WOOPER

Wrapper for Object-Oriented Programming in Erlang

This documentation is also available directly from the WOOPER homepage.
Latest stable WOOPER archives are:

- wooper-0.3.tar.bz2
- wooper-0.3.zip

Table of Contents

1 Overview

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

WOOPER is a (completely autonomous) part of the Ceylan project.

1.1 Motivations & Purpose

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

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

1.2 WOOPER Mode of Operation In A Nutshell

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

WOOPER concept	**Corresponding Erlang mapping**
class definition	module
instance	process
instance reference	process identifier (PID)
new operators	WOOPER-provided functions, making use of user-defined `construct/N` function
delete operators	WOOPER-provided functions, unless user-specified
method definition	module functions that respect some conventions
method invocation	sending of appropriate inter-process messages
method look-up	class-specific virtual table taking into account inheritance transparently
instance state	instance-specific hashtable kept by the instance-specific WOOPER tail-recursive infinite loop
instance attributes	key/value pairs stored in the instance hashtable
class (static) method	module function

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

1.3 Example

Here is a simple example of how WOOPER instances can be managed. This shows `new/delete` operators, method calling (both request and oneway), and inheritance (a cat is here a viviparous mammal, as defined in the example below):

```
-module(class_Cat).

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

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

% Declaring all variations of WOOPER standard life-cycle operations:
% (template pasted, just two replacements performed to update arities)
-define( wooper_construct_export, new/4, new_link/4,
    synchronous_new/4, synchronous_new_link/4,
    synchronous_timed_new/4, synchronous_timed_new_link/4,
    remote_new/5, remote_new_link/5, remote_synchronous_new/5,
    remote_synchronous_new_link/5, remote_synchronous_timed_new/5,
    remote_synchronous_timed_new_link/5, construct/5 ).

% Method declarations.
-define( wooper_method_export, getTeatCount/1, canEat/2,
  getWhiskerColor/1 ).

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

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

% No guarantee on biological fidelity:
getTeatCount(State) ->
    ?wooper_return_state_result( State, 6 ).

% Cats are supposed carnivorous though:
canEat(State,soup) ->
    ?wooper_return_state_result( State, true );

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

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

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

% This method is cat-specific:
getWhiskerColor(State)->
    ?wooper_return_state_result( State, ?getAttr(whisker_color) ).
```

Straightforward, isn't it? We will discuss it in-depth though.
To test this class, just extract your WOOPER archive, like in:

```
tar xvjf wooper-x.y.tar.bz2
cd wooper-x.y
```

and run, from the root of this archive (the `wooper-x.y` directory):

```
make all && cd wooper/examples && make class_Cat_run
```

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

```
[..]
Deleting cat <0.41.0>! (overridden destructor)
Deleting mammal <0.41.0>! (overridden destructor)
--> This cat could be created and be synchronously deleted, as expected.
--> End of test for module class_Cat.
```

2 Why Adding Object-Oriented Capabilities To Erlang?

Although applying blindly OOP with 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 actor types have also often to share numerous states and behaviours, while being able to specialise them on a per-type basis.

The example chosen here is a simulation of the interacting lives of numerous animals from various species. Obviously, they have to share behaviours (ex: all ovoviviparous beings may lay eggs, all creatures can live and die, all have an age, etc.), which cannot be mapped easily (read: automatically) to Erlang concepts without adding some generic constructs.

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

The same programs could 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.

3 How to Use WOOPER: Detailed Description & Concept Mappings

3.1 Classes

3.1.1 Class & Names

A class is a blueprint to create objects, a common scheme describing the behaviour and the internal data types of its instances, i.e. the attributes and methods that the created objects all share.

With WOOPER each class must have a unique name.

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

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

The class name can be obtained through its `get_class_name` WOOPER-defined static method:

```
> class_Cat:get_class_name().
class_Cat
```

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

3.1.2 Inheritance & Superclasses

A WOOPER class can inherit from other classes, in this case the behaviour and the internal data defined in the mother classes are available by default to this child class.

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

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

As for our cat, this animal could be modelled both as a mammal (itself a specialised creature) and a viviparous being[1]. Hence its mother classes could be described as: `-define(wooper_superclasses,[class_Mammal, class_ViviparousBeing]).`

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

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

3.2 Instances

3.2.1 Instance Mapping

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

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

3.2.2 Instance State

Another need is to rely on **state management** and **encapsulation**: each instance should be stateful, have its state 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.

3.3 Methods

Instances may declare **methods** that can be publicly called, whether locally or remotely (i.e. on other networked computers, like with RMI or with CORBA). Distribution is seamlessly managed thanks to Erlang.

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

For example, our cat may define following methods:

[1] Neither of them is a subset of the other, these are mostly unrelated concepts; at least in the context of that example!

- `canEat`, taking one parameter specifying the type of food, and returning whether the cat can eat that kind of food. The implementation should be cat-specific here, whereas the method signature is shared by all beings
- `getWhiskersColor`, taking no parameter, returning the color of its whiskers. This is indeed a purely cat-specific method
- `declareBirthday`, incrementing the age of our cat, not taking any parameter nor returning anything. It will be therefore be implemented as a oneway call (i.e. not returning any result to the caller, hence not even needing to know it), only interesting for its effect on the cat state: here, making it one year older

Declaring a birthday is not cat-specific, nor mammal-specific: we can consider it being creature-specific. Cat instances should then inherit this method, preferably indirectly from the `class_Creature` class, in all cases without having to specify anything, since the `wooper_superclasses` define already implies it.

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.

3.3.1 Method Declaration

The cat-specific methods are to be declared:

- in the `class_Cat`
- thanks to the `wooper_method_export` clause

Their arity should be equal to the number of parameters they should be called with, plus one.

The additional parameter is an implicit one (automatically managed by WOOPER), corresponding to the state of the 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 to the parameters of incoming method calls.

In our example, the declarations would result in: `-define( wooper_method_export, canEat/2, getWhiskersColor/1 ).`.

As `declareBirthday` will be inherited but not overridden, no need to declare it.

Some method names are reserved for WOOPER: no user method should have a name starting by `wooper`.

The complete list of reserved function names that do not start with the `wooper_` prefix is:

- `get_class_name`
- `get_superclasses`
- `executeRequest`
- `executeOneway`
- `delete_any_process_in`
- `is_wooper_debug`

They are reserved for all arities.

3.3.2 Method Invocation

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

All methods, either defined directly by the actual class or inherited, are to be called thanks to a proper Erlang message.

When the caller needs a result to be sent back, it must specify to the instance what is its PID (i.e. the caller PID), so that the instance knows to whom the answer should be sent.

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

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

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

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

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

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

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

3.3.3 Method Name

Methods are designated by their atom name, as declared in the `wooper_method_export` clause of the class in the inheritance tree that defines them.

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

3.3.4 Method Parameters

As detailed below, there are:

- *requests* methods: they perform some processing and then return a result to the caller
- *oneway* methods: they only change the state of the instance, with no reply being sent back

Both can take any number of parameters, including none. The **marshalling** of these parameters and, if relevant, of returned values is performed automatically by Erlang.

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

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

Note

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

3.3.5 Two Kinds of Methods

3.3.5.1 Request Methods

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

`receive` should then be used by the caller to retrieve the request result, like in:

```
MyPoint ! {getCoordinates,[],self()},
receive
        {wooper_result,[X,Y]} ->
                [...];
        % Could be left out to ignore errors.
        % Otherwise one might prefer making this caller block:
        Error ->
                [...]
end,
```

3.3.5.2 Oneway Methods

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 would be 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 do 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.

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.

3.3.6 Method Results

3.3.6.1 Execution Success: `{wooper_result,ActualResult}`

If the execution of a method succeeded, and if it is a request (not a oneway, which would not return anything), then `{wooper_result,ActualResult}` will be sent back.

Otherwise one of the following error messages will be emitted.

3.3.6.2 Execution Failures

When the execution of a method fails, three main error results can be returned.

A summary could be:

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

3.3.6.2.1 `wooper_method_not_found`

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

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

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

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

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

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

3.3.6.2.2 `wooper_method_failed`

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

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

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

Such a method error means there is a runtime failure, it is deemed a instance-side issue (the caller should not be responsible for it), 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.

3.3.6.2.3 `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 when no other error matched (this is a catch-all case).

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

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

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

3.3.6.2.4 Caller-Side Error Management

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

```
MyPoint ! {getCoordinates,[],self()},
receive
    {wooper_result, [X,Y] } ->
             [...];
    {wooper_method_not_found, Pid, Class, Method, Arity, Params} ->
             [...];
    {wooper_method_failed, Pid, Class, Method, Arity, Params, ErrorTerm} ->
             [...];
    % Error term can be a tuple {Pid,Error} as well, depending on the exit:
    {wooper_method_failed, Pid, Class, Method, Ar-
ity, Params, {Pid,Error}} ->
             [...];
    {wooper_method_faulty_return, Pid, Class, Method, Arity, Params, Unex-
pectedTerm} ->
             [...];
    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[2]:

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

```
                [...]
end.
```

3.3.7 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,Number) -> [..]`.

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

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

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

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

State is unchanged here.

Note that when a method "returns" the state of the instance, it returns it to the (local, process-wise) private WOOPER-based main loop of that instance: in other words, the state variable will never be exported outside of its process. Encapsulation is ensured, as the instance is the only process able to access its own state. On method ending, the instance then just loops again, with an updated state.

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.

More precisely, depending on its returning a specific result, the method signature will correspond either to a request or a oneway, and will use, respectively, either the `wooper_return_state_result` or the `wooper_return_state_only` macro.

3.3.7.1 For Requests

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

For example:

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

All methods are of course given the parameters specified at their call.

For example, we can declare:

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

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

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

Requests can access to one more information than oneways: the PID of the caller that sent the request.

This can be done by using the `getSender` macro, which is automatically set by WOOPER:

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

[2] Then, in case of failure, the method call will become blocking.

Thus requests have access to their caller PID without having to specify it twice, i.e. with no need to specify it in the parameters as well as in the third element of the call tuple: instead of `MyCat ! {giveBirth,[2,3,self()],self()}.`, only `MyCat ! {giveBirth,[2,3],self()}.` can be used, while still letting the possibility for the called request (here `giveBirth/3`, for a state and two parameters) to access the caller PID thanks to the `getSender` macro, and maybe store it for a later use.

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

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

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

3.3.7.2 For Oneways

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

For example:

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

can be called that way:

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

Oneways may leave the state unchanged, only being called for side-effects, for example:

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

3.3.7.3 Usefulness Of These Two Return Macros

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

For example, in debug mode, `?wooper_return_state_result(AState,AResult)` will simply translate into `{wooper_result,AState,AResult}`, and when the execution of the method is over, WOOPER will attempt to match the method returned value with that triplet (3-tuple).

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

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

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

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

3.3.8 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**, which are possibly overloaded.

For example, when developing a `declareBirthday` method of `class_Mammal` (which among other things has to increment the mammal age), you may want to perform a call to the `setAge` method of the current instance.

If you just call `setAge` or `class_Mammal:setAge`, then you will never call the potentially overloaded versions from child classes: if an instance of child class `class_Cat` (which inherited `declareBirthday` "as is") overloaded `setAge`, you may want that `declareBirthday` calls automatically `class_Cat:setAge` instead of `class_Mammal:setAge`.

This call can be easily performed asynchronously: a classical message-based method call can be used, like in `self() ! {setAge,10}`. If this approach is useful when not directly needing from the method the result of the call and/or not needing to have it executed at once, there are cases when one wants to have that possibly overridden method being executed *directly* and to access to the corresponding modified state and, possibly, output result.

In these cases, one should call the WOOPER-defined `executeRequest` or `executeOneway` function, depending of 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 a call made by an instance to itself) and that no message exchange 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/3` or `executeRequest/2`, its parameters being the current state, the name of the request-method, and, if specified, 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:

- request taking more than one parameter: `{NewState,Result} = executeRequest(CurrentState, my_request_name, [ "hello", 42 ])`
- request taking exactly one parameter: `{NewState,Result} = executeRequest(CurrentState, another_request_name, 42)`
- request taking no parameter: `{NewState,Result} = executeRequest(CurrentState, third_request_name)`

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

`executeOneway` returns the new state.

For example:

- oneway taking more than one parameter: `NewState = executeOneway(CurrentState,my_oneway_name,[ "hello", 42 ])`
- oneway taking exactly one parameter: `NewState = executeOneway(CurrentState,another_oneway_name,4`
- oneway taking no parameter: `NewState = executeOneway(CurrentState,third_oneway_name)`

3.4 State Management

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

Its state is only directly accessible from inside its dedicated class module: the state of an instance is private, the outside can access to it only through the methods declared by the class of this instance.

An instance state (the one which is given by WOOPER as the `State` variable, first parameter of all methods) is defined as a **set of attributes**.

Each attribute is designated by a name and has a mutable value, which can be any Erlang term.

The current state of an instance can be thought as a list[3] of `{attribute_name,attribute_value}` pairs, like in: `[ {color,black} , {age,5} , {name,"Tortilla"} ]`.

A set of macros allows to operate on these state variables, notably to read and write the attributes they contain.

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

This current state can be then modified in the method, and its updated version will be returned locally to WOOPER, thanks to the final call in the method, one of the two `wooper_return_state_*` macros.

Then the code automatically instantiated by the WOOPER header in the class implementation will loop again with the updated state for this instance, waiting for the next method call.

3.4.1 Modifying State

3.4.1.1 The `setAttribute/3` Macro

Setting an attribute (creating and/or modifying it) should be done with the **setAttribute** macro: `?setAttribute(AState,AttributeName,NewAttributeValue)`.

For example, `AgeState = ?setAttribute(State,age,3)` will return a new state, bound to `AgeState`, exact copy of `State` (with all the attribute pairs equal) but for the `age` attribute, whose value will be set to 3 (whether or not this attribute was already defined in `State`).

Therefore during the method execution multiple states can be defined (ex: `State` and `AgeState`), 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 hashtables (ex: `State` and `AgeState`) actually **share** all their terms except the one modified, thanks to the immutability of Erlang variables, which allows to reference rather than copy.

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

A better solution is to use the **setAttributes** macro (note the plural) to set a list of attribute name/attribute value pairs.

For example, `ConstructedState = ?setAttributes(State,[ {age,3}, {whisker_color,white} ])` will return a new state, exact copy of `State` but for the listed attributes, set to their respective specified value.

3.4.1.2 The `removeAttribute/2` Macro

An attribute may also be removed, using the **removeAttribute** macro.

For example: `NewState = ?removeAttribute(State,an_attribute)`. The resulting state will have no key corresponding to `an_attribute`.

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

[3] Actually it is a hashtable, for efficiency reasons. It uses the hash value of a key (like the `age` key) as an index used to find the corresponding value (here, `5`) in the relevant bucket of the table. The point is that this look-up is performed in constant time on average, regardless of how many key/value pairs are stored in the table, whereas most data structures, like plain lists, will have look-up durations that will increase with the number of pairs they contain, thus being most often slower than their hashtable-based counterparts.

3.4.2 Reading State

3.4.2.1 The `hasAttribute/2` Macro

To test whether an attribute is defined, use the **hasAttribute** macro: `?hasAttribute(AState,AttributeName)`, which returns either `true` or `false`, and cannot fail.

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

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

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

3.4.2.2 The `getAttribute/2` Macro

Getting the value of an attribute should be done with the **getAttribute** macro: `?getAttribute(AState,AttributeName)`

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

The requested attribute may not exist in the specified state. In this case, a bad match is triggered.

In the previous example, if the attribute `whisker_color` had not been defined, then `getAttribute` would have returned: `{{badmatch,{hashtable_key_not_found,whisker_color}},[{hashtable,getEntry,2},...`

3.4.2.3 The `getAttr/2` Macro

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

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

3.4.3 Read-Modify-Write Operations

Some more helper macros are provided for the most common operations, to keep the syntax as lightweight as possible.

3.4.3.1 The `addToAttribute/3` Macro

The corresponding signature is `addToAttribute(State,AttributeName,Value)`: when having a numerical attribute, adds specified number to the attribute.

For example: `NewState = ?addToAttribute(State,a_numerical_attribute,6)`.

If the target attribute does not exist, will trigger `{{badmatch,undefined},[{hashtable,addToEntry,3},...`

If it exists but no addition can be performed on it (meaningless for the type of the current value), will trigger `{badarith,[{hashtable,addToEntry,3},...`

3.4.3.2 The `substractFromAttribute/3` Macro

The corresponding signature is `substractFromAttribute(State,AttributeName,Value)`: when having a numerical attribute, subtracts specified number from the attribute.

For example: `NewState = ?substractFromAttribute(State,a_numerical_attribute,1)`.

If the target attribute does not exist, will trigger `{{badmatch,undefined},[{hashtable,substractFromEntry,3},..`

If it exists but no subtraction can be performed on it (meaningless for the type of the current value), will trigger `{badarith,[{hashtable,substractFromEntry,3},...`

3.4.3.3 The `toggleAttribute/2` Macro

The corresponding signature is `toggleAttribute(State,BooleanAttributeName)`: when having a boolean attribute, whose values are either `true` or `false`, sets the opposite logical value to the current one.

For example: `NewState = ?toggleAttribute(State,a_boolean_attribute)`.

If the target attribute does not exist, will trigger `{{case_clause,undefined},[{hashtable,toggleEntry,2},...`

If it exists but is neither true or false, will trigger `{{case_clause,{value,..}},[{hashtable,toggleEntry,2},...`

3.4.3.4 The `appendToAttribute/3` Macro

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

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

If the target attribute does not exist, will trigger `{{badmatch,undefined},[{hashtable,appendToEntry,3},...`

If it exists but is not already a list, it will not crash but will create an ill-formed list (ex: `[8|false]` when appending 8 to `false`, which is not a list).

3.4.3.5 The `deleteFromAttribute/3` Macro

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

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

If the target attribute does not exist, will trigger `{{badmatch,undefined},[{hashtable,deleteFromEntry,3},...`

If it exists but is not already a list, it will trigger `{function_clause,[{lists,delete,[..,..]},{hashtable,delete`

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

3.4.3.6 The `popFromAttribute/2` Macro

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

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

3.4.3.7 The `addKeyValueToAttribute/4` Macro

The corresponding signature is `addKeyValueToAttribute(State,AttributeName,Key,Value)`: when having an attribute whose value is a hashtable (therefore, it is a hashtable in the WOOPER hashtable), adds specified key/value pair to that hashtable attribute.

For example: `WithTableState = ?setAttribute( State, my_hashtable, hashtable:new() ), NewState = ?addKeyValueToAttribute( WithTableState, my_hashtable, my_key, my_value )` will result in having the attribute `my_hashtable` in state variable `WithTableState` being an hashtable with only one entry, whose key is `my_key` and whose value is `my_value`.

See wooper.hrl for the actual definition of most of these WOOPER constructs.

3.5 Multiple Inheritance & Polymorphism

3.5.1 The General Case

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

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

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

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

should output some like:

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

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

3.5.2 The Special Case of Diamond-Shaped Inheritance

In the case of a diamond-shaped inheritance, as the method table is constructed in the order specified in the declaration of the superclasses (`-define(wooper_superclasses,[class_X,class_Y, etc.]).`), and as child classes override mother ones, when an incoming WOOPER message arrives the selected **method** should be the one defined in the last 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 in the child class at the base of the diamond so that they perform explicitly a direct call to the wanted module is by far the most reasonable solution, in terms of clarity and maintainability.

Regarding the instance state, the **attributes** are set by the constructors, therefore the developer can select in which order the direct mother classes should be constructed. However it always leads to calling multiple times the constructor of the class that sits at the top of the diamond. Any side-effect it would induce would then occur as many times as this class is a common ancestor of the actual class.

Note

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

3.6 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):

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

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

3.6.1.1 Role of a new/construct Pair

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

All calls to a `new` operator result in an underlying call to the corresponding `construct` method.

For example, `MyCat = class_Cat:new(A,B,C,D)` will rely on `class_Cat:construct/5` to set-up a proper initial state for the `MyCat` instance: `class_Cat:construct(State,A,B,C,D)` will be called for

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`.

The `new` and `new_link` operators are automatically defined by WOOPER, but they rely on the class-specific user-defined `construct` special method (only WOOPER is expected to call this method). This `construct` method is the one that must be implemented by the class developer.

Only one version of `new`, `new_link` and `construct` can be defined, but they may branch to as many subconstructors as needed.

For example:

```
MyFirstDog  = Class_Dog:new( create_from_colors, [sand,white] ),
MySecondDog = Class_Dog:new( create_from_age, 5 ),
MyThirdDog  = Class_Dog:new( create_from_weight, 4.4 ).
```

3.6.1.2 The Various Ways of Creating An Instance

As shown with the `new_link` operator, even with the same set of constructing parameters, many variations of `new` can be imagined: 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 construction needs N actual parameters, the following construction operators are built-in:

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

– blocking with time-out: `remote_synchronous_timed_new/N+1` and `remote_synchronous_timed_new_lin`

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

The supported `new` variations are detailed below.

3.6.1.2.1 Asynchronous new

This corresponds to the plain `new`, `new_link` operators etc. discussed earlier. These basic operators are **asynchronous** (non-blocking): they trigger the creation of a new instance, and return immediately, without waiting for it to complete.

3.6.1.2.2 Synchronous new

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

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

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

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

The implementation of these synchronous operations relies on a message (`{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.

3.6.1.2.3 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.

This is why the `*synchronous_timed_new*` operators are defined: if the time-out (its default duration is 5 seconds) expires while waiting for the created instance to answer, then they will return the `time_out` atom instead of the PID of the created instance.

The caller is then able to check whether the creation succeeded thanks to a simple pattern-matching.

3.6.1.2.4 Remote new

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

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

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

3.6.1.3 Declaration of the new/construct Pair

When an instance is created, user-specified parameters are given to the relevant `new` operator, notably to set up the instance initial state, i.e. its attributes. These parameters must be declared thanks to the `wooper_construct_parameters` define.

For example, `-define( wooper_construct_parameters, Age, Gender ).` implies that two parameters, an age and a gender, are needed for an instance to be created.

If the instance is created without needing any parameter, then no `wooper_construct_parameters` macro should be defined at all (i.e. using `-define( wooper_construct_parameters,).` is **not** allowed).

In the general case where there is at least one parameter, the WOOPER-defined `new` operators automatically transmit these parameters to the `construct` method.

Based on the `wooper_construct_parameters` define (let us suppose N constructor parameters are listed in it), all the variations of the `new` operator and `construct` can be declared thanks to:

```
% Declaring all variations of WOOPER standard life-cycle operations:
% (template pasted, two replacements performed to update arities)
-define( wooper_construct_export, new/N, new_link/N,
    synchronous_new/N, synchronous_new_link/N,
    synchronous_timed_new/N, synchronous_timed_new_link/N,
    remote_new/N+1, remote_new_link/N+1, remote_synchronous_new/N+1,
    remote_synchronous_new_link/N+1, remote_synchronous_timed_new/N+1,
    remote_synchronous_timed_new_link/N+1, construct/N+1 ).
```

As there are quite a lot of construction operators available, the recommended mode of operation of declaring them all is to use the following template:

```
% Declaring all variations of WOOPER standard life-cycle operations:
% (template pasted, two replacements performed to update arities)
-define( wooper_construct_export, new/A, new_link/A,
    synchronous_new/A, synchronous_new_link/A,
    synchronous_timed_new/A, synchronous_timed_new_link/A,
    remote_new/B, remote_new_link/B, remote_synchronous_new/B,
    remote_synchronous_new_link/B, remote_synchronous_timed_new/B,
    remote_synchronous_timed_new_link/B, construct/B, delete/1 ).
```

and to use your text editor to replace `A` with your actual `N` and `B` with `N+1`.

If the class does not declare a specific destructor, then the corresponding declaration in the previous template (`delete/1`) must be removed.

For example, if wanting to create a bird (`MyBird = class_Bird:new(Age,Gender)`), we would have `N=2` (two actual constructing parameters: age and gender), thus `A` has to be replaced by `2`, and B by `3`:

We see that `construct` needs `N+1` parameters instead of `N`, to account for its first additional parameter, which is the `State` variable.

3.6.1.4 Some Examples

They can be used as templates:

```
-module(class_Bird).
-define( wooper_superclasses, [class_X,class_Y] ).
-define( wooper_construct_parameters, Age, Gender ).

% Declaring all variations of WOOPER standard life-cycle operations:
% (template pasted, two replacements performed to update arities)
-define( wooper_construct_export, new/2, new_link/2,
    synchronous_new/2, synchronous_new_link/2,
    synchronous_timed_new/2, synchronous_timed_new_link/2,
    remote_new/3, remote_new_link/3, remote_synchronous_new/3,
    remote_synchronous_new_link/3, remote_synchronous_timed_new/3,
    remote_synchronous_timed_new_link/3, construct/3, delete/1 ).

% Method declarations.
```

```
-define( wooper_method_export, my_first_method/X, second_method/Y ).

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

% Using the construct macro is prefered to duplicating the list of
% parameters:
construct( State, ?wooper_construct_parameters ) ->
  ...
```

If no construction-related parameter was needed, then it would become:

```
-module(class_OtherBird).
-define( wooper_superclasses, [class_X,class_Y] ).
% No '-define( wooper_construct_parameters,).' here!

% Declaring all variations of WOOPER standard life-cycle operations:
% (template pasted, two replacements performed to update arities)
-define( wooper_construct_export, new/0, new_link/0,
    synchronous_new/0, synchronous_new_link/0,
    synchronous_timed_new/0, synchronous_timed_new_link/0,
    remote_new/1, remote_new_link/1, remote_synchronous_new/1,
    remote_synchronous_new_link/1, remote_synchronous_timed_new/1,
    remote_synchronous_timed_new_link/1, construct/1, delete/1 ).

% Method declarations.
-define( wooper_method_export, my_first_method/X, second_method/Y ).

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

construct( State ) ->
  ...
```

All variations of the `new` operator are always defined automatically by WOOPER: nothing special is to be done for them, besides the `wooper_construct_export` declaration we just mentioned here.

3.6.1.5 Definition of the `construct` Method

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

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

For example, `class_Cat` inherits directly from `class_Mammal` and from `class_ViviparousBeing`, and has only one attribute (`whisker_color`) on its own:

```
[..]
-define(wooper_superclasses,[class_Mammal,class_ViviparousBeing]).
[..]
-define(wooper_construct_parameters,Age,Gender,FurColor,WhiskerColor).
[..]
-define( wooper_construct_export, new/4, new_link/4,
    synchronous_new/4, synchronous_new_link/4,
    synchronous_timed_new/4, synchronous_timed_new_link/4,
```

```
    remote_new/5, remote_new_link/5, remote_synchronous_new/5,
    remote_synchronous_new_link/5, remote_synchronous_timed_new/5,
    remote_synchronous_timed_new_link/5, construct/5 ).

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

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

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

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

This would be correct:

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

Recommended form:

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

3.6.2 Instance Deletion

3.6.2.1 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.

Conversely, with WOOPER, when defining a destructor for a class (`delete/1`), one only has to specify what are the *specific* operations (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.

Note also that as soon as you define a destructor, you have to declare it in the `wooper_construct_export` section.

For example:

```
-define( wooper_construct_export, new/N, [..], construct/N+1, delete/1 ).
```

Otherwise a warning will be issued (`delete/1` is unused), and the overridden destructor would not be called then.

3.6.2.2 Asynchronous Destructor: `delete/1`

More precisely, either the class implementer does not define at all a `delete/1` operator, or it defines it without needing to call the ones of the mother class(es), like in:

```
delete(State) ->
  io:format("An instance of class ~w is being deleted now!", [?MODULE] ).
```

In both cases, when the instance will be deleted (i.e. `MyInstance ! delete` is issued), WOOPER will take care of:

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

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

3.6.2.3 Synchronous Destructor: `synchronous_delete/1`

Finally, the `delete/1` operator does not need to be exported, since it will be triggered only thanks to messages.

4 Miscellaneous Technical Points

4.1 EXIT Messages

A class instance may receive EXIT messages from other processes.

A given class can process these EXIT notifications:

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

For example:

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

results in:

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

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

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

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

4.2 Practical Build Hints

All WOOPER classes must include wooper.hrl: `-include("wooper.hrl").`.

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

One should have `utils.beam`, `hashtable.beam` and `wooper_class_manager.beam` available to the interpreter before using WOOPER-based classes.

On UNIX-like platforms, using the Makefiles included in the WOOPER archive is recommended.

One just has to go at the root of the sources (from an extracted archive, you are expected to be in the `wooper-x.y` root directory) and simply run: `make` (assuming GNU make is available, so that the WOOPER `GNUmakefile` is used).

On other platforms, these modules must be compiled one way or another before using WOOPER. For example:

```
1> c(hashtable).
{ok,hashtable}
```

We provide as well a WOOPER-aware neditrc configuration file for syntax highlighting (on black backgrounds), inspired from Daniel Solaz's Erlang Nedit mode.

4.3 Similarity With Python

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

5 WOOPER Example

We created a small set of classes allowing to show multiple inheritance:

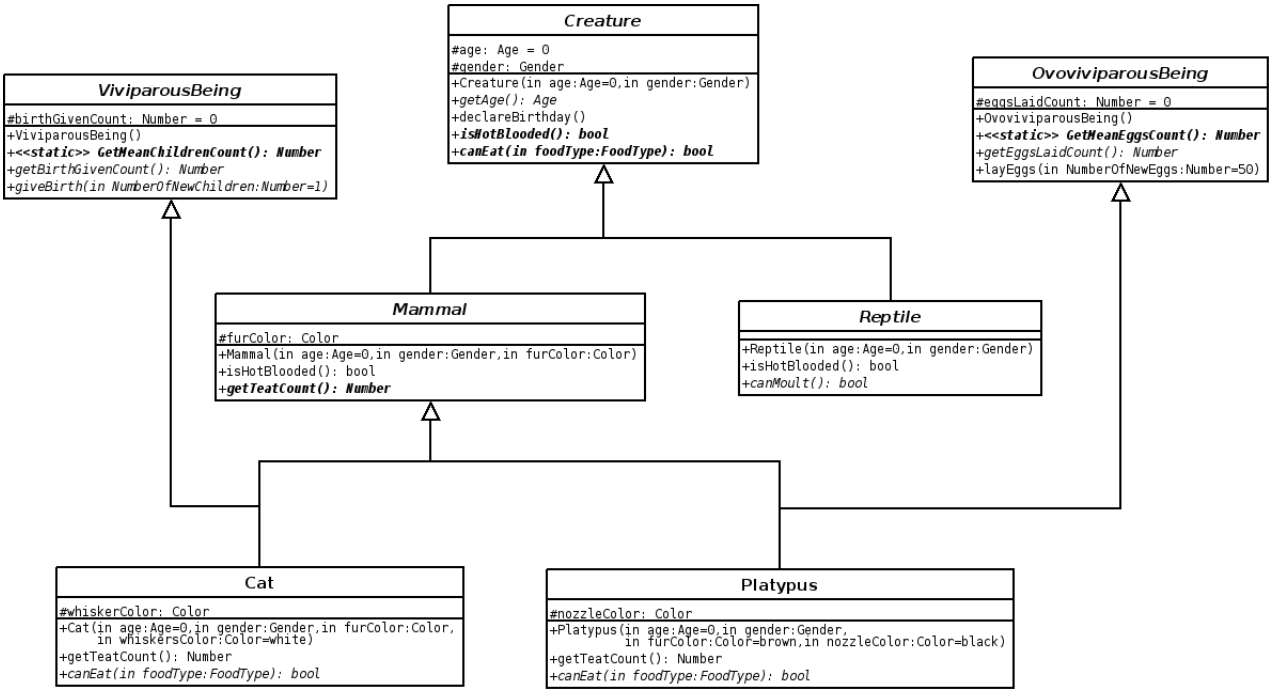

Figure 1: Example of an inheritance graph to be handled by WOOPER

5.1 Class implementations

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

5.2 Tests

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

- class_Platypus_test.erl

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

6 Troubleshooting

6.1 General Case

6.1.1 Compilation Warnings

A basic rule of thumb in all languages is to enable all warnings and eradicate them before ever 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 allow to catch mistakes when state variables are being properly taken care of.

6.1.2 Runtime Errors

Most errors while using WOOPER should result in relatively clear messages (ex: `wooper_method_failed` or `wooper_method_faulty_return`).

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

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

Then all incoming method calls will be traced, for easier debugging.

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

6.2 Mismatches In Method Call

6.2.1 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), which most probably would not 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_Cat,wooper_main_loop,1}]}`.

6.2.2 List Parameter Incorrectly Specified In Call

As explained in the single method parameter is a list 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.

6.2.3 Error With Exit Value: `{undef,[{hashtable,new,[..]}..`

You most probably forgot to build the `common` directory, which, among other things, contains the `hashtable.erl` file.

Check that you have a `hashtable.beam` file indeed, and that it can be found from the paths specified to the interpreter.

7 Current Stable Version & Download

7.1 Using Stable Release Archive

WOOPER 0.2 is ready to be used and can be downloaded here.

Either a `.zip` or a `.tar.bz2` can be retrieved. Starting from the 0.1 version, WOOPER should be fully functional (pun intended!).

One way of building all of WOOPER (base files and examples) is, from UNIX or on Windows from a Cygwin or MSYS shell, once the archive is downloaded and extracted, to execute `make all` from the WOOPER directory.

For example:

```
tar xvjf wooper-x.y.tar.bz2 && cd wooper-x.y && make all
```

It will build and run all, including the various WOOPER test cases.

7.2 Using Cutting-Edge SVN

A SVN (anonymous) check-out of WOOPER code can be obtained thanks to, for example:

```
svn co https://ceylan.svn.sourceforge.net/svnroot/ceylan/Ceylan/trunk/src/code/scripts/erlang W
code-checkout
```

Ceylan developers should used their Sourceforge user name so that they can commit changes:

```
svn co --username Your_SF_User_Name https://ceylan.svn.sourceforge.net/svnroot/ceylan/Ceylan/tr
code-checkout
```

Check-out of WOOPER documentation can be performed thanks to:

```
svn co --username YourSFUserName https://ceylan.svn.sourceforge.net/svnroot/ceylan/Ceylan/trunk
doc-checkout
```

If just wanting a SVN anonymous export, use for example:

```
svn export http://ceylan.svn.sourceforge.net/svnroot/ceylan/Ceylan/trunk/src/code/scripts/erlan
code-export
```

and:

```
svn export http://ceylan.svn.sourceforge.net/svnroot/ceylan/Ceylan/trunk/src/doc/web/main/docum
doc-export
```

8 Version History & Changes

Versions

8.1 Version 0.4 [cutting-edge, in SVN only]

Not released yet (work-in-progress).

Should be mainly a BFO (*Bug Fixes Only*, if bugs were to be found) version, as functional coverage is pretty complete already.

8.2 Version 0.3 [current stable]

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

8.3 Version 0.2

Released on Friday, December 21, 2007. Still fully functional!

Main changes are:

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

8.4 Version 0.1

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

9 WOOPER Inner Workings

Each instance runs a main loop (`wooper_main_loop`, 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(S) ->
  receive
      {command,{M,P}} ->
          S_new = f(S,M,P),
          my_server( S_new )
  end.
```

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

The per-instance kept state is twofold.

9.1 Method Virtual Table

This per-class hashtable 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 hashtable 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 inheritance tree, that defines that method.

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

As the table is built only once and is shared by all instances[4], it adds very little overhead, space-wise and time-wise. Thanks to the hashtable, method look-up is efficient too?

9.2 Attribute Table

This is another hashtable, this time per-instance.

Keys are attribute names, values are attribute values.

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

WOOPER relies only on these specific files:

- wooper.hrl: the WOOPER core, which gives to the modules using it all the OOP constructs discussed
- wooper_class_manager.hrl: the tiny class manager header
- wooper_class_manager.erl: the class manager itself, the unique process that is automatically spawned to share virtual tables among instances on a node
- hashtable.erl: efficient associative table used by WOOPER for virtual tables, methods, attributes
- utils.erl: a small module used by the hashtable and other modules

The latest two come from the `common` sub-module.

[4] Provided that Erlang does not copy these shared immutable structures.

10 Issues & Planned Enhancements

- is **wooper_main_loop** (in `wooper.hrl`) really tail-recursive? I think so
- integrate automatic **persistent storage** of instance states into Mnesia databases
- would there be a **more efficient** implementation of hashtables? (ex: using proplists, process dictionary, generated modules, dict or ETS-based?); more generally speaking, some profiling could be done to further increase overall performances
- even when pasting a template, having to declare all the new-related operators (ex: `new_link/N`) is a bit laborious; maybe an appropriate parse transform could do the trick and automate this declaration?
- ensure that all instances of a given class *reference* the same hashtable 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); some checking should be performed; 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 to convert a method name to the name of the module that should be called in the context of that class, inheritance-wise
- ensure that each of these references remains purely *local* to the node (no network access wanted for method look-up!); this should be the case thanks to the local WOOPER class manager; otherwise, other types of tables could be used (maybe ETS)

11 Licence

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

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

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

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

Enhancements are expected to be back-contributed, so that everyone can benefit from them.

12 Sources, Inspirations & Alternate Solutions

- **Concurrent Programming in Erlang**, Joe Armstrong, Robert Virding, Claes Wikström et Mike Williams. Chapter 18, page 299: Object-oriented Programming. This book describes a simple way of implementing multiple inheritance, without virtual table, at the expense of a (probably slow) systematic method look-up (at each method call). No specific state management is supported
- Chris Rathman's approach to life cycle management and polymorphism. Inheritance not supported
- As Burkhard Neppert suggested, an alternative way of implementing OOP here could be to use Erlang behaviours. This is the way OTP handles generic functionalities that can be specialised (e.g. `gen_server`). One approach could be to map each object-oriented base class to an Erlang **behaviour**. See some guidelines about defining your own behaviours and making them cascade
- As mentioned by Niclas Eklund, despite relying on quite different operating modes, WOOPER and Orber, an Erlang implementation of a **CORBA ORB** (*Object Request Broker*) offer similar OOP features, as CORBA IDL implies an object-oriented approach (see their OMG IDL to Erlang Mapping)

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

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

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

The WOOPER name is also a tribute to the underestimated Wargames movie (remember the WOPR, the NORAD central computer?), which the author enjoyed a lot. It is as well a second-order tribute to the *Double Wooper King Size* (*Whopper* in most if not all countries), which is/was a great hamburger indeed (in France, they are not available any more).

13 Support

Bugs, questions, remarks, patches, requests for enhancements, etc. are to be sent to the ceylan-wooper at lists dot sourceforge dot net.

One must register first.

14 For WOOPER Developers

When a new WOOPER version is released, tag the corresponding file versions, like in:

```
svn copy https://ceylan.svn.sourceforge.net/svnroot/ceylan/Ceylan/trunk/src/code/scripts/erlang
release-0.1 -m "First release (0.1) of WOOPER, already fully functional."
```

15 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!

Generated on: 2009-03-26.