhiskerColor(State,NewColor)->
    NewState = setAttribute( State, whisker_color, NewColor ),
    wooper:return_state(NewState).

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

canEat(State,croquette) ->
    wooper:const_return_result(true);

canEat(State,meat) ->
    wooper:const_return_result(true);

canEat(State,_OtherFood) ->
    wooper:const_return_result(false).

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

Straightforward, isn't it? We will discuss it in-depth, though.
To test this class (provided that GNU make and the latest stable version of

`Erlang` are available in one's environment, see Prerequisites for more information), one can easily install `Ceylan-WOOPER`, which depends on Ceylan-Myriad, hence is to be installed first:

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


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

(for OTP compliance, using short names, such as `myriad` or `wooper`, for clones rather than long ones, such as `Ceylan-Myriad` or `Ceylan-WOOPER`, is recommended)

Running the cat-related example just then 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)

--> Testing module class_Cat_test.
[..]
Deleting cat <0.80.0>! (overridden destructor)
Deleting mammal <0.80.0>! (overridden destructor)
Actual class from destructor: class_Cat.
Deleting mammal <0.82.0>! (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 rely on similar kinds of states and behaviours, while still being able to be further specialised on a per-type basis.

The example mentioned through the current guide is an illustration[1] of the interacting lives of numerous animals of various species. Obviously, they have to share behaviours (e.g. 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 user-side 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 instead (a free software discrete-time simulation engine).

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 all objects created from that class shall support.

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`.

So a class modeling, for example, 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.

Note that, unless specifically ambiguous, for the sake of brevity classes are often referred to by their name without their `class_` prefix.

For example `PinkFlamingo` can be understood as a shorthand for the actual classname, `class_PinkFlamingo`.

Class Description

A class should not be implemented without adding at least a short description of it. Rather than describing it through a mere in-code comment (hence only addressed to the class maintainer), a better approach is to used the `class_description` define, like in:

```
-define(class_description,"Class in charge of implementing the "
                          "Foobar service.").
```

Doing so allows that information to be available to humans and tools alike[2].

Inheritance & Superclasses

A WOOPER class can inherit from other classes, in which case the state and behaviour defined in the mother classes will be 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 their mother classes, and so on. This is to lead to a class hierarchy that forms a direct, acyclic graph.

The direct mother classes (and only them) are to be declared in WOOPER thanks to the `superclasses` define. For example, a class with no mother class should specify, once having declared its module:

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

In this particular case, with no mother class to be declared, this `superclasses` define could be omitted as a whole (yet this would be probably less obvious to the reader).

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:

_____

[2]More generally, over time we tend to see any remaining comment as a potential candidate to "metadata promotion". This way, the corresponding information can be used in multiple contexts (e.g. when generating documentation from code).

11

```
-define(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[4] (automatically defined by WOOPER):

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

Instances

Instance Mapping

With WOOPER, which focuses on multi-agent systems, all **active 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[5]).

Such an instance process simply loops over its state forever, waiting for incoming method calls and processing them one after the other.

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.

---

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

[4]Note that, to anticipate a bit, a static method (i.e. a class method that does not apply to any specific instance of it) of a class X is nothing more than an Erlang function, exported by WOOPER from the corresponding `class_X` module and which would return its result R as: `wooper:return_static(R)`. So the corresponding type specification would be `-spec get_superclasses() -> static_return([wooper:classname()]).` here.

[5]For some uses, such a concurrent feature (with *active* instances) may not be needed, in which case one may prefer dealing with purely *passive* instances (implemented as mere Erlang *terms* instead of 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 `new_passive/N`, returning to the caller process an opaque term corresponding to the initial state of a new passive instance, a term that can be then stored and interacted upon at will. See the passive instance section for more details. Most of this document concentrates on active instances, so, unless specified otherwise, just mentioning *instance* by itself refers to an active one.

Methods

They can be either:

- **member methods**: they applies to a specific *instance* (of a given class), like in: `MyCatPid !  declareBirthday`

- or **static methods**: they are general to a *class*, not targeting specifically an instance of it, like in: `class_Cat:get_default_mew_duration()`

Unless specified otherwise, just mentioning *method* by itself refers to a *member method*. Static methods are discussed in their specific subsection (see Static Methods).

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 that are 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 have 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 state of this cat: 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 `superclasses` define already implies it (implying one time for all that cats *are* creatures and thus, unless specified otherwise, are and behave as such). 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

All cat-specific methods (member or static ones) are to be defined in the context of `class_Cat` (defined, as mentioned, in `class_Cat.erl`). Defining a method automatically declares it, so no method should be explicitly exported (knowing WOOPER is to take care of it).

The arity of member methods should be equal to the number of parameters they should be called with, plus one that is automatically managed by WOOPER and that corresponds to the (strictly private, never exported or sent to anyone) 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.

> **Note**
>
> To respect the principle of least astonishment, WOOPER demands that this first parameter is named exactly `State` (doing otherwise will result in a compile-time WOOPER error being issued).

Method Invocation

Let's suppose that the `MyCat` variable designates an (active) instance of `class_Cat`. Then this `MyCat` reference is actually just the PID of the Erlang process hosting this instance; so it may be named `MyCatPid` instead for additional clarity.

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 properly formatted Erlang message, sent to the targeted instance via its PID.

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

Oneways, as for them, are to be triggered with no caller information[6], since no answer is to be sent back.

Therefore the `self()` parameter in the call tuples for requests 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,
```

---

[6]Should the caller PID be nevertheless of use for a given oneway (this may happen), this information shall be listed among its expected parameters.

```
% 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), i.e. the one specified when defining them (e.g. `canEat`).

We recommend that their name is spelled in CamelCase and remains short and descriptive, and start with a verb, like in: `getColor`, `computeSum`, `registerDefaultSettings`, etc.

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

The list of the other reserved names (that shall thus not be defined by a class developer) includes:

- `get_classname` and `get_superclasses`

- `executeRequest` and `executeRequestAs`, `executeConstRequest` and `executeConstRequestAs`

- `executeOneway` and `executeOnewayAs`, `executeConstOneway` and `executeConstOnewayAs`

- `new` and other related construction operators (`new_link`, `synchronous_new`, etc.; see below)

- `delete_any_instance_referenced_in`, `delete_synchronously_any_instance_referenced_in`, `delete_synchronously_instances`

They are reserved for all arities.

The method name is always the first information given when calling it (typically in the method call tuple).

Method Parameters

All methods are free to change the state of their instance and possibly to trigger any side-effect (e.g. sending a message, writing a file, kidnapping Santa Claus, etc.).

As detailed below, there are two kinds of member 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, like in: `{getWhiskersColor,[],self()}`.

If only a single, non-list, parameter is needed, the list can be omitted, and the parameter can be directly specified. So `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 ! {setFavouriteFoods,Foods}` would result in a call to `setFavouriteFoods/4`, i.e. a call to `setFavouriteFoods(State,meat,soup,croquette)`, whereas the intent of the programmer is probably to call a `setFavouriteFoods/2` method like `setFavouriteFoods(State,Foods) when is_list(Foods) -> [..]` instead.

The proper call would then be `MyCat ! {setFavouriteFoods,[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.

Note also that, of course, strings *are* lists. So `Joe ! {setName,"Armstrong"}.` is likely not the call you are looking for. Most probably you should prefer: `Joe ! {setName,["Armstrong"]}.`.

Two Kinds of Member Methods

Request Methods   A **request** is a member 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 member 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[7].

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 run-time error (it has a bug).	Called instance
`wooper_method_faulty_return`	Method does not respect the WOOPER return convention.	Called instance

> **Note**
>
> As mentioned above, failure detection may better be done through the use of (Erlang) links, either explicitly set (with `erlang:link/1`) or, preferably (e.g. to avoid race conditions), with a linked variation of the `new` operator (e.g. `new_link/N`), as discussed later in this document. So a reader in a hurry may want to skip these considerations and directly jump to the Method Definition section.

`wooper_method_not_found`   The corresponding error message is:

```
{wooper_method_not_found,InstancePid,Classname,MethodName,
 MethodArity,ListOfActualParameters}
```

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.

---

[7] Note, though, that in general terms there is little interest in pattern-matching these messages (defensive programming is not always the best option; linking created active instances to their creator is usually a better approach).

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 that there is a runtime failure, it is generally deemed an 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 (more on that later).

It means that the method is not implemented correctly (it has a bug), or, possibly, 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 {Pid,Error} tuple 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,
                UnexpectedTerm}->
        [..];
    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[8]:

```
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) -> [..]`.

---

[8] In which case, should a failure happen, the method call will become blocking; linking or monitoring instances can then be done.

For the sake of clarity, this variable should always be named `State` exactly (implying it shall not be named for example `MyState`, or muted as `_State`[9]). This convention is now enforced at compile-time.

A method must always return at least the newer instance state, so that WOOPER can rely on it from now onward.

Note that when a method "returns" the state of the (active) 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 as a message/visible from outside of its process (unless of course a developer writes specific methods for that).

Encapsulation is ensured, as the instance is the only process able to access its own state. On method termination, the instance then just loops again, on 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 state 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, has already been determined at startup, 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, its specified result is sent back to the caller

- then the instance loops again, on the 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, short of knowing the calling PID).

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 a corresponding method terminator (typically either, respectively, `wooper:return_state_result/2` or `wooper:return_state/1`) to ensure that a new state *and* a result are returned, or just a new state.

Note that all clauses of a given method must end directly with such a method terminator; this is so not only to be clearer for the reader, but also for WOOPER itself, so that it can determine the type of method at hand.

Finally, a recommended good practice is to add a type specification (see Dialyzer) to each method definition, which allows to indicate even more clearly whether it is a request or a oneway, whether it is a `const` method, etc.; more

---

[9]The only legit place for `_State` is when a method clause does not use at all the state variable, which, in practice, happens only when a clause throws an exception.

generally WOOPER is fully compatible with the type-checking tools used by Myriad.

Method-level comments are surely welcome additions as well, and may be considered as mandatory.

For Requests

Requests in general   Requests will use `wooper:return_state_result(NewState,Result)` to terminate their clauses: the new state will be kept by the instance, whereas the result will be sent to the caller. Hence `wooper:return_state_result/2` means that the method returns a state **and** a result.

For example:

```
declareSettings(State,Settings) ->
    NewState = register_settings(Settings,State),
    wooper:return_state_result(NewState,settings_declared).
```

Two remarks there:

- `register_settings/2` is an helper function here; the `State` parameter is intentionally put in last position to help the reader distinguishing it from methods (see Helper vs Static for more information on this topic)

- returning a constant atom (`settings_declared`) has actually an interest: it allows to make that operation synchronous (i.e. the caller is to wait for that result atom; it is only when the caller receives it that it will know for sure that the operation was performed; otherwise a oneway shall be used)

All methods are of course called with the parameters that were specified in their call tuple.

For example, if we declare following request:

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

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

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

Const Requests   Some clauses of a request may return an unchanged state. It is then a `const` clause, and rather than using the `wooper:return_state_result/2` request terminator, it shall use the `wooper:const_return_result/1` one.

A request whose clauses are all `const` is itself a const request.

For example, instead of:

```
getWhiskerColor(State) ->
    wooper:return_state_result(State,?getAttr(whisker_color)).
```

one should prefer writing this const request as (and WOOPER will enforce it):

```
getWhiskerColor(State) ->
    wooper:const_return_result(?getAttr(whisker_color)).
```

Note that `State` can be used as always, and that even there it is not reported as unused (so one should not attempt to mute it, for example as `_State`).

Sender PID   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, and may be useful for some of them.

From a request body, the caller PID can indeed be retrieved by using the `getSender/0` 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/0` 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/0` 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`.

Failsafe Request Calls   Whereas some WOOPER integrated calls include a time-out, in the general case when a request call is made by a client process, this client will wait *indefinitely* for the result from the corresponding requested (active) instance.

It may be desirable instead that this waiting gets interrupted, should a crash of said instance be detected. This can be done by having a **link** between the client process (which may or may not trap EXIT messages) and the requested

instance. Such a link can be created at instance creation (e.g. with `new link`) or later.

Another, often more appropriate, way for the client to be notified is to **monitor** the requested instance: if this last one crashes while being monitored, the monitoring processes (hence possibly the client process) will receive a `DOWN` message and possibly generate an exit call. This way a client will never block waiting for a crashed instance[10].

In the future, the API for WOOPER integrated calls may include `execute_monitored_request/{2,3,4}` functions (see gen_server:do_call/4 as an inspiration).

See also our process monitors section, so that a WOOPER instance can monitor processes (possibly other WOOPER - active - instances).

Request Type Specifications   Using them is not mandatory, yet is very much recommended, and WOOPER provides suitable constructs for that.

As mentioned, a request is to return a new state and a result. The former is always `wooper:state()`, so it may be made implicit. The latter can be any type `T()`. So a request may be considered as returning the WOOPER `request_return(T())` type.

As for const requests, they shall be considered returning the `const_request_return(T())` type.

Making the previous examples more complete:

```
-spec declareSettings(wooper:state(),settings()) ->
                request_return('settings_declared').
declareSettings(State,Settings) ->
    NewState = register_settings(Settings,State),
    wooper:return_state_result(NewState,settings_declared).

-spec getWhiskerColor(wooper:state()) ->
                const_request_return(color()).
getWhiskerColor(State) ->
    wooper:const_return_result(?getAttr(whisker_color)).
```

(of course the developer is responsible for the definition of the `settings()` and `color()` types here)

Note that we prefer surrounding atoms in single quotes when specified as a type.

Of course, should type specifications be used, they must be correct; WOOPER will for example raise a compile-time error should `request_return/1` be used on a function that is not detected as a request.

For Oneways   After relevant adaptations, most of the conventions for requests apply to oneways.

Oneways in general   Oneways will use `wooper:return_state(NewState)` to terminate their clauses: the new state will be kept by the instance, and

---

[10]Similarly to a gen_server:call/{2,3}, which automatically monitors the server process and thus can exit the calling process with an exit term containing `Reason = timeout` if appropriate.

no result will be returned to the caller (which is not even known - hence no `getSender/0` macro applies to oneways either).

For example:

```
setAge(State,NewAge) ->
    wooper:return_state(setAttribute(State,age,NewAge)).
```

This oneway can be called that way:

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

Const Oneways   Even if it is less frequent than for requests, oneways may also be `const`, i.e. may leave the state unchanged, and consequently are only called for side-effects; for example, rather than specifying:

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

WOOPER will ensure that, in this case, `wooper:const_return/0` is preferred to `wooper:return_state/1`:

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

A oneway whose clauses are all `const` is itself a const oneway.

Oneway Type Specifications   The type specification of a oneway should rely, for its return type, either on `oneway_return()` or on `const_oneway_return()`, depending on its constness (no result to account for in either case).

Making the previous examples more complete:

```
-spec setAge(wooper:state,age()) -> oneway_return().
setAge(State,NewAge) ->
    wooper:return_state(setAttribute(State,age,NewAge)).

-spec displayAge(wooper:state) -> const_oneway_return().
displayAge(State) ->
    io:format("My age is ~B~n.",[?getAttr(age)]),
    wooper:const_return().
```

Usefulness Of the Method Terminators   The actual definition of the method terminators (e.g. `wooper:return_state_result/2`, `wooper:return_state/1`) is actually quite straightforward.

For example `wooper:return_state_result(AState,AResult)` will simply translate into `{AState,AResult}`, and `wooper:return_state(AState)` will translate into `AState`.

Their purpose is just to structure the method implementations, helping the method developer not mixing updated states and results, and helping WOOPER in categorizing appropriately all Erlang-level functions.

More precisely, as mentioned, all clauses of a method must directly end with a call to its corresponding WOOPER method terminator.

For example, the following extract is correct:

```
% Returns the name of this instance.
-spec getName(wooper:state()) -> request_return(name()).
getName(State) ->
  Name = nested_in_request(State),
  wooper:const_return_result(Name).

% (helper)
nested_in_request(State) ->
  ?getAttr(name).
```

Whereas the next one is wrong, as `getName/1` would be identified as a unexported plain function (instead of as a const request), and the other way round for `nested_in_request/1`:

```
% Returns the name of this instance.
-spec getName(wooper:state()) -> request_return(name()).
getName(State) ->
  nested_in_request(State).

% (helper)
nested_in_request(State) ->
  wooper:const_return_result(?getAttr(name)).
```

Defining `nested_in_request/1` as shown below would not help either of course:

```
% (helper)
nested_in_request(State) ->
  ?getAttr(name).
```

So, should a method be reported as unused, most probably that no method terminator was used (hence it was not identified as such, and thus not auto-exported, and thus may be reported as unused).

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 the `declareBirthday/1` oneway of `class_Mammal` (which, among other things, is expected to increment the mammal age), one may want to perform a call to its `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 (e.g. called as `class_Creature:setAge(...)`) or that is local to the current module (a direct `setAge(...)` local call designating then `class_Mammal:setAge/2`).

However, in the future, child classes of `class_Mammal` may be introduced (e.g. `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.

Inheritance-based Self-Invocation To perform the self-invocation of a method whose actual implementation is automatically determined based on the inheritance of the class at hand, 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)
```

Const requests can be called[11] as well, like in:

```
      Color = executeConstRequest(CurrentState,getColor)
```

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)
```

Const oneways can also be called[12] as well, like in:

```
      executeConstOneway(CurrentState,displayAge)
```

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.

Self-Invocation of an Explicitly-Designated Method   One can specify **explicitly** the class (of course belonging to the inheritance graph of the class at hand) defining the version of the method that one wants to execute, bypassing 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/2` method instead of the original `class_Creature:setAge/2`.

What if our specific method of `class_Cat` wanted, for any reason, to call the `class_Creature` version of `setAge/2`, now shadowed by an overridden version of it? In this case a `execute*As` function should be used.

---

[11] Note that currently WOOPER will not check that a called request is indeed const, and will silently drop any updated state.

[12] Note that currently WOOPER will not check that a called oneway is indeed const, and will silently drop any updated state.

These functions, which are `executeRequestAs/{3,4}` and `executeOnewayAs/{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 = executeOnewayAs(State,class_Creature,setAge,NewAge)
```

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

Finally, as one could expect, these functions have their const counterparts, namely: `executeConstRequestAs/{3,4}` and `executeConstOnewayAs/{3,4}`, whose usage offers no surprise, like in:

```
Color = executeConstRequestAs(MyState,class_Vehicle,
                              getColorOf,[wheels])
```

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 a specified process or PID, they are just to be called thanks to their module name, exactly as any exported standard function.

In order to further separate them from member methods, we recommend that the names of static methods obey the `snake_case` convention (as opposed to `CamelCase` one): a static method may for example be named `get_default_settings` (rather than `getDefaultSettings`).

Being class-level, their actual definition does not involve any specific instance state, and so only a result is to be returned thanks to their method terminator, which is `wooper:return_static/1`.

The same applies to their result type in terms of type specification, which is to be expressed using `static_return(T())`.

Here are a few examples of rather straightforward static methods, with or without type specifications:

```
get_default_whisker_color() ->
    wooper:return_static(black).

-spec determine_croquette_appeal(cat_name()) ->
          static_return('strong'|'moderate'|'weak').
determine_croquette_appeal(_CatName="Tortilla") ->
    wooper:return_static(strong);
```

```
determine_croquette_appeal(_CatName="Abysse") ->
    wooper:return_static(moderate).
```

An example of use:

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

See also the section about Methods Not Returning Anything of Interest, which may apply to static methods notably.

Finally, having static methods leaves little interest to defining and exporting one's standard, plain (helper) functions; when doing so, one should wonder whether a static method could not be a solution at least as good.

So the main purpose left to helpers is to factor out common, framework-internal code (not targeted at users) across methods (and possibly classes), especially when it involves an instance state (e.g. `display_foo(Color,Index,State) ->`...).

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 (we recommend using `snake_case` for them; all names starting with `wooper_` are reserved), and is associated to a mutable **value**, which can be any Erlang term; therefore `attribute_name() :: atom()` and `attribute_value() :: any()`.

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

Instance Attributes

Declaring them   Class-specific attributes may be **declared**, with some qualifiers.

Attribute declarations are fully optional[13], yet specifying them is nevertheless recommended, at the first place for the developer and for any upcoming maintainer. As a result, by default WOOPER will issue a warning should no attribute declaration be found.

To do so, the `class_attributes` define must be set (prior to including the WOOPER header) to a list of attribute declarations, like in:

```
-define(class_attributes,[
            ATTR_DECL1,
            ATTR_DECL2,
            [...]
            ATTR_DECLN]).
[...]
-include("wooper.hrl").
[...]
```

These declarations are to relate only to the **class-specific** attributes, i.e. the ones specifically introduced by the class at hand, regardless of the ones inherited from the mother classes.

The most general form of an **attribute declaration** includes the following four information:

---

[13]Current versions of WOOPER do not specifically use these information, but future versions may.

```
{Name, Type, QualifierInfo, Description}
```

where:

- `Name` is the name of that attribute, as an atom (e.g. `fur_color`)

- `Type` corresponds to the type specification of that attribute (e.g. `[atom()]`, `foo:color_index()`); note that the Erlang parser will not support the `|` (i.e. union) operator, like in `'foo'|integer()`; we recommend to use the `union` variadic pseudo-function instead (with any arity greater or equal to 2), like in: `union('foo',integer())`

- `QualifierInfo` is detailed just below

- `Description` is a plain string describing the purpose of this attribute; this is a comment aimed only at humans, which preferably does not start with a capital letter and does not end with a dot (e.g. `"describes the color of the fur of this animal (not including whiskers)"` or a shorter, maybe better, `"color of the fur of this animal (not including whiskers)"`)

A **qualifier information** is either a single qualifier, or a list of qualifiers.
A **qualifier** can be:

- a *scope* qualifier: `public`, `protected` or `private`; in future versions, a public attribute will correspond to the union of `settable` and `gettable` and will result in accessor methods being automatically generated; for example, should the `fur_color` attribute be declared public, then:

    - the `getFurColor/1` const request would be added (with its spec):

        ```
        getFurColor(State) ->
            wooper:const_return_result(?getAttr(fur_color)).
        ```

    - the `setFurColor/2` oneway would be added (with its spec):

        ```
        setFurColor(State,FurColor) ->
            wooper:return_state(
                setAttribute(State, fur_color, FurColor)).
        ```

- an *initialisation* qualifier: `{initial,18}` would denote that the initial value of the corresponding attribute is `18` (this value would then be set even before entering any constructor)

- a *mutability* qualifier: `{const,24}` would denote that the corresponding attribute is `const` and that its (fixed) value is `24` (thus `const` implies here `initial`, which should not specified in that case); `const` can also be specified just by itself (with no initial value), so that it can be initialised later, in constructors, and, of course, just once (this is useful for non-immediate, yet const, values)

- the *none* qualifier: `none` implies that no specific qualifier is specified, and as a result the defaults apply; this qualifier can only be used by itself (not in a list), as an alternative to specifying an empty qualifier list

The defaults are:

- `protected`

- mutable (i.e. non-`const`)

- no specific initial value enforced (not even `undefined`)

So an example of attribute declaration could be:

```
{age, integer(), {initial,18},
  "stores the current age of this creature"}
```

> **Note**
> Currently, these information are only of use for the developer (i.e. for documentation purpose). No check is made about whether they are used, whether no other attributes are used, whether the type is meaningful and indeed enforced, the default initial value is not set, etc.; yet at least some of these information might be handled by future WOOPER versions.

Shorter attribute declarations can also used, then with less than the 4 aforementioned pieces of information mentioned:

- only 3 of them: `{Name, Type, Description}` (implying: qualifier is `none`)

- only 2 of them: `{Name, Description}` (implying: type is `any()`, qualifier is `none`)

- only 1 of them: `Name` (implying: type is `any()`, qualifier is `none`, no description)

(and, of course, any number of attributes may not be specified at all)
Finally, a full example of the declaration of class attributes can be:

```
-define(class_attributes,[
    name,
    {age, integer(), "stores the current age of this creature"},
    birth_date,
    {weight, "total weight measured"}]).
```

Storing them   The attributes of a class instance can be seen as a series of key/value pairs stored in an associative table, whose type has been chosen for its look-up/update efficiency and scalability.

This is a dynamic datastructure, allowing attributes to be added, removed or modified at any time (the safer conventions that apply will be discussed later).

This table, among other elements, is itself stored in the overall instance state, i.e. in the variable designated by `State` specified at the beginning of each member method (and constructors, and destructor), on which the process corresponding to active instances is looping, and whose type is `wooper:state()`.

For clarity, we strongly advise to suffix the name of the various state variables used with `State` (e.g. `RegisteredState`, `FinalState`, etc.).

Managing the State of an Instance

A set of WOOPER-provided 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 a method terminator.

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[14] and/or modifying it) should be done with the `setAttribute/3` function:

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

Its type specification is:

```
-spec setAttribute(wooper:state(), attribute_name(), attribute_value()) ->
                                 wooper:state().
```

For example, `AgeState = setAttribute(State, age, 3)` will return a new state, bound to `AgeState`, exact copy of `State` (notably 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 (e.g. `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 (e.g. `State` and `AgeState`) actually **share** all their terms except the one modified - thanks to the immutability of Erlang variables that 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.

Its type specification is:

---

[14] Attribute creation should (by convention) only be done in constructors (not in methods).

```
-spec setAttributes(wooper:state(), [{attribute_name(), attribute_value()}]) ->
                                wooper:state().
```

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 `swapInAttribute/3` function   This function allows to swap, in the specified state, the current value of the specified attribute with the specified value.
Its type specification is:

```
-spec swapInAttribute(wooper:state(), attribute_name(),
                    NewValue :: attribute_value()) ->
            {wooper:state(), PastValue:: attribute_value()}.
```

For example:

```
{NewState,PastColor} = swapInAttribute(AState,fur_color,_NewValue=black)
```

The `removeAttribute/2` function

Note

The `removeAttribute/2` function is now deprecated and should not be used anymore - unless, maybe, if wanting to favour composition over inheritance, or surely if performing hot code update (for example when a version upgrade results in a given attribute not being defined anymore once relying on the newer class definition).

This function may be used in order to fully remove an attribute entry (i.e. the whole key/value pair).
Its type specification is:

```
-spec removeAttribute(wooper:state(), attribute_name()) ->
                            wooper:state().
```

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

Except in the aforementioned very specific cases, this function is not used, as we prefer defining all attributes once for all, at construction time, and never adding or removing them dynamically: the good practice 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 (kids: don't do that).

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

Reading State

The `hasAttribute/2` function   This function tells whether, in the specified state, the specified attribute exists.

Its type specification is:

```
-spec hasAttribute(wooper:state(), attribute_name()) ->
                                  boolean().
```

For example:

```
case hasAttribute(AState, fur_color) of

   true ->
     [...]
```

> **Note**
>
> Using the `hasAttribute/2` function is generally not recommended, as, once an instance is constructed, a given attribute shall always exist or never - unless wanting to favour composition over inheritance.

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, as already mentioned, 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 from an explicitly-designated state is to be done with the `getAttribute/2` function:

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

Its type specification is:

```
-spec getAttribute(wooper:state(), attribute_name()) ->
                                 attribute_value().
```

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/1 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/1` *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.

The getMaybeAttribute/2 function Getting the value of an attribute that may or may not be defined is to be done with the `getMaybeAttribute/2` function.

Its type specification is:

```
-spec getMaybeAttribute(wooper:state(), attribute_name()) ->
          maybe({'value', attribute_value()}).
```

Note that:

- thanks to Myriad, `maybe(T)` :: `T | 'undefined'`

- the `value` atom tag is used to discriminate a non-existing attribute from one whose value is `undefined`

For example:

```
case getMaybeAttribute(AState, AttributeName) of

    undefined ->
      [...];

    {value, AttrValue} ->
      [...]

end.
```

> **Note**
>
> Using the `getMaybeAttribute/2` function is generally not recommended, as, once an instance is constructed, a given attribute shall always exist or never - unless wanting to favour composition over inheritance.

The getMaybeAttr/1 macro   Like the getAttribute/2 function has the
getAttr/1 macro, to the getMaybeAttribute/2 function is associated a conve-
nience macro, getMaybeAttr/1.

This macro should be used *only* if the attribute of interest cannot have the
undefined value, as this macro returns directly the value V obtained (if any),
instead of its {value,V} tagged version[15].

For example, to return the value of a possibly-defined attribute named
my_description, whose value is known to never be equal to undefined (for
example because it is a string in all cases):

```
get_maybe_description(State) ->
   ?getMaybeAttr(my_description).
```

Read-Modify-Write Operations   Some additional helper functions are pro-
vided for the most common operations, to keep the syntax as lightweight as pos-
sible (and possibly increase a bit the performances due to fewer inter-module
calls).

The addToAttribute/3 function   When having a numerical attribute,
addToAttribute/3 adds the specified number (i.e. integer or float) to the at-
tribute.

Its type specification is:

```
-spec addToAttribute(wooper:state(), attribute_name(), number()) ->
                               wooper:state().
```

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 at-
tribute, subtractFromAttribute/3 subtracts the specified number (i.e. integer
or float) from the attribute.

Its type specification is:

```
-spec subtractFromAttribute(wooper:state(), attribute_name(), number()) ->
                               wooper:state().
```

---

[15]Indeed, for convenience, this macro returns values of type maybe(attribute_value()) (not
of the maybe({'value', attribute_value()}) type, thus an attribute set to the undefined
value could not be discriminated from a non-defined attribute).

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.

Its type specification is:

```
-spec toggleAttribute(wooper:state(), attribute_name()) ->
                                 wooper:state().
```

For example:

```
NewState = toggleAttribute(State, my_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 When having an attribute whose value is a list, appends the specified element to this list, at first position.

Its type specification is:

```
-spec appendToAttribute(wooper:state(), attribute_name(), term()) ->
                                 wooper:state().
```

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

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

The `deleteFromAttribute/3` function When having an attribute whose value is a list, deletes the first match of the specified element from the attribute list.

Its type specification is:

```
-spec deleteFromAttribute(wooper:state(), attribute_name(), term()) ->
                          wooper:state().
```

For example: NewState = deleteFromAttribute(State, my_list_attribute,
hello), with the value corresponding to the my_list_attribute attribute
in State variable being [goodbye, hello, cheers, hello, see_you] should
return a state whose my_list_attribute attribute would be equal to [goodbye,
cheers, hello, see_you] (firs hello is removed), 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
badmatch runtime error. If the attribute exists, but is not a list, a function_clause
runtime error will be issued.

The popFromAttribute/2 function   When having an attribute whose value
is a list, 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.

Its type specification is:

```
-spec popFromAttribute(wooper:state(), attribute_name()) ->
                       {wooper:state(), term()}.
```

For example: {NewState, Head} = popFromAttribute(State, my_list_attribute).
If the value of the attribute my_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   When having an attribute whose
value is a table (that is a Myriad table:table() pseudo-type), adds specified
key/value entry to that table attribute.

Its type specification is:

```
-spec addKeyValueToAttribute(wooper:state(), attribute_name(),
                      table:key(), table:value()) -> wooper:state().
```

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 NewState 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/1` request, 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],
[ begin
    PetPid ! {getAge,[],self()},
    receive
      {wooper_result,Age} ->
        io:format("This is a ~B year old creature.",[Age])
    end
  end || PetPid <- MyPetList ].
```

Running this code should output something like:

```
This is a 4 year old cat.
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

Principle   A diamond-shaped inheritance occurs, in a multiple inheritance context, whenever a class inherits more than once (usually indirectly) of a given class.

> **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, if applicable, be avoided anyway.

This may happen when, like in the WOOPER example presented later in this document:

- a very general "root" class (e.g. `class_Creature`) sits at the top of at least a part of the hierarchy[16]

- various child classes derive from it (e.g. `class_Mammal` and a `class_OvoviviparousBeing` may derive from `class_Creature` for obvious reasons)

- a more concrete class (e.g. `class_Platypus`) has to derive from at least two of these child classes (since a platypus happens to be both a mammal and an ovoviviparous being)

Then the concrete class will end up deriving more than once from the same root class (here: `class_Platypus` will derive indirectly from `class_Creature` twice).

Diamond-shaped inheritance may lead to issues in terms of state management (e.g. attributes having possibly in turn to be defined more than once), behaviour (multiple implementations having to be selected for some methods) and initialisation (extraneous constructor and destructor calls being made).

Modelling & Implementation Choices   The root of such issue is to determine what is the meaning of the multiple `is-a` relationships involved in this diamond-shaped inheritance.

If one design solution is to favour composition over inheritance to sort out these cases, by default we tend with WOOPER to consider that a single is-a relationship should apply in that case, i.e. that for example a platypus "is a" creature in a single way (having to inherit the attributes from creature only once, and its member methods as they are defined at the creature level); it is a case of virtual inheritance.

More precisely, with WOOPER, as the method table is constructed in the order specified in the declaration of the superclasses, like in:

```
-define(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 trying to guess which version of the method in the inheritance graph would be called implicitly.

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.

---

[16]For example, it might be the case with class_TraceEmitter if relying on Ceylan-Traces, or class_Actor if using Sim-Diasca, as such base classes are rather ubiquitous in their respective contexts.

However, in such a diamond-shaped inheritance scheme, the constructor of the class that sits at the top of a given diamond will be called more than once, and 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.

This can be the case naturally, or can be ensured simply whenever a given root class introduces at least one attribute (which is the general case): by testing whether the initial state given to its constructor already includes this attribute, the constructor can determine whether it has already been applied, in which case it may return the state that it received as it is; otherwise (it has not been applied already), it shall then be fully executed, by setting its attributes[17], triggering any relevant processing and side effects, and so on.

When introducing such trait-specific attributes, care must be taken to name them so that they should never clash with other attribute names.

For example, should the `class_Creature` consider that each creature has a date of birth, its constructor may be like:

```
construct(State, DateOfBirth) ->
  case hasAttribute(State, birth_date) of

      % Already initialised, nothing more to do, shortcut:
      true ->
          State;

      % Full, actual initialisation, to happen exactly once per instance:
      false ->
          % Trigger side-effects, change state, etc.:
          InitState = init(State),
          % At least this attribute is defined:
          setAttribute(InitState, birth_date, DateOfBirth)

  end.
```

The next section will show that these idempotent constructors lead naturally to the concepts of composition and interfaces.

Interfaces: to be able to favour Composition over Inheritance

A few specific features for traits, interfaces and composition had been introduced; yet, short of a specific extra support providing enough interest, they have been disabled since WOOPER offers already, natively, everything needed to perform composition over inheritance. Relying on such composition can be as simple as replacing the inheritance from some mother classes by the definition of passive instances in attributes with corresponding accessor methods.

WOOPER also provides a few general-purpose interfaces, such as `Identifiable`, `Describable` and `StaticDescribable`, `Serialisable` (refer to the serialisation section), `Upgradable` (refer to the hot code update section). They can serve as a source of inspiration for any other interface to be introduced.

---

[17] A single set of them exists thus, as opposed to as many sets as the number of times the class of interest inherits from the root class (no state duplication can exist in WOOPER in so far as, for a given instance, attribute names are unique).

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, and are automatically generated), 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 the `new` / `new_link` operators is to *create* a working (active) instance on the user's behalf, the role of `construct` is to *initialise* an instance of that class (regardless of how it was created, i.e. of which `new` variation was triggered), 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 corresponding constructor (`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 (one may note that, because of its first parameter, which accounts for the WOOPER-provided initial `State` parameter, the arity of `construct` is equal to the one of `new` / `new_link` plus one).

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`[18].

The `new` and `new_link` operators are automatically defined by WOOPER (thanks to a relevant parse transform), but they rely on their corresponding, class-specific, user-defined `construct` operator (only WOOPER is expected to make use of it). More precisely, for each of the `construct/N+1` operator defined by the class developer, WOOPER creates a full set of corresponding `new` variations, including `new/N` and `new_link/N`.

At least one `construct` operator must be defined by the class developer (otherwise WOOPER will raise a compile-time error), knowing that any number of them can then be defined, each with its own arity (e.g. `construct/1`, `construct/2`, `construct/3`, etc.), and each with possibly multiple clauses that will be, as usual, selected at runtime based on pattern-matching.

`construct` operators may not be exported explicitly by the class developer, as WOOPER will automatically take care of that if necessary.

For example:

```
% If having defined class_Dog:construct/{1,3}:
MyFirstDog  = class_Dog:new(),
MySecondDog = class_Dog:new(_Weight=4.4,_Colors=[sand,white]).
```

---

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

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 (hence having a `construct/N+1` defined), the following new operator variations, detailed in the next section, are built-in:

- if an **active** 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 an **active** 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+1`
  - blocking creation with time-out: `remote_synchronous_timed_new/N+1` and `remote_synchronous_timed_new_link/N+1`

- if a **passive** instance is to be created by the current **process**: `new_passive/N`

> **Note**
>
> All `remote_*` variations require one additional parameter (that shall 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 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 succeed, and the execution of the calling process continues while (hopefully, i.e. with no guarantee - the corresponding process may immediately crash) 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), unless triggering a later request on it.

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

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

For example, once (if) `MyMammal = class_Mammal:synchronous_new(...)` returns, 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. In some cases a time-out may be useful, so that the caller may be unblocked and may react appropriately.

This is why the `synchronous_timed_new*` operators have been introduced: if the caller-side time-out[19] expires while waiting for the created instance to answer, then they will throw an appropriate exception:

- either `{synchronous_time_out,Classname}` if it was a node-local creation (where `Classname` is the name of the class corresponding to the instance to create; e.g. `class_Cat`)

- or `{remote_synchronous_time_out,Node,Classname}`, where `Node` is the name of the node (as an atom) on which the instance was to be created

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 (active) 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 a corresponding constructor has been defined indeed.

---

[19]Depending on whether or not the class to instantiate was compiled in debug mode, the time-out is to last by default for, respectively, 5 seconds (shorter, to ease debugging) or for 30 minutes (longer, to favour robustness).

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

```
% Local asynchronous creation:
MyFirstCat = class_Cat:new(_Age=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
% (unless it traps EXIT signals):
MyThirdCat = class_Cat:remote_synchronous_timed_new_link(
                                OtherNode,3,male,grey,black),
[...]
```

Definition of the `construct` Operator Each class must define at least one `construct` 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.

The type specification of a constructor relying on `N` construction parameters (hence `construct/N+1`) is:

```
-spec construct(wooper:state(),P1,P2,..,PN) -> wooper:state().
```

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 (should there be multiple direct mother classes, usually their constructors are to be called in the same order as their declaration order in the `superclasses` define).

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 four pieces of information:

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

-define(class_attributes,[whisker_color]).

% Constructs a new Cat.
construct(State,Age,Gender,FurColor,WhiskerColor) ->
  % First the (chained) direct mother classes:
```

```
        MammalState = class_Mammal:construct(State,Age,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/4` (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 this case (where no mother class has been declared):

```
% 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}]).
```

The WOOPER defaults would imply that, in the first case, at compilation time the `AgeState` variable would be reported as unused, and this warning would be considered as a fatal error.

Note

There is no strict relationship between *construction parameters* and *instance attributes*, neither in terms of cardinality, type nor value.

For example, 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 any of its constructors: 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 (e.g. 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 in turn to the destructors of the instance direct mother classes.

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).

Note also that, as always, there is a single destructor associated to a given class.

As constructors, destructors should not be exported, as WOOPER is to automatically take care of that.

Asynchronous Destruction: using `destruct/1`   The type specification of a destructor (`destruct/1`) is:

```
-spec destruct(wooper:state()) -> wooper:state().
```

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 (e.g. `MyInstance !  delete` is issued), WOOPER will take care of:

- calling any user-defined destructor 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:destruct/1`, then `class_B:destruct/1`, then `class_A:destruct/1`, 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`).

For a more concise way of doing the same, see also:

- `wooper:delete_synchronously_instance/1` (for a single instance)

- `wooper:delete_synchronously_instances/1` (for multiple ones)

Passive Instances

A passive instance is an instance of a WOOPER class that is not powered by a dedicated (Erlang) process: it is just a mere (opaque) term, a pure data-structure that holds the state[20] of that instance, and that is returned to the process having created that instance (which can then do whatever it wants with it - most probably store it as a "super attribute").

As a consequence, such a passive instance will not be able to perform any spontaneous behaviour on its own, or to have its member methods triggered by other processes. However most operations that can be done on "standard" (active) WOOPER instances can also be done on passive ones: like their active counterparts, they are constructed thanks to, well, one of the constructors defined by their class, they are destructed thanks to, well, their destructor, and in-between they will retain their inner state and be able to execute any request or oneway triggered by the process holding that term (and of course any underlying multiple inheritance will be respected).

Triggering a method onto a passive instance will result in a relevant function to be evaluated, not involving any message.

To create a passive instance, the `new_passive` operator shall be used, like in:

```
MyPassiveCat = class_Cat:new_passive(_Age=2, female, _Fur=brown, _Whiskers=white)
```

Then methods can be triggered on it, like in:

```
{WhiskerCat,white} = wooper:execute_request(MyPassiveCat, getWhiskerColor),
OlderCat = wooper:execute_oneway(WhiskerCat, declareBirthday),
RedCat = wooper:execute_oneway(OlderCat, setFurColor, red),
[...]
```

Note that, in addition to `execute_request/{2,3}`, `execute_const_request/{2,3}` are available; the latter return only the request result, as the state of the passive instance shall be constant (this is checked in debug mode).

Symmetrically, in addition to `execute_oneway/{2,3}`, `execute_const_oneway/{2,3}` are available even though their use is less common.

Until, finally:

```
wooper:delete_passive(RedCat).
```

Types may be defined in order to clarify the fact that a term corresponds to a (WOOPER) passive instance.

For example, for any kind of instance owning a passive calendar instance (created with, say, `class_Calendar:new_passive/2`), one could define and use, as attribute type[21]:

```
-type calendar_passive() :: wooper:passive_instance().
```

See the `passive_instance_test` module for more details about passive instances.

---

[20]This term is mostly the same state term as the one on which the process dedicated to an active instance is looping. So one could even imagine a WOOPER instance going back and forth between an active and a passive mode of operation, if it was useful.

[21]A type preferably defined that way and exported in `class_Calendar`.

Serialisation

WOOPER offers a rather complete support for **serialisation and deserialisation**, so that instances can be written to and read from a series of bytes (this can be a WSF file, standing for `WOOPER Serialisation File`, a network stream, etc.).

For that the `class_Serialisable` interface provides all default yet overridable primitives, including:

- "serialisation hooks" (e.g. the `onPreSerialisation/2` request), meant to accommodate any class-specific transformation that shall happen prior to each operation

- entry transformers, that are generally specific to a project but transverse to most if not all its classes, so that values of attributes can be recursively scanned for transient terms (like a PID, a reference), which are generally replaced with restoration markers at serialisation time, and recreated appropriately at deserialisation time

The usual variations (e.g. deserialisations may be synchronous or not, linked or not) of operations are supported.

Care as been taken so that these operations are done:

- generically: only the class-specific transformations are left to implement, in well-defined hooks, with relevant helpers already available (e.g. with examples of entry transformers)

- safely: optional checking that no transient term is still present in attribute values that are ready for serialisation, and that no restoration marker remains once a deserialised instance is ready

- efficiently: operations are intensely concurrent (each instance takes care of itself as much as possible), and by default storage is done in a compact, binary form

In practice, the `Serialisable` interface is defined in `class_Serialisable.{hrl,erl}`, and the extended features are implemented in the `wooper_serialisation` module. The `serialisable_test` test module showcases various examples, which involve the (serialisable) `class_Reptile` module.

Hot Code Update

The objective here is to be able to **update classes at runtime** whereas the program is still running, and to have their instances adapt in terms of state and behaviour to the corresponding new versions. Hot code reloading is typically useful in the case of always-available applications (never-stopping services), where bug fixes and additional features have to be deployed in spite of the continuous operation.

WOOPER offers such features, based on the `Upgradable` trait, implemented by the class_Upgradable interface.

An update protocol is proposed, with all support defined so that, for a given class that shall be updated (i.e. whose version must be either upgraded or downgraded):

1. all its instances are synchronously frozen first, to avoid that ongoing method calls experience code discrepancies

2. then the module corresponding to that class is reloaded (possibly after having been recompiled), being soft-purged (and possibly hard-purged if needed)

3. then all its instances perform (concurrently) any state update that they need in order to support the new version

In practice, for a given class (e.g. `class_TestUpgradable`), various versions thereof may be introduced over time (e.g. `1.2.3`, then `1.2.4`). Each version of a given class is free to override the default implementations of the `upgradeVersion/4` and `downgradeVersion/4` methods, as inherited from the `class_Upgradable` interface, in order to account for all changes introduced between two versions (e.g. attributes being added or removed, changing types, having their value transformed).

Refer to class_Upgradable_test for a full test thereof.

Miscellaneous Technical Points

Helper vs Static

In the context of a class, one may wonder what is the difference between an **exported helper function** and a **static method**?

In terms of actual runtime evaluation, none.

In terms of source code, there is little difference: the former just has to use the `wooper:return_static/1` to terminate its clauses and, should it use a type spec, this one should specify a return type based on `return_static/1`.

In terms of semantics, there are more differences: a static method is meant to stand by itself, whether or not an instance of that class exists, and to provide higher-level services.

On the contrary, an exported helper function is just a convenience for code-reuse, a means of sharing code between classes. Typically it centralises code of use by multiple methods, which often results in such an helper to have as a parameter a `State` variable (preferably listed last among its parameters, for clarity/uniformity) - whereas this cannot happen with static methods.

Calling a Method Clause from Another

Sometimes, defining a clause based on the result of another one (of the same function) proves convenient. With methods, there are different ways to do so, some better than others.

For example, let's suppose we defined this (non-const) request:

```
-spec getPriceFor(wooper:state(),string()) ->
        request_return(price()).
getPriceFor(State,ArticleStringDescription) ->
   {Price,NewState} = [...some code...]
   wooper:return_state_result(NewState,Price).
```

Now, we would like to support article descriptions being also possibly binary strings, without having to bother introducing a new price-establishing intermediary function (such as the `foo/2` one, below) that would be common to the two corresponding clauses, like in:

```
getPriceFor(State,ArticleStringDescription) ->
        when is_list(ArticleStringDescription) ->
   {Price,NewState} = foo(ArticleStringDescription,State),
   wooper:return_state_result(NewState,Price);

getPriceFor(State,ArticleBinDescription)
        when is_binary(ArticleBinDescription) ->
   {Price,NewState} = foo(binary_to_list(ArticleBinDescription),
                          State),
   wooper:return_state_result(NewState,Price).

% Too cumbersome to define an extra function such as:
foo(ArticleStringDescription,State)->
   [...some code...]
```

So our first impulse would be to write instead something like:

```
% (same clause)
getPriceFor(State,ArticleStringDescription) ->
        when is_list(ArticleStringDescription) ->
    {Price,NewState} = [...some code...]
    wooper:return_state_result(NewState,Price);

% Added clause:
getPriceFor(State,ArticleBinDescription) ->
        when is_binary(ArticleBinDescription) ->
    getPriceFor(State,
        binary_to_list(ArticleBinDescription)).
```

However this would not be correct, as the second clause lacks a terminator, and WOOPER does not accept that (it might be the sign that the developer believes he is writing an helper function).

So we would like to use `wooper:return_state_result/2` for the second clause as well, however we shall then have a state and a price to feed this terminator. How can we obtain them from the presumably opaque term returned by the `wooper:return_state_result/2` call of the first clause?

The answer is quite simple: all WOOPER method terminators being actually no-op, they are replaced at compilation-time by the actual terms they were given.

So a terminator taking only a state as parameter is replaced literally by this state (e.g. `wooper:return_state(FooState)` is actually `FooState`), while one taking two returns a pair (e.g. `wooper:return_state_result(FooState,Res)` is actually `{FooState,Res}`). No black magic here!

Therefore the implementation we target could be best[22] written as:

```
% (same clause)
getPriceFor(State,ArticleStringDescription) ->
        when is_list(ArticleStringDescription) ->
    {Price,NewState} = [...some code...]
    wooper:return_state_result(NewState,Price);

 % Added clause:
 getPriceFor(State,ArticleBinDescription)
        when is_binary(ArticleBinDescription) ->
    {NewState,Price} = getPriceFor(State,
        binary_to_list(ArticleBinDescription)),
    wooper:return_state_result(NewState,Price).
```

---

[22]`wooper:executeRequest/3` could be used instead of matching the result of `wooper:return_state_result/2` and breaking its opaqueness - but (even it is a matter of taste here) we prefer the more direct and efficient approach that was presented. Note though that in general they do not bear the same semantics: `wooper:executeRequest/3` calls a possibly overridden version of the specified method, whereas a direct call targets specifically the class-local version of that method.

Integrated Call APIs

Single Calls

Calls made to **passive instances** require the use of a dedicated API that is provided by the `wooper` module, namely:

- for requests: `execute_request/{2,3}` and `execute_const_request/{2,3}`

- for oneways: `execute_oneway/{2,3}` and `execute_const_oneway/{2,3}`

As for **active instances**, doing the same directly through Erlang base constructs (message sending and receiving) is so easy that generally no specific API is involved - even though it exists (notably to allow for a few usage variations):

- for requests: `execute_request/{2,3,4}`

- for oneways: `execute_oneway/{2,3}`

(with active instances, constness does not need to be taken at this level)

Multi-Calls

Sometimes, in the context of *active instances*, it may be more convenient to reason not in terms of single calls, but of multiple calls, with various flavours in the number of callers, callees, method calls involved, parallelism and synchronisation.

So the `wooper` module provides facilities in order to execute:

- a given **request sequentially** on **multiple callees**: `send_request_in_turn/3` or, if wanting that a time-out applies, `send_request_in_turn/4`

- the same for a **oneway** sending (off-band, through a direct message sending) a user-specified **acknowledgement term** (e.g. a conventional atom): `send_acknowledged_oneway_in_turn/5`

- a given **request**, whose only result in a user-specified **acknowledgement term**, in **parallel** on **multiple callees**: `send_requests_and_wait_acks/4` or, if wanting that a time-out applies, `send_requests_and_wait_acks/5`

- a given **request** in **parallel** on **multiple callees**, obtaining back an **unordered** list of their results: `obtain_results_for_requests/3`

- a **series of requests** on a **single** callee: `send_request_series/2` (just for the sending) and `obtain_results_for_request_series/2` (to include also the collecting of their ordered results)

Many more flavours can be considered (e.g. for active/passive instances, requests or oneways, with or without time-out, synchronisable or not, etc.) and may be supported in the future, on a per-need basis.

However it may be more convenient to compose any extra multi-call that is needed thanks to the various basic helpers that are provided in the `wooper` module in order to send calls or to collect their results. Doing so allows for example to perform interleaving (triggering concurrent calls as soon as possible, collecting their results as late as possible) and maximise the overall parallelism.

Methods Not Returning Anything of Interest

Not all functions or static methods (or even requests) are pure, and sometimes only the side-effects of their execution are of interest, in which case nothing relevant may have to be returned[23].

For such needs, our Myriad base layer introduced the opaque pseudo-builtin type `void()` (nothing fancy), which is to be used by WOOPER methods as well.

Should one need a value of the `void/0` type, we recommend using the `void` atom.

As a result (pun!), one may define:

```
-spec stop() -> static_return(void()).
stop() ->
  [...] % Do some side-effects
  wooper:return_static(void).
```

Introducing specific method terminators to convey that meaning would have been possible, yet multiplying the syntactical constructs would probably hurt the terseness of the underlying language.

However, as, fairly frequently, static methods have nothing meaningful to return, the `wooper:return_static_void/0` method terminator and its spec-level counterpart `static_void_return()` type have been introduced[24].

Methods Not Returning

Some static methods may not return (see `class_Serialisable:deserialise/4` for an example thereof).

Then their type specification shall use `static_no_return()` and each of their clause shall end with `wooper:no_return_static()`.

Exception-Throwing Methods

A **clause of a method may simply always throw an exception**, and WOOPER will manage it automatically if possible.

More precisely, such a clause is considered licit yet not giving information about the method type. So, should there be other, non-throwing clauses, WOOPER will still be able to identify the nature of the method.

Otherwise (all clauses always throw), the class developer will have to give a hint about the actual nature of that function - thanks to a type specification.

For example, if defining the API of an abstract class (say, an `AbstractShape` class) whereas, for a given method signature (e.g. `getArea/1`), there is no meaningful default implementation, one may for example write:

---

[23]So, why one use a request instead of a oneway then? One reason is synchronisation, so that the caller code can block until the request has completed. Nevertheless for that use case we recommend that such a request returns a conventional, self-describing, preferably unique atom, such as `foobar_server_stopped`. We will focus here mostly on the static methods that would have no meaningful value to return.

[24]This method terminator is replaced at compilation-time by a no-op, thus the static method returns actually the previous expression - unless there is none, in which case the `wooper_void_return` atom is returned (the caller is not expected to match a void result anyway).

```
getArea(_State) ->
    % For an abstract shape, no possible definition;
    % so catching any child class not overriding it,
    % thanks to:
    %
    throw(not_overridden).
```

However, from the point of view of WOOPER, `getArea/1`, short of using any method terminator, can be of any nature (for example a method or an helper function). WOOPER will then request the developer to clarify this ambiguity by adding a type spec, such as:

```
-spec getArea(wooper:state()) -> const_request_return(area()).
```

This will be sufficient for WOOPER.

In other related cases, it may be convenient instead to centralise the management of a range of failures in specific **helper functions that always throw** (such a function never returns), like in:

```
manage_foobar_failure(X,Y) ->
    [...]
    throw({foobar_failed,{X,Y}}).
```

Writing a method relying on such an helper function could be:

```
doSomething(State,A) ->
    case foobar(A,State) of

        {value,Result} ->
            wooper:const_return_result(Result);

        {error,Error} ->
            manage_foobar_failure(A,Error)

    end.
```

The problem is that this call to `manage_foobar_failure/2` does not include a method terminator, and thus this branch would be, from the WOOPER point of view, the one of an helper function, thus clashing with the other branches (here telling it is a const request).

A solution could be to write instead:

```
[...]
        {error,Error} ->
            Dummy = manage_foobar_failure(A,Error),
            wooper:const_return_result(Dummy)
```

but that resulting code would not be that satisfactory (a bit ugly, not so readable).

So, instead, WOOPER introduced the `throwing/1` method terminator, allowing the following, clearer, more elegant, code:

```
    [...]
        {error,Error} ->
            wooper:throwing( manage_foobar_failure(A,Error) )
```

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_pids` containing a list of PID instances

and if the instance at hand that shall be deleted 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_pids).
```

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 Signals / Messages

A class instance may (if trapping EXIT signals) 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:

```
-spec onWOOPERExitReceived(wooper:state(),basic_utils:pid_or_port(),
          basic_utils:exit_reason()) -> const_oneway_return().
onWOOPERExitReceived(State,PidOrPort,ExitReason) ->
   io:format("MyClass EXIT handler ignored signal '~p' "
              "from ~w.~n", [ExitReason,PidOrPort]),
   wooper:const_return().
```

may result in an output like:

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

If no class-level `onWOOPERExitReceived/3` oneway is available, the default WOOPER EXIT handler (namely `wooper:default_exit_handler/3`) will be used (it just performs console-based notification).

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>.
```

`DOWN` Messages for Process Monitors

A class instance may receive DOWN messages from other (monitored) processes.

A given class can process these DOWN notifications:

- either by defining and exporting the `onWOOPERDownNotified/5` oneway

- or by inheriting it

If no class-level `onWOOPERDownNotified/5` oneway is available, the default WOOPER DOWN handler (namely `wooper:default_down_handler/5`) will be used (it just performs console-based notification).

Note that `DOWN` messages shall not be mixed up with the `nodedown` messages of the next section.

Node Monitors

Quite similarly to `EXIT` messages, node monitors and `nodeup` / `nodedown` messages are also managed by WOOPER, see the `onWOOPERNodeConnection/3` and `onWOOPERNodeDisconnection/3` oneways.

Should these oneways be not available for the class at hand, the default WOOPER node handlers (namely `wooper:default_node_up_handler/3` and `default_node_down_handler/3` respectively) will be used (they just perform console-based notifications).

Note that `nodedown` messages shall not be mixed up with the `DOWN` messages of the previous section.

A bit of Introspection

Here are the class-level information that can be fetched from the name of a class (as a result they are static methods):

- `get_superclasses/0` will return the (ordered) list of the declared direct mother classes of this class

- `get_class_specific_attributes/0` will return the (in-order) list of declared class-specific attribute metadata, as a list of `wooper_info:attribute_info/0` records, containing, for a given attribute: its name, type, qualifiers and textual description

For example:

```
class_Platypus:get_superclasses().
[class_Mammal,class_OvoviviparousBeing]
```

When having at hand the state of an instance (so either from its methods and/or for a passive instance), various helpers in the `wooper` module can be used (e.g. `get_classname/1`, `get_all_superclasses/1`, `is_instance_of/2`, `check_instance_of/2`, `get_attribute_pairs/1`, `state_to_string/1`, `get_class_filename/1`). They are mostly intended for testing, as a good OOP design generally does not rely on such primitives.

Developer Guidelines

General Guidelines

All WOOPER classes must include wooper.hrl:

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

> **Note**
>
> This include should come, in the source file of a class, *after* all WOOPER-related defines (such as `superclasses`, `class_attributes`, etc.).

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.

For examples of re-use of WOOPER by upper layers, one may refer to Ceylan-Traces, to US-Web or to the Sim-Diasca simulation engine.

OTP Guidelines

For server-like activities, it may be useful to have instances of WOOPER classes (e.g. `class_Foobar`) behave like OTP workers in supervision trees.

Exactly like by default any Erlang process is not able to readily take part to an OTP supervision tree[25], a WOOPER instance must specifically support such an integration.

A supervisor bridge (i.e. a process in charge of integrating properly such a non-OTP process into an OTP supervision tree) is the tool of choice here[26].

Rather than declaring the corresponding behaviour and its implemented API directly in the target class (e.g. in `class_Foobar`), we recommend for *clarity* (as, even if the bridge was declared in the same module, it would be still an intermediate, autonomous process sitting between the OTP supervisor

---

[25]For that, such process has to comply with a relevant applicative protocol in order to properly interact with its supervisor, either directly by itself or through its implementation of a worker-compliant OTP behaviour like gen_server.

[26]Otherwise, as we understand it, should WOOPER instances be directly declared in a supervision tree as mere workers, they would have to implement typically the `gen_server` behaviour, which implies relying on `handle_{call,cast,info}` functions and on a behaviour-specific main loop - rather than on the counterpart method-based facilities offered by WOOPER.

A consequence is the need for an extra, mostly idle process sitting between such a supervised WOOPER instance and its supervisor. In some cases (e.g. a webserver processing each request based on a WOOPER instance), this overhead would be excessive; then defining a WOOPER-specific supervisor, able to handle directly its WOOPER instances as workers, could be devised.

and the WOOPER instance of interest) and for *separation of concerns* (leaving an OTP integration as an optional feature) to introduce a specific module for that, like done with traces_bridge_sup for class_TraceAggregator.erl or us_common_config_bridge_sup for class_USConfigServer.erl.

However, when the purpose of a given class is solely to integrate within an OTP application through a supervision tree, one may of course have the bridge directly defined in the WOOPER class the interfacing of which it is enabling; see class_USSensorManager.erl for an example thereof.

In all cases, following the aforementioned us_common_config_bridge_sup simple example, the additions mostly boil down to, in a relevant module as discussed above:

- declaring the corresponding OTP behaviour, the exported functions that this implies, and any relevant define, like in:

```
-behaviour(supervisor_bridge).

% User API of the bridge (can be arbitrarily chosen, including of course the arity):
-export([start_link/2]).

% Callbacks of the supervisor_bridge behaviour:
-export([init/1, terminate/2]).

-define(bridge_name, ?MODULE).
```

- implementing how the bridge shall be spawned by the user, like in:

```
start_link(X,Y) ->
  Z=f(X),
  % Supposing here that a local registration of the bridge is wanted:
  supervisor_bridge:start_link({local, ?bridge_name},
      _Module=?MODULE, _InitArgs=[Y,hello,Z]).
```

- implementing what the bridge shall do in terms of initialisation once spawned (notably regarding the WOOPER instance that it interfaces), like in:

```
-spec init( list() ) -> { 'ok', pid(), BridgeState :: term()}
      | 'ignore' | {'error', Error :: term()}.
init(_Args=[_A, Atom, B]) ->
  % No synchronous creation supposed useful here:
  MyFoobarInstancePid = class_Foobar:new_link(Atom,B),

  InitialBridgeState = {MyFoobarInstancePid, Atom},

  {ok, MyFoobarInstancePid, InitialBridgeState}.
```

- do the same regarding termination, like in:

```
terminate(Reason, _BridgeState={FoobarPid, _Atom}) ->
  % No synchronicity especially needed:
  FoobarPid ! delete.
```

Of course:

- any user API and bridge state can be elected, but the supervisor_bridge behaviour requires `init/1` and `terminate/2` to be defined with these arities and to return the terms documented

- the various functions introduced here could be defined as static methods instead, but maybe such exported plain functions will be clearer for developers with an OTP background

- spawned processes must be linked, so that crashes do not remain unnoticed

- we recommend to add type specs and relevant traces in actual code

WOOPER API

One can browse the WOOPER API documentation. WOOPER being, in some way, a dialect deriving from Erlang (hence closer to a language of its own than to a pure library), such an API documentation is most probably less useful than counterpart ones made for libraries like Myriad.

The next level of information is simply to read the corresponding source files, which are intensely commented and generally straightforward.

Class Developer Cheat Sheet

When specifying a **class**: `-module(class_Foobar)`.
Any **description** thereof: `-define(class_description,"Class for...")`.
When specifying its **superclasses**: `-define(superclasses,[A,B])`.
When specifying its **class attributes**: `-define(class_attributes,[ATTR1,ATTR2,...])`
A given `ATTRn` may be one of:

- `{Name,Type,QualifierInfo,Description}`

- `{Name,Type,Description}`

- `{Name,Description}`

- `Name`

`QualifierInfo` can be, for example, `public`, or `[private,const]`.
All member methods have `State` for initial parameter, and are expected to return at least a (possibly const) state.
When defining a **request**:

- its **spec** should rely, for its return type, either on `request_return(T())` or on `const_request_return(T())`

- each of its clause should terminate with either `wooper:return_state_result(S,R)` or `wooper:const_return_result(R)`

When defining a **oneway**:

- its **spec** should rely, for its return type, either on `oneway_return()` or on `const_oneway_return()`

- each of its clause should terminate with either `wooper:return_state(S)` or `wooper:const_return()`

When defining a **static method**:

- its **spec** should rely, for its return type, either on `static_return(T())` or on `static_void_return()`

- each of its clause should terminate with `wooper:return_static(R)` or with `wooper:return_static_void()`

Finding it difficult not to mix the keywords for spec return types (e.g. `request_return/1`) and method terminators (e.g. `wooper:return_state_result/2`)?

A mnemonic could be that a spec return type designates a returned *term* (hence for example a `request_return/1` - more generally such types always end with `_return`) whereas a method terminator designates an *action* (returning a state and a result, hence to `wooper:return_state_result/2` for example).

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

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.) 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).

WOOPER limitations

Actually no significant implementation shortcoming has been identified.

The main limitations that existed beforehand were (from the most recently alleviated to the earlier ones):

- the per-instance memory footprint has been reduced by sharing the "virtual table" of a given class between all its instances, thanks to `persistent_term`

- the efficiency of the per-instance associative table to store the attribute values has been improved by switching from our hashtable implementation to a map-based one

- the macro-based constructor declarations were too cumbersome and limiting (e.g. a single constructor per class was supported); this is fully addressed thanks to the WOOPER parse transform now

We expect the now default JIT compilation to benefit a lot to WOOPER, with no change having to be operated at its level.

As always, there is certainly room for improvement, to grind a bit more runtime performances and/or to further reduce the per-instance memory footprint. Profiling actions welcome!

WOOPER Example

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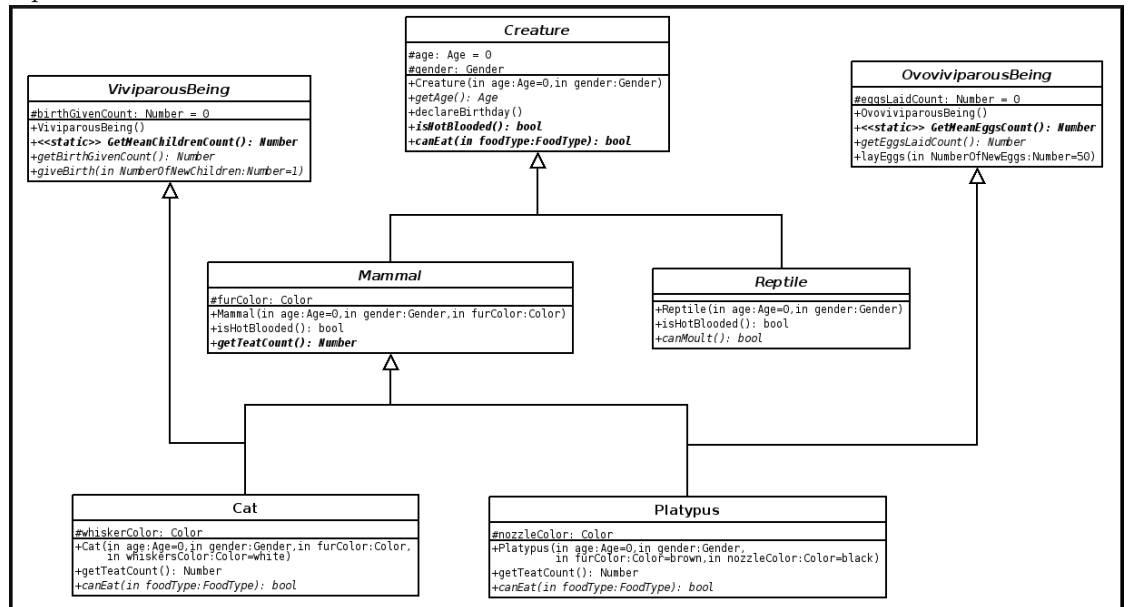
Class implementations

- class_Creature.erl

- class_ViviparousBeing.erl

- class_OvoviviparousBeing.erl

- class_Mammal.erl

- class_Reptile.erl

- class_Cat.erl

- class_Platypus.erl

Tests

Here are a few examples of 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 (e.g. `class_Cat_test.erl`), when WOOPER has already been compiled, one just has to enter, from the corresponding directory: `make class_Cat_run`.

Good Practices

When using WOOPER, the following conventions are deemed useful to respect (even if they are not mandatory).

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`; its type should then go from `T()` to `maybe(T())`.

Class-specific attributes should be specified (using the `class_attributes` define), as doing so brings up much useful information to the developer/maintainer.

Type specifications should be used for at least most non-internal functions (such as constructors, methods, etc.).

There are always welcome, and in some corner cases they are also needed by WOOPER in order to discriminate between equally possible cases (see Exception-Throwing Methods for more information).

The **naming conventions** (e.g. `CamelCase` / `snake_case`) shall be respected; notably, helper functions and static methods (which, from an Erlang point of view, are mostly just exported functions) should be named like C functions, in `snake_case` (e.g. `compute_sum`) rather than being written in CamelCase (e.g. 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 (e.g. `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-level conventions** that we like:

- when, in code, more than one parameter is specified in a function signature, parameter names can be surrounded by spaces (e.g. `f( Color )`, or `g( Age, Height )`)

- functions should be separated by (at least) three newlines, whereas clauses for a given function should be separated by one or two newlines, depending on their size

- 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 (unfortunately) 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.

Supported Platforms

WOOPER can be readily built and run on most Unices (including of course GNU/Linux) and on Windows.

Refer to the Myriad counterpart section for more details.

Troubleshooting

Debug Mode

We recommend that, as a WOOPER user, one enables its debug mode when developing (ensure in `GNUmakevars.inc` that `ENABLE_DEBUG` has been set to true - which is the case by default), as it may catch various user errors more easily (not only WOOPER-internal errors, but also, and most importantly, any user-originating mistake).

Then only, once one's code is mature enough, this debug mode may be disabled in order to obtain best performances.

"No attribute declaration found" whereas a `class_attributes` define exists

Most probably that your `-define(class_attributes,[...]).` define happens *after* the WOOPER header include.

The correct order is:

```
-define(class_attributes,[
          ATTR_DECL1,
          ATTR_DECL2,
          [...]
          ATTR_DECLN]).
[...]
-include("wooper.hrl").
[...]
```

Compilation Error Pointing to `wooper_for_classes.hrl`

As the comment in `wooper_get_class_attributes/0` (in said header file) hints, the `class_attributes` define in the class at hand is most probably incorrect (syntax error).

The compilation error message should help; if it is `syntax error before: '|'`, a union type for an attribute must have been defined with `|`, instead of with the `union` variadic pseudo-operator. See the section about attribute declaration for more details.

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 (e.g. when a state is defined but never re-used later).

Runtime Errors

Most errors while using WOOPER should result in relatively clear messages (e.g. `wooper_method_failed` or `wooper_method_faulty_return`), associated with all relevant runtime information that was available to WOOPER, including context and stacktrace.

Another way of overcoming WOOPER issues is to activate the debug mode for all WOOPER-enabled compiled modules (e.g. uncomment `-define(wooper_debug_mode,).` in `wooper.hrl` or, preferably, ensure in `GNUmakevars.inc` that `ENABLE_DEBUG` has been set to true), 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 (e.g. 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 `myriad` directory 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 (e.g. `table:new()`), for a better substituability (this is obtained thanks to a parse-transform provided by `Ceylan-Myriad`) .

Current Stable Version & Download

Prerequisites

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

- Erlang

- 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`.

Refer to the corresponding Myriad prerequisite section for more precise guidelines, knowing that Ceylan-WOOPER does not need modules with conditional support such as `crypto` or `wx`.

Using Cutting-Edge GIT

This is the installation method that we use and recommend; the WOOPER `master` branch is meant to stick to the latest stable version: we try to ensure that this main line always stays functional (sorry for the pun). Evolutions are to take place in feature branches and to be merged only when ready.

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

Using OTP-Related Build/Runtime Conventions

> **Note**
>
> Over the years, we slowly migrated from our custom, make-based build to integrate more closely to the OTP ecosystem (using applications, releases; then rebar3, hex). Only mixed results were obtained (operations were slower, more complex, more fragile, and not always suitable for "complex" builds involving parse transforms). So, at least for the moment, we prefer mostly relying on our custom build system (still supporting OTP conventions) and experience no drawback.

As discussed for Myriad, we added the (optional) possibility of generating a WOOPER *OTP application* out of the build tree (obtained thanks to the method described in the previous section), ready to be integrated into an *(OTP) release*. For that we rely on rebar3, relx and hex.

OTP conventions mandate using:

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

rather than:

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

Unlike Myriad, which is an OTP *library* application, WOOPER is an OTP *active* application, meaning the reliance on an application that can be started/stopped (`wooper_app`), a root supervisor (`wooper_sup`) and a class manager that is a background server process (`gen_server`) in a supervision tree.

> **Note**
>
> By the way, why are the WOOPER instances not more tightly integrated within the OTP framework? The most natural match would be to have them implement as well the `gen_server` behaviour - yet this would involve non-negligible overhead in terms of additional messages exchanges and function calls, and the API would not as pleasant as the current form (requests would have to be aggregated in `handle_call/3`, oneways in `handle_cast/2`, etc.).

Using Rebar3

> **Note**
>
> In a nutshell: the usual rebar3 machinery is in place and functional, so the only WOOPER prerequisite (Myriad) and WOOPER itself can be obtained simply thanks to:
>
> ```
> $ git clone https://github.com/Olivier-Boudeville/Ceylan-WOOPER.git wooper
> $ cd wooper
> $ rebar3 compile
> ```
>
> Then WOOPER and its tests shall be ready for a successful execution. Note that rebar3 is an alternate way of building WOOPER, as one may rely directly on our make-based system instead.

The same procedures as explained for Myriad apply, once rebar3 is available:

```
$ make rebar3-application
$ make rebar3-release
```

More precisely, to test the WOOPER OTP application support, provided that `make rebar3-compile` was issued beforehand, one can run from the root of the WOOPER source tree:

```
$ cd test
$ make wooper_otp_application_run
        Running unitary test wooper_otp_application_run (third form)
            from wooper_otp_application_test

  --> Testing module wooper_otp_application_test.
```

```
Starting the WOOPER application.
[debug] Starting WOOPER application (type: normal, start arguments: []).
[debug] Starting the WOOPER root supervisor.
[debug] Initializing the WOOPER root supervisor (args: []).
[debug] Starting and linking the WOOPER class manager.
[debug] WOOPER class manager created, as <0.87.0>.
WOOPER version: {2,0,5}.
Class filename corresponding to 'class_Tiger': 'class_Tiger.erl'.
Stopping the WOOPER application.
[debug] Stopping WOOPER application (state: []).
Successful end of test of the WOOPER application.
=INFO REPORT==== 19-Jul-2019::22:53:28.243821 ===
    application: wooper
    exited: stopped
    type: temporary
=INFO REPORT==== 19-Jul-2019::22:53:28.260437 ===
    application: myriad
    exited: stopped
    type: temporary

--> Successful end of test.

(test finished, interpreter halted)
```

One may run `make create-wooper-checkout` in order to create, based on our conventions, a suitable `_checkouts` directory so that rebar3 can directly take into account local, directly available (in-development) dependencies (here, only Myriad).

Using Hex

Hex packages for WOOPER used to be published for most versions, yet finally our workflow does not rely on Hex, so we do not update the Hex packages anymore. Just drop us an email if needing a recent one.

For more details, one may have a look at:

- rebar.config.template, the general rebar configuration file used when generating the WOOPER OTP application and release (implying the automatic management of Myriad)

- rebar-for-hex.config.template, to generate a corresponding Hex package for WOOPER (whose structure and conventions is quite different from the previous OTP elements)

- rebar-for-testing.config.template, the simplest test of the previous Hex package: an empty rebar project having for sole dependency that Hex package

Testing WOOPER

Once Myriad and WOOPER itself have been built (for that refer to either Using Cutting-Edge GIT or Using Rebar3), just run from the root directory of

WOOPER:

```
$ make test
```

The testing shall complete successfully (if it is not the case, see our support section).

> **Note**
>
> WOOPER is built and tested at each commit through continuous integration, and the same holds for its only prerequisite (Myriad). Reciprocally this procedure applies to the projects based on it, directly (e.g. Traces) or not (e.g. US-Web), so in terms of usability, confidence should be high.

Version History & Changes

Version 2.0 [current stable]

Released officially on Sunday, February 3, 2019 initially, and consistently improved since then.

It has been a large rewriting of this layer, with much improvements notably:

- multiple different-arity constructors per class are supported now

- no more `wooper_construct_parameters`, longer `wooper_construct_export` or `wooper_construct_export` defines

- automatic detection and export of constructors, any destructor and methods

- WOOPER method terminators introduced (e.g. `wooper:return_state_result/2`, instead of the `wooper_return_state_result` macro)

- the `class_attributes` optional parse attribute define introduced (`-define(class_attributes,[...])`

- `execute*With` renamed as `execute*As` (clearer)

- convenience method wrappers such as `wooper:execute_request/3` have their parameters reordered (target - either a PID or a passive instance - comes first now)

- passive instances supported (still a bit experimental, not used a lot, yet working)

More generally, many macros and definitions in the WOOPER header files moved to code generated thanks to a parse-transform.

Although this version does not share any code with the various experiments and candidate 2.0 versions that had been previously developed (thanks at least to Ulf and Nicolas in 2010, and to Enrique and Roland in 2013, both operating at that time on the so-called *Zero-Overhead WOOPER* version), these preliminary works surely helped defining the API that we deem suitable now, and that we implemented in the current version.

Version 1.x

Since 2016 we switched back to a "rolling version", not really defining specific release milestones.

Many minor improvements integrated, API enriched in a backward compatible manner.

These versions have been pretty stable very soon, and did the job for nearly a decade (2008-2018), during which various attempts of radical improvements were performed.

Version 1.0

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

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/0` 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

General Principles

Understanding Compilation

WOOPER is the second level of a software stack beginning with Erlang and then Myriad.

If the initial versions of WOOPER were mostly based on macros and headers, newer ones rely on the Erlang way of doing metaprogramming, namely parse-transforms.

More precisely, the sources of a user-defined class are transformed by the standard Erlang toolchain (`erlc` compiler) into an AST (*Abstract Syntax Tree*), which is first transformed by WOOPER (e.g. to generate the new operators, to export any destructor, etc.) and then by Myriad (e.g. to support newer types such as `void/0`, `maybe/1` or `table/2`), which also provides much of the AST transformation support.

Understanding the Mode of Operation of a WOOPER Instance

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 of that class[27], 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).

Taking class_Platypus.erl as a (small) example (shallow inheritance tree, and just a few attributes), by uncommenting traces in the class manager, we can see that the size of the `class_Platypus` virtual table is 1040 bytes.

At runtime, after its full construction, the total size of a Platypus instance (i.e. the overall size of its corresponding process) is 8712 bytes.

Knowing that (with OTP 23, AMD64 on GNU/Linux) the size of a blank process is 2688 bytes[28], the actual payload specific to this instance is `8712-2688=6024 bytes` (and the virtual table accounts for roughly 17% of it).

After having used the persistent_term module (see our `persistent_term` branch) to share these (immutable) per-class virtual tables, the size of the same test Platypus instance became 6840 bytes, corresponding thus to a per-instance shrinking of `8712-6840=1872 bytes` here.

When the class manager registers (once for all) the class_Platypus virtual table, the size of the persistent_term registry grows of 1072 bytes (which is consistent with a virtual table of 1040 bytes).

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.

---

[27] 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.

[28] Measured with `P=spawn(basic_utils, freeze, [])`, then `basic_utils:get_process_size(P)`.

Issues & Planned Enhancements

- integrate automatic persistent storage of instance states, for example in Mnesia databases

- integrate specific constructs for code reflection

- check that a class specified in `execute*As` is indeed a (direct or not) mother class of this one, at least in debug mode

- check that the declared attributes are legit (existing, not reserved, etc.) and their access as well (e.g. regarding constness)

- support qualifier-based declarations of methods and attributes (`public`, `protected`, `private`, `final`, `const`, `pure`, etc.)

- generate automatically a graphical class diagram out of the sources of a set of projects (e.g. using PlantUML)

- 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`; note that this should be the case now that the use of `persistent_term` has been integrated

Licence

Ceylan-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:

- the 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)

- the GNU General Public License (GPL), version 3.0 or later

- the 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 (e.g. 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

- ECT, an Object-Oriented Extension to Erlang, very promising yet apparently not maintained anymore

- 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**, and to make 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 (with on OMG IDL to Erlang Mapping)

WOOPER and Orber are rather different beasts, though: WOOPER is quite lightweight (less than 25 000 lines of code, including blank lines, numerous comments, tests and examples), does not involve a specific (IDL) compiler generating several stub/skeleton Erlang files, nor depends on OTP or on Mnesia (but depends on Myriad), whereas Orber offers a full, standard, 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`; warning: probably a dead link now).

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[29].

---

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

Support

Bugs, questions, remarks, patches, requests for enhancements, etc. are to be reported to the project interface (typically issues) or directly at the email 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 Ceylan-WOOPER!