Narval - Extension Programmer Handbook

Alexandre Fayolle
Olivier Cayrol
Sylvain Thénault

Narval - Extension Programmer Handbook

Alexandre Fayolle

Olivier Cayrol

Sylvain Thénault

Legal Notice

Abstract

Narval's kernel is able to manipulate plans that can perform almost any task, provided that the required actions and transformations are available. The purpose of this document is to explain how actions and transformations are written.

Revisions

Num.	Date	Author	Remarks
$Revision: 1.4 $	$Date: 2001/10/16 12:59:35 $	$Author: alf $	

Table of Content

Intended audience

This document is aimed at programmers who want to code modules for Narval. The preference language for coding modules is Python, and some knowledge of the python programming language is assumed. Having a previous experience in python programming and a good acquaintance with the python standard library definitely helps. Narval represents the data it manipulates using python objects. Usually those objects should be exportable/importable to/from XML.

Introduction to narval extensions programming

Narval is a powerful system. Yet, left alone it is as dumb as a computer with only a bare bone operating system. Modules and actions provide means for Narval to interact with the Outside World Where It's Cold. Actions can be very simple or very complex, they can do the processing themselves or act as interfaces to outside programs, enabling different proprietary pieces of software to talk to each other.

Why should you write modules? Well, basically, there are two families of reasons. The first one is that you need to do something with Narval, and nothing exists yet to do it. This situation is likely to become less frequent as time passes, but since Narval is still young it may be the case. So you have checked the repositories, asked a few questions on the mailing lists, and it looks like you have found something missing. Well in that case, it is time to sharpen your Swiss army coding knife and confront yourself with writing the missing action. This book is here to help you, and so are the various mailing lists about Narval.

The second one is that you may have written a new program, and you think that the Narval system is *soooooo* cool that you just *have* to write a module to interface your program with Narval, so that other users may pilot your program through recipes. This book is here to help you in that case too.

Modules are used to extend narval with new interfaces, new elements and new actions. Interfaces and elements are used to describe physical or logical objects that can be manipulated by the narval interpreter. Actions are used to get a specific behaviour, optionaly using input elements and creating some others as output.

In both cases, we thank you for contributing your efforts to the Narval user community.

Chapter I

Interfaces, adapters and elements

Everything in narval is based on what we call *elements*. Those elements are used to hold any information but also to describe a desired behaviour of the agent, using elements such as actions and recipes, or to control it, using elements such as start-plan, plans, etc... All of them are manipulated by the interpreter and contained in it's *memory*.

Elements are actually python class instances, and so defined in python modules and (usually automatically) registered to the interpreter. Also, for a greater flexibility, the concepts of *interface* and *adapter* are used:

- an interface is an abstract definition of available attributes and methods that should define an element to provide a logical set of data and services
- an element implements some interfaces by having the required attributes and methods. The implemented interfaces are explicitly listed
- an adapter is glue component providing a way adapt an interface to another

Interfaces and adapters, as elements, are defined by classes in python modules and registered to the interpreter.

1. Writing a new interface and adapter

Interfaces and adapter are usually defined in modules within the "interfaces" subpackage of narval. An interface must inherit (possibly indirectly) from the narval.public.Interface class, while an adapter must inherit (also possibly indirectly) from the narval.public.Adapter class.

Suppose we want to describe a URL element. This element should contains the URL address and an optional attribute giving the encoding of the file locating at the address. We also want some methods to access to the different part of the url address or to do some actions on it such as normalizing it.

This example is taken from the narval standard library. Notice that, by convention, all classes which are actually interfaces have a name beginning with a capitalized I.

```
from narval.public import Interface
class IURL(Interface):
    """interface for url elements

    :ivar address: the URL string
    :ivar encoding: optional encoding of the file located at <address>
    """
    def normalize(self):
        """return the expanded normalized url string"""
    def protocol(self):
        """return the normalized url string"""
    def path(self):
        """return the expanded normalized path string"""
```

You can notice 3 main points:

- the class inherits from Interface
- attribute are defined in the class'docstring (mainly because of a lack of the underlying interface implementation)
- methods are defined using regular python methods but with only a docstring documenting the method's aim as body

Example 1 - Defining an interface for URL object

An adapter is used to "transform" an object implementing an interface into another one implementing another interface. They are usually defined in the same module as the source or/and target interface.

The standard library contains the IURL interface as defined above, but also a IOpen interface used to describe "openable" object such as file. We can easily provide a way to open a URL element using an adapter from IURL to IOpen.

This example is taken from the narval standard library. Notice that, by convention, all classes which are actually adapters have a name using the pattern SourceInterfaceToTargetInterface.

```
from narval.public import Adapter, Interface
import urllib2
class IOpen(Interface):
    """open anything and return a file-like object
    """
    def open(self):
        """return a readable file-like object"""
class IURLToIOpen(Adapter):
    """adapt IURL to IOpen"""
    __sources__ = (IURL,)
    __implements__ = (IOpen,)

    def open(self):
        """return a file-like object from an IURL object"""
        return urllib2.urlopen(self.original.normalize())
```

You can notice 4 main points:

- the class inherits from Adapter
- source interfaces (may be multiple) are listed using the __sources__ attribute
- target interfaces (may be multiple) are listed using the __implements__ attribute
- the adapted object is stored by the base class in the *original* attribute

Example 2 - Defining an adapter from IURL to IOpen

2. Writing a new element

As you've seen in the earlier sections, interfaces are used to describe some data and services, while adapters are used to pass from an interface to another. But none of those classes defines concrete element living in the narval's memory. To do so you have to define another class, called *element* in this document.

Elements are usually defined in modules within the "elements" subpackage of narval. Every narval elements should implement an internal interface (allowing for instance automatic xml marshalling). To ease things, narval provide a base class implementing this internal interface and other facilities to make development of new elements easier, the ALElement class.

The best way is probably to start with an example. You can see below a simple element definition implementing the IURL interface:

```
from narval.public import NO_NS, normalize_url
from narval.element import NSAttribute, ALElement
from narval.interfaces.base
class URLElement(ALElement):
    """IURL implementation"""
    __implements__ = (IURL,)
    __xml_element__ = (NO_NS, 'url')
    address = NSAttribute(NO_NS, None, str, str)
    encoding = NSAttribute(NO_NS, None, str, str)
    def raw_address(self):
        return self.address

    def normalize(self):
        """return the expanded normalized url string"""
        return normalize_url(self.raw_address())[0]
    def protocol(self):
        """return the normalized url string"""
        return normalize_url(self.raw_address())[1][0] or 'file'
    def path(self):
        """return the expanded normalized path string"""
        return normalize_url(self.raw_address())[1][2]
```

You can notice here, well, many points:

- the class inherits from ALElement
- implements interfaces (may be multiple) are listed using the __implements__ attribute
- the __xml_element__ attribute is used to defined that this element will be serialized into the xml "url" element, within an empty namespace (NO_NS)
- attributes are usually defined using the NSAttribute property, which allow automatic marshalling of the attribute to and from xml. In the example for instance, we tell that the element have a "address" attribute which is in an empty xml namespace, has None as default value, and is deserializable and serializable using the same "str" function (a python builtin).

For more information you should take a look at the ALElement implementation and at existing elements in the standard library.

Example 3 - Defining an element class implementing IURL

3. Manipulating interfaces, adapters and elements

As interfaces and element classes are the base criteria to filter elements in prototypes, you need to know a few things about how to manipulate them. Notice that every interfaces and element classes name are available as identifier in the evaluation context of filter expressions.

The expression `isinstance(elmt, AClass)` will be evaluated to True if the element is an instance of the AClass class (i.e. it's an instance of this class or of a children class).

The expression `implements(elmt, IFace)` will be evaluated to True if the element is an instance of the a class implementing the IFace interface (i.e. it implements the IFace interface itself or a children interface). Notice that *implements* doesn't consider adaptation, that means that even if an adapter exists for an interface implemented by the element to the IFace interface, *implements* will be evaluated to False unless the element explicitly implements IFace.

The expression `IFace(elmt)` will be evaluated to the element itself or to an adapted element if the element implements explicitly IFace or if an adapter for an interface implemented by the element to the IFace interface exists. In other cases, this will raise an error (and so in the context of filter expression evaluation, the element will be skipped, which is the desired behaviour).

Chapter II

Writing an action

The action is the fundamental brick used by Narval to perform tasks. It is composed of two parts: an XML prototype and a python stub, both of which appear in the same file. An action is included in a python module.

Since release 1.2, Narval use xml namespaces, and so action and prototype definition should belong to a narval specific namespace: *'http://www.logilab.org/namespaces/Narval/1.2'*. Usualy this namespace is bound to the *al* prefix. This convention will be used in this document.

1. Writing a prototype

1.1. The prototype DTD

```
<!ELEMENT action (description*|input*|output*)>    (1)
<!ATTLIST action name CDATA #REQUIRED>    (2)
<!ATTLIST action func CDATA #REQUIRED>    (3)
<!ELEMENT description (#PCDATA)>    (4)
<!ATTLIST description lang CDATA #REQUIRED>
<!ELEMENT input (match*)>    (5)
<!ATTLIST input optional  (yes|no) "no">    (6)
<!ATTLIST input use       (yes|no) "no">    (7)
<!ATTLIST input list      (yes|no) "no">    (8)
<!ATTLIST input outdates  (yes|no) "no">    (9)
<!ATTLIST input from_context  (step|plan|parent_plan|memory) "step">
<!ATTLIST input to_context    (step|plan|parent_plan|memory) "memory">
<!ATTLIST input id        CDATA   #IMPLIED>     (10)
<!ELEMENT output (match*)>    (11)
<!ATTLIST output optional (yes|no) "no">    (6)
<!ATTLIST output id        CDATA   #IMPLIED>     (10)
<!ELEMENT match (#PCDATA)>    (12)
```

(1) the `action` element is used as a container for the action prototype. It holds a number of `input` and `output` child nodes.

(2) `name` is the name of the action. It is used to identify an action in a recipe. To avoid name clashes, the name of the module is prepended and used as a namespace, so the name of an action needs only to be unique within a given module.

(3) `func` is the name of the python function that implements the action stub. This function *must* be in the same module. The name is often the name of the action prefixed with `act_`.

(4) `description` elements are used to provide useful information about the action. The `lang` attribute is used to specify the language of the description. It is useful for graphical interfaces and documentation generation which can use this for localization.

(5) the `input` element describes one input of the action. This description is a list of `match` elements each of which contain an python expression. In order to be accepted for the input, an element must match all these expressions.

(6) if `optional` is set to `yes`, then the lack of element matching the `input` or the `output` will not cause an error.

(7) if `use` is set to `yes` this means that once an element has been passed as an input to the action, it will be flagged as used and not be reused by the action in subsequent evaluations of a plan instantiated from the same recipe

(8) if `list` is set to `yes`, several arguments matching the input can be passed to the action. This implies that the function stub is coded accordingly.

(8) the `from_context` maybe used to control the memory area to use for the matching elements lookup. This attribute defines the starting context: from the step, the plan, the parent_plan or the memory.

(8) the `to_context` maybe used to control the memory area to use for the matching elements lookup. This attribute defines the stoping context: to the step, the plan, the parent_plan or the memory.

(11) the `output` element describes one output of the action. This description is a list of `match` elements each of which contain an XPath. Narval considers that an action has failed if each output is not matched exactly once by one different element output by the action (with the possible exception of `optional` outputs. all these XPaths.

(9) if `outdates` is set to `yes`, once an element has been passed as an input to the action, it will be flagged as outdated, and not be reused by any action or transition in Narval. You can think of it as a super `use` attribute, that affects all the recipes, and not just the recipe that instanciated the current plan.

(10) The `id` attribute is used to provide a way for steps to alter the prototype of the action, by modifying the attributes of the input, or by adding `match` nodes to the prototype. It's also used in the action's stub to retreive arguments corresponding to a given input.

(12) A `match` contains an python expression that describes an aspect of the expected element. In this expression, the *elmt* identifier is used to represent the element matched against the expression. Notice that as inputs are evaluated in the order of their definition in the xml, the identifier of previously matched inputs will be bind to the matched element(s) and so can be used to match interdependant inputs.

1.2. Examples

Example 4 illustrates a minimal action: its takes no inputs, outputs nothing either. It could however, depending on what is in the function stub, have an effect. For example, it could be used to increment a hit counter on a web page. If no description is provided, it is not possible to tell what an action does, especially if the action name is not explicit.

```
<al:action name='NOP' func='act_NOP'/>
```

Example 4 - the NOP action prototype

Example 5 presents a typical action. A description is provided in English and in French. We notice that both inputs have a 'use' attribute: this is because we do not want to reuse the same header over and over again to produce an endless suite of identical mails. The `match` elements used in the prototype are self explanatory.

```
<al:action name='make_mail' func='act_make_mail'>
    <al:description lang='en'>Builds an email element given an element
    implementing (or adaptable to) IEmailAddress and another one
    implementing (without considering adaptation) IData.
and an email body</al:description>
    <al:description lang='fr'>Construit un élément email à partir d'un élément
implémentant (ou adaptable) l'interface IEmailAdress (adresse
électronique) et d'un élément implémentant IData (corps du message)</al:description>
    <al:input use='yes'>
        <al:match>IEmailAdress(elmt)</al:match>
    </al:input>
    <al:input use='yes'>
        <al:match>implements(elmt, IData)</al:match>
    </al:input>
    <al:output>
        <al:match>IEmail(elmt)</al:match>
    </al:output>
</al:action>
```

Example 5 - the make_mail action prototype

Example 6 shows a complex action: two out of the three input arguments are optional and the `match` for the first argument is much more elaborated than those we have seen so far. If you are not yet familiar with python, here is what it means: we are looking for a element implementing `IHTTPRequest` with at least a non null `url` attribute and a, also non null, `header` attribute.

```
<al:action name='http_get_ext' func='http_get_ext_f'>
    <al:description lang='en'>Fetches a page on the web using an optional proxy,
and optionally filtering spam out</al:description>
    <al:description lang='fr'>Ramène une page depuis le Web, en passant par un
proxy (optionnel), et en supprimant le spam (optionnel)</al:description>
    <al:input>
        <al:match>IHTTPRequest(elmt).url and IHTTPRequest(elmt).header</al:match>
    </al:input>
    <al:input optional="yes">
        <al:match>IProxy(elmt).type == 'http'</al:match>
    </al:input>
    <al:input optional="yes">
        <al:match>ISpamPolicy(elmt)</al:match>
    </al:input>
    <al:output>
        <al:match>IHTTPResponse(elmt)</al:match>
    </al:output>
</al:action>'''
```

Example 6 - the http_get_ext action prototype

2. Coding the action stub

2.1. Prototype of a stub

An action stub is a python function. Methods will not work, because there is no way to pass the object along with the call[1]. This function will be called passing one and only one argument, which will be a dictionary with input identifiers as key and matched elements for the input as value.

2.2. Retrieving inputs

If an input has the list attribute set to "yes", the value associated to the input identifier in the dictionary will be a list of matched elements. If not, it will be the matched element or None if the input is optional and has no matched element..

It's so very easy to get elements associated to each input, as shown in the example below :

Given the following action prototype:

```
<al:action name='dance-boogie-woogie' function='act_dance_boogie_woogie'>
    <al:input id='tempo'>
        <al:match>elmt.tempo</al:match>
    </al:input>
    <al:input id='dancers' list='yes'>
```

[1] Well, at least, this is what we believe. If there is a python hack that would do the trick, it's fine, but unless it is a very clean hack, we do not intend to support it.

```
        <al:match>IDancer(elmt)</al:match>
    </al:input>
    <al:input id='song' optional='yes'>
        <al:match>ISong(elmt)</al:match>
    </al:input>
</al:action>
```

We would write the following code in the stub to bind the various input elements to python identifier:

```
def act_dance_boogie_woogie(inputs):
    # access to the tempo element and get the value of its tempo
    # attribute
    tempo = inputs['tempo'].tempo
    # get dancer elements
    # as it's a non optional list, the dancers identifier will be
    # bound to a list with at least one dancer element
    dancers = inputs['dancers']
    # get the song element
    # as it's a optional element, the song identifier will be bound to
    # the song element or to None if no such element was found
    song = inputs['song']
    # do some stuff now
```

Example 7 - Retrieving inputs

2.3. Processing

Well basically, you can do anything you want here: create arbitrary elements, modify elements you got as input, read and write files on hard disk, get a web page...

2.3.1. Writing to disk

Nothing prevents you from writing anywhere on the disk, apart from Operating System restrictions.

2.4. Returning the results

Obviously, once the processing is done, we want to return a result. Narval expects to get returned a dictionary containing output elements. As for inputs, the output dictionary has output identifiers for keys and associated elements for values. The same rules as for inputs apply (i.e. list or None value according to the value of the list and optional attributes). Moreover you can omit entries for optional output without element associated.

```
<al:action name='pastry' function='act_pastry'>
    <al:output id='muffin'>
        <al:match>isinstance(elmt, muffin)</al:match>
    </al:output>
</al:action>
```

We could write the following code:

```
def act_pastry(arg)
    # no inputs to read

    # we choose the flavour of the muffin
    from random import choice
    flavour = choice(['pumpkin','raisin', 'blueberry'])
    # build the output
    muffin_elmt = muffin(flavour=flavour)
    # return the result
    return {'muffin': muffin_elmt}
```

Example 8 - building the outputs

Chapter III

Building a module

A naive way to define a module is saying that it is an action container. While this is true, there is also much more to modules than that. For one, functions in a module should have something in common, so a module is more something like an action library or a tool box. More generally, a module is a python file. Within narval, modules are used (appart for the core itself) to defines actions, but also interfaces, adapters and elements (Chapter I *“Interfaces, adapters and elements”*). Furthermore, Section 2 *“Modules considered as interfaces”* - Chapter IV will present yet another aspect of (more specifically actions') modules.

1. The Module concept

1.1. Packing actions

Building a new module is quite easy. In the python module where all the actions are declared, you must have a global variable called `MOD_XML` that contains all the XML declaration for the module and the actions. The usual way to do this is by initializing the variable at the beginning of the file and building incrementally as function stubs are declared:

```
from narval.public import AL_NS
MOD_XML = "<module xmlns:al='%s'>" % AL_NS
##
## Http Get Ext
##
def act_http_get_ext(args):
    pass
    # function code intentionally skipped
MOD_XML = MOD_XML+'''
<al:action name='http_get_ext' func='act_http_get_ext'>
    <al:input>
        <al:match>IHTTPRequest(elmt).url and IHTTPRequest(elmt).header</al:match>
    </al:input>
    <al:input optional="yes">
        <al:match>IProxy(elmt).type == 'http'</al:match>
    </al:input>
    <al:input optional="yes">
        <al:match>ISpamPolicy(elmt)</al:match>
    </al:input>
    <al:output>
        <al:match>IHTTPResponse(elmt)</al:match>
    </al:output>
</al:action>'''
##
## Write back to socket
##
def Write_back_to_socket_f(args) :
    pass
    # function code intentionally skipped
MOD_XML=MOD_XML+'''
<al:action name='Write_back_to_socket' func='Write_back_to_socket_f'>
  <al:description lang="en">Send back response to client</al:description>
  <al:description lang="fr">Renvoie la réponse au client</al:description>
  <al:input use="yes">
    <al:match>IHTTPResponse(elmt)</al:match>
  </al:input>
  <al:input>
    <al:match>isinstance(elmt, socket)</al:match>
  </al:input>
</al:action>'''
MOD_XML=MOD_XML+'</module>'
```

This makes it very easy to add new functions to a module, since you only have to add the code in the file and add the action prototype to `MOD_XML` (MOD_XML is a special identifier that is used at actions'modules load time to extract available actions in the module, so you can't use it for another purpose or use a different identifier to hold actions'prototypes definitions).

1.2. What makes a good candidate for an action?

Choosing what to put in an action is difficult. This is really the same challenge as designing a software library. With Narval however, a new factor comes in play. Actions can be used in recipes. A typical recipe should use from three to a dozen actions to perform its task, so actions should not have a too small granularity. Actions are supposed to perform elementary tasks at the scale of the recipe, which itself is very high level, so actions are already quite high level. A typical recipe will for instance manage an address book, so the required actions for such a recipe would be adding or removing an address. Proposing an action to read the address book from hard disk is too fine grained.

Note

Remark

As always, everything is a matter of context, there may be recipes which require reading files from disk, and an action that does just that is provided in the standard distribution of Narval.

This means that something that can be a good candidate for an action in a module can be a bad one in another module. Writing test recipes is a good way to tell if the actions in a module are too low-level: if you get the impression that you are writing a program in a programing language like Python, Java or Younameit, then you probably got it wrong. Narval's ultimate goal is to bring the Power of computers in the hand of the average person in the street (well, to be honest we are still far away from that, so the "ultimate"...)[2], so using your actions to write a recipe should not become something like programming. When you add a new action to a module, always ask yourself whether you would really like to have it in a recipe. It is much better to share code between the implementation of action stubs than to add an action that will have to be inserted in every recipe that uses actions from the module.

1.3. Considering a actions'module as a unit

Deciding to pack actions in modules is one thing, deciding *how* to pack them is another one. The logical decision is to group them around a common theme. Since all actions in a module share a common file [3], this encourages sharing utility code between action stubs. In other words, all function within a module file need not be action stubs. There can be any number of helper functions provided in a module to

[2] and when we say *Power*, we mean *Real* Power, and not just surfing the web and downloading WaReZ and thinking we are now 31337 hackers. Computers are good because they can save time by doing boring stuff for you. Using a word processor is not a progress over using a type machine if you have to open each of the 456 files in your directory to *manually* change the logo of your company on the first page. Most people who use computers nowadays will have to do it that way, though. Narval should enable them to quickly write a recipe that will do this automatically.

[3] This is the case right now. It might be possible to build modules that would be stored in directories, with the `MOD_XML` variable initialized in the `__init__.py` file. It has not been tested yet, but if it is possible, we shall provide support for this in a future release

avoid code duplication in action stubs, and ease the coding of new actions.

Each module may want to deal with it's own set of elements and / or interface. Before introducing a new element or interface, you should carefully consider existing ones and see if one of them would be fine for what you need. If not so, you should write a python class for the element / interface (or event an adapter).

2. Testing strategies

Before trying to test your module with Narval, you should perform some unitary testing, that will enable you to check that your module will behave as expected, or at least that it will not crash in a stupid way when loaded. This section introduces some testing techniques that isolates the module from Narval and thus make it easier to find some bugs. They especially enable the use of a python debugger and other standard debugging methods, which are rather difficult to set up when Narval is running.

2.1. Checking the XML syntax

One of the first thing you want to check in a module is that the prototypes of the actions are syntactically correct, since this will prevent Narval from being able to load the module. This is done by adding a python `main` function that will parse the string held by `MOD_XML`:

```
if __name__ == "__main__" :
    print MOD_XML
    from xml.dom.ext.reader import Sax2
    doc=Sax2.FromXml(MOD_XML)
    print doc.documentElement
```

Once this is done, you can run your module from the command line as you would for any other program. This can bring up two kinds of errors:

- Python syntax errors, which you are presumably familiar with. These are beyond the scope of this document
- XML syntax errors. The exception you will get will tell you where the error occurred. Common errors include:
  - missing '/' in the closing tag of an element;
  - missing '/' before the '>' of an empty element;
  - quoting mismatch in attributes;
  - typos that cause an opened element not to be closed.

2.2. Testing the actions individually

Unfortunately, it is not possible to test all actions outside of Narval. If an action uses the socket forwarder, for instance, it will not be possible to test it if nothing is listening on the socket forwarder port, for instance. Similarly, if a group of actions are very tightly coupled, for instance if they share a common object and behave differently according to some internal state of the object, testing will be difficult.

The proposed method for testing is to call the stub of the action by passing it elements as Narval would do it. The outputs of the method can be retrieved and compared to what is expected, and thus the action is validated.

2.3. The test framework

For actions which can't be easily unittested, or to test how actions go together, or to test recipes, narval comes with a test framework. The principle is simple: launch narval with a memory file describing a recipe and starting it, make it stop when all plans are terminated, and then check the narval's memory after execution. You've so to write the initial memory file and a memory validation file. You can get more information about this in the "narval testing how-to" document.

2.4. The Big Game

Well, obviously, when you created your actions, you had some precise idea about how they should be used in recipes. The time has now come to write test recipes, and run them. The Recipe Coding Manual is here to help you, and so is the Horn User Manual.

3. Releasing

Writing modules is a Good Thing. Making them available for everyone is a Better Thing. So now that your module is coded and tested, now that you have sample recipes illustrating how to use your actions, it's release time!

3.1. Documentation

Maybe you have so far coded for your eyes only. You know your code, how it works, and that's good. However, maybe a bit of tidying would be welcome.

3.1.1. Documenting action prototypes

We have seen in Section 1.1 “*The prototype DTD*” - Chapter II that the `description` is optional. It is *strongly* recommended that you should use it for every action you write. This is one of the three indications a recipe programmer will have about what your action does, the other two being the module name and the action name. As the module and the action name are one or two words, this leaves only the description for something a bit more consistent. Keep in mind that due to screen space limitations, it is best to keep the description string as a one-liner. Avoid if possible repeating what can be guessed by looking at the XML prototype, and rather elaborate about the *action* that takes place to transform the inputs into outputs.

3.1.2. Documenting stubs

The standard python documentation advices apply here. Use doc strings wherever applies, use comments where needed. We believe at Logilab that it is worth spending a *lot* of time on the code so that it can be understood without using too many comments. This includes using good variable names and good function names, rewriting shaggy code again and again until it becomes clean, using standard Design Patterns and naming the objects accordingly. We encourage you to do the same, for the benefit of everyone.

3.2. Licensing

We have chosen to release Narval under LGPL. We do not wish to impose anything on the developer community about the modules they contribute, so you are free to distribute your modules under the license you wish. However, we encourage you to use a well known LGPL-compatible license. This will enable us to redistribute your modules in future Narval releases, and to make it available on our web site without worrying about possible legal problems. Please include a license statement with the modules you release.

Chapter IV

Going further

1. Multiple actions for a single function

There is no obligation of having a one to one correspondence between actions and stubs. It is perfectly acceptable to have a single stub that would behave differently according to the inputs is received. A typical example would be an action that acts as a proxy to some outer program using the same interface provided by the program. One could argue that it could be possible to have an single action with `optional` inputs that would be associated with the stub. This is true, but would nevertheless not be a good idea, because it would make recipes much less easy to read, whereas providing several actions with distinct and clear names can make things much easier to understand.

2. Modules considered as interfaces

Something great about modules is that they can behave as interfaces to programs, and as any OO programmer will tell you, Interfaces are Good Things. For instance, it is possible to identify the basic requirements for a mail module. However, the implementation of the stubs depend on the underlying operating system: under Unix, mail is often read in `/var/spool/mail` or another system mailbox, whereas Windows users generally use a POP3 or IMAP server. Yet, from an action point of view, they all receive and send mails, sometimes with attached documents, and that's about it. Once the module interface has been defined, it is possible to choose which implementation of the module should be installed on a given system, and all the recipes will keep on working *regardless of the implementation*.