Introduction to Waf 1.6 (draft)

Contents

Introduction

Waf 1.5 limitations

The Waf build system was first created to provide an alternative for the autotool-based build systems based on a single programming language. To ease the development, a few assumptions were made, and over the time it became clear that they had become a limitation for specific corner cases, among others:

- in some projects, files are generated in the source directory and are kept in the source control system
- there may be more than one kind of build command
- some configuration API may be useful in the build section
- it may be interesting to use Waf as a library in another system
- some API was private (node objects)

Goals of Waf 1.6

The main goals of Waf 1.6 are:

1. to remove most assumptions/restrictions present in Waf 1.5 (nodes, build directory, reduce the coupling between the modules)
2. to optimize the code base (execution time, source code size, compatibility with different python versions)
3. to improve the user experience (make the API more intuitive, expose more classes and functions, improve the error handling)

Purpose of this document

The Waf API (modules, classes. functions) had to be changed to make the changes described previously. This document describes the main changes compared to the previous version, Waf 1.5. The Waf book for Waf 1.6 should be the reference for new users and for projects upgrading from previous Waf versions such as Waf 1.4.

1 Main changes in user scripts

1.1 The attributes *srcdir* and *blddir*

The attributes *srcdir* and *blddir* have been renamed to *top* and *out* respectively:

```
#srcdir = '.' # old
#blddir = 'build' # old
top = '.'
out = 'build'

def configure(ctx):
    pass
```

1.2 The command *options*

The method set_options has been renamed to *options*:

```
top = '.'
out = 'build'

def options(opt):
        pass

def configure(ctx):
        pass
```

1.3 The method *recurse*

The context class for all commands has a unique method named *recurse*. The methods *add_subdirs*, *sub_config* and *sub_options* have been removed:

```
def options(ctx):
        #ctx.sub_options('src') # old
        ctx.recurse('src')

def configure(ctx):
        #ctx.sub_config('src') # old
        ctx.recurse('src')

def build(ctx):
        #ctx.add_subdirs('src') # old
        ctx.recurse('src')
```

1.4 Tool declaration

For consistency with the user scripts, the method *detect* in Waf tools has been renamed to *configure*. The method *set_options* is now called *options* and is consistent with Waf scripts.

1.5 Task generator declaration

The build context method *new_task_gen* disappears, and is replaced by call. The task generator subclasses have been removed completely, so only keyword arguments are accepted. For example:

```
top = '.'
out = 'build'

def configure(ctx):
        pass

def build(ctx):
        # ctx.new_task_gen('cc', 'cprogram', source='main.c', target='app') # old
        ctx(features='c cprogram', source='main.c', target='app')
```

2 Waf modules refactoring

2.1 Syntax and namespaces

Python 3 is now the default syntax. Waf now runs unmodified for 2.6, 2.7, 3.0 and 3.1. Upon execution, a detection is performed and the code is then converted for python 2.3, 2.4 and 2.5 if necessary. It is far easier to modify Python3 code to run on Python2 than porting Python 2 code to Python 3.

The Waf tools and modules have been reorganized in the following directory structure:

```
waflib
|- extras
`- Tools
```

To avoid conflicts with other projects, python namespaces are now used in the Waf library. This means that the import system has changed:

```
top = '.'
out = 'build'

def configure(ctx):
        # import Options, Utils # old
        from waflib import Options, Utils

        print(Options.options)
        print(Utils.to_list("a b c"))
```

2.2 Core modules added and removed

The following modules have been removed:

- py3fixes: the code uses the python 3 syntax by default
- Constants.py: the constants have been moved to the modules that use them
- pproc.py: the system subprocess module is now used by default

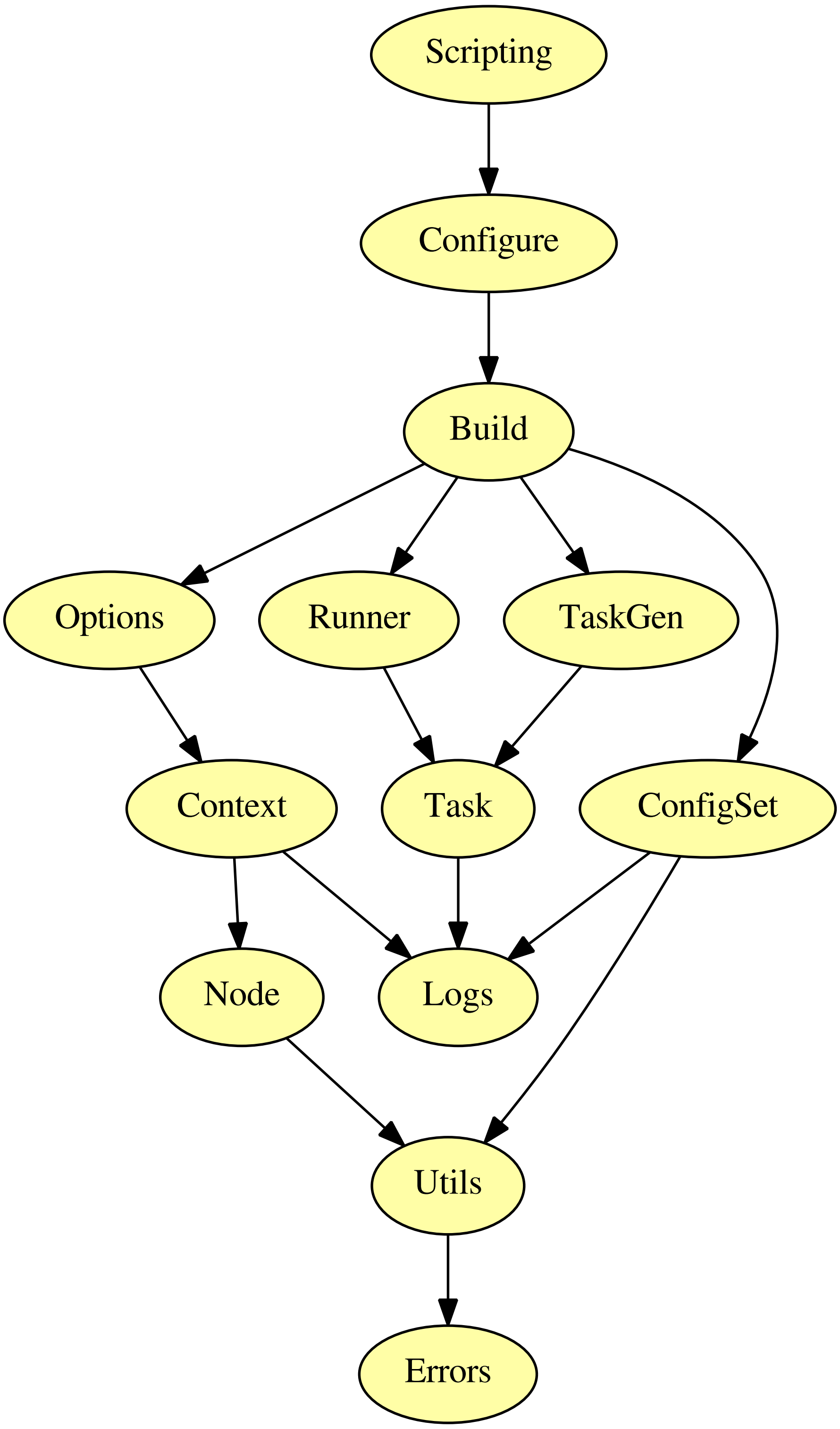
The following modules have been added:

- Context.py: base class for Waf commands (build context, configuration context, etc)
- Errors.py: exceptions used in the Waf code
- fixpy2.py: routines to make the Waf code executable on python 2.3, 2.4 and 2.6

The module Environment.py has been renamed to ConfigSet.py

2.3 Module dependencies

The core modules have been refactored to remove the cyclic imports. The following diagram represents the global dependencies between the core modules. The non-essential dependencies - such as the module *Scripting* using the module *Utils* are hidden.



Dependencies between the core modules

3 New context classes

3.1 Context subclasses

The following will create a build context subclass that will be instantiated when calling *waf build*. It will execute the user function *build* from the scripts.

```
from waflib.Build import BuildContext
class debug_context(BuildContext):
    cmd = 'debug'
    fun = 'build'
```

3.2 Context subclasses and nodes

Nodes objects are now accessible from the command contexts.

```
from waflib.Context import Context
class package_class(Context):
    cmd = 'package'
    fun = 'package'

def package(ctx):
    print(ctx.path.ant_glob('wscript'))
```

The output will be:

```
$ waf package
[/disk/comp/waf-1.6/demos/c/wscript]
'package' finished successfully (0.006s)
```

3.3 New variant system

The variant system present in the previous Waf versions has been removed. The new system makes it easier to declare an output directory for the tasks:

```
top = '.'
out = 'build'

def configure(ctx):
    ctx.load('gcc')

def build(ctx):
    tg = ctx(features='c', source='main.c')
    if ctx.cmd == 'debug':
        tg.cflags = ['-g']

from waflib.Build import BuildContext, CleanContext
class debug_build(BuildContext):
    cmd = 'debug'
    variant = 'debug_'

class debug_clean(CleanContext):
    cmd = 'clean_debug'
    variant = 'debug_'
```

The outputs from the default build will be written to *build/* but the outputs from the debug one will go into *build/debug_*:

```
waf clean clean_debug build debug -v
'clean' finished successfully (0.034s)
'clean_debug' finished successfully (0.007s)
Waf: Entering directory `/disk/comp/waf-1.6/demos/c/gnu/build'
[1/1] c: main.c -> build/main.c.0.o
00:36:41 runner ['/usr/bin/gcc', '../main.c', '-c', '-o', 'main.c.0.o']
Waf: Leaving directory `/disk/comp/waf-1.6/demos/c/gnu/build'
'build' finished successfully (0.075s)
Waf: Entering directory `/disk/comp/waf-1.6/demos/c/gnu/build/debug_'
[1/1] c: main.c -> build/debug_/main.c.0.o
00:36:41 runner ['/usr/bin/gcc', '-g', '../../main.c', '-c', '-o', 'main.c.0.o']
Waf: Leaving directory `/disk/comp/waf-1.6/demos/c/gnu/build/debug_'
'debug' finished successfully (0.082s)
```

4 The *Node* class

4.1 New node class design

The node objects are used to to represent files or and folders. In Waf 1.6 the build folders are no more virtual and do not depend on the variant anymore. They are represented by exactly one node:

```
top = '.'
out = 'build'

def configure(ctx):
    pass

def build(ctx):
        #print(ctx.path.abspath(ctx.env)) # old
        print(ctx.path.get_bld().abspath()) # Waf 1.6
```

The method *ant_glob* no longer accepts the *bld* argument to enumerate the build files, and must be called with the appropriate node:

```
top = '.'
out = 'build'

def configure(ctx):
    pass

def build(ctx):
    # ctx.path.ant_glob('*.o', src=False, bld=True, dir=False) # old
    ctx.path.get_bld().ant_glob('*.o', src=True, dir=False) # equivalent in Waf 1.6
```

The method *ant_glob* is now the default for finding files and folders. As a consequence, the methods *Node.find_iter*, *task_gen.find_source*
and the glob functionality in *bld.install_files* have been removed.

4.2 Targets in the source directory or in any location

The build may now update files under the source directory (versioned files). In this case, the task signature may not be used as node signature (source files signatures are computed from a hash of the file). Here is an example:

```
top = '.'
out = 'build'

def configure(ctx):
        pass

def build(ctx):
        # create the node next to the current wscript file - no build directory
        node = ctx.path.make_node('foo.txt')
        ctx(rule='touch ${TARGET}', always=True, update_outputs=True, target=node)
```

For nodes present out of the build directory, the node signature must be set to the file hash immediately after the file is created or modified (not the task signature set by default):

```
import Task

@Task.update_outputs
class copy(Task.Task):
        run_str = '${CP} ${SRC} ${TGT}'
```

Although discouraged, it is even possible to avoid the use of a build directory:

```
top = '.'
out = '.'

def configure(ctx):
    pass
```

4.3 Use nodes in various attributes

Nodes are now accepted in the task generator source and target attributes

```
def configure(ctx):
    pass

def build(ctx):
    src = ctx.path.find_resource('foo.a')
    tgt = ctx.path.find_or_declare('foo.b')

    ctx(rule='cp ${SRC} ${TGT}', source=[src], target=[tgt])
```

In the case of includes, the node may be used to control exactly the include path to add:

```
def configure(ctx):
    ctx.load('gcc')

def build(ctx):
        ctx(features='c', source='main.c', includes=[ctx.path])
        ctx(features='c', source='main.c', includes=[ctx.path.get_bld()])
```

The output will be:

```
$ waf -v
Waf: Entering directory `/foo/test/build'
[1/2] c: main.c -> build/main.c.0.o
03:01:20 runner ['/usr/bin/gcc', '-I/foo/test', '../main.c', '-c', '-o', 'main.c.0.o']
[2/2] c: main.c -> build/main.c.1.o
03:01:20 runner ['/usr/bin/gcc', '-I/foo/test/bld', '../main.c', '-c', '-o', 'main.c.1.o']
Waf: Leaving directory `/foo/test/build'
'build' finished successfully (0.476s)
```

Because node objects are accepted directly, the attribute *allnodes* has been removed, and the nodes should be added to the list of source directly:

```
from waflib.TaskGen import extension
@extension('.c.in')
def some_method(self, node):
        out_node = node.change_ext('.c')
        task = self.create_task('in2c', node, out_node)

        #self.allnodes.append(out_node) # old
        self.source.append(out_node)
```

5 c/cxx/d/fortran/assembly/go

The support for the languages *c*, *cxx*, *d*, *fortran*, *assembly*, and *go* is now based on 4 methods executed in the following order: *apply_link*, *apply_uselib_local*, *propagate_uselib_vars* and *apply_incpaths*.

5.1 apply_link

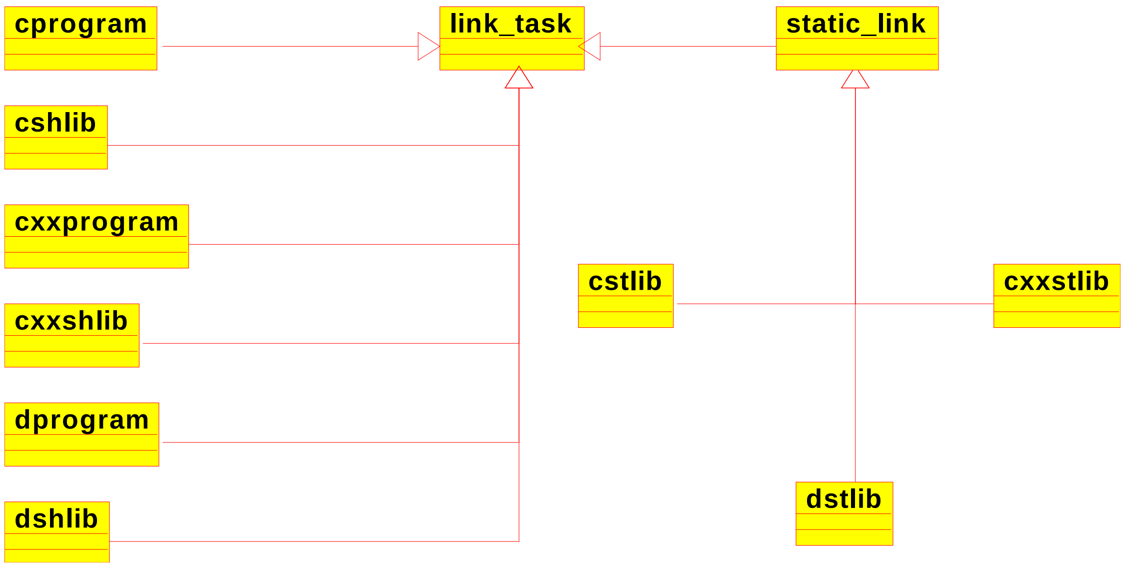
The task generator method apply_link scans the task generator attribute for the feature names and tries to find a corresponding task class, if possible. If a task class is found and has the attribute *inst_to* set, it will be used to create a link task. The tasks stored in the attribute *compiled_tasks* will be used to set the input object files (.o) to the link task.

The following classes are now used and match the names of the task generator attribute *features*: cprogram, cshlib, cstlib, cxxprogram, cxxshlib, cxxstlib, dprogram, dshlib and dstlib.

The method *create_compiled_tasks* should be used to create tasks that provide *.o* files, as it adds the tasks to the attribute *compiled_tasks*, for use with the method *apply_link*. This is of course optional.

5.2 apply_uselib_local

The method apply_uselib_local has been changed to take into account static and shared library types. Two base classes for link tasks have been added in the module *waflib.Tools.ccroot*: *link_task* and *stlink_task*. The names for the subclasses correspond to *features* names to use for the method *apply_link* from the previous section:



5.3 propagate_uselib_vars

The uselib attribute processing has been split from the local library processing. The main operations are the following:

1. the variables names are extracted from the task generator attribute *features*, for example *CFLAGS* and *LINKFLAGS*
2. for each variable, the task generator attributes in lowercase are processed, for example the contents of *tg.cflags* is added to *tg.env.CFLAGS*
3. for each feature, the corresponding variables are added if they are set, for example, if *tg.features = "c cprogram"* is set, then *c_CFLAGS* and *cshlib_CFLAGS* are added to *tg.env.CFLAGS*
4. for each variable defined in the task generator attribute *uselib*, add the corresponding variable, for example, if *tg.uselib = "M"* is set, then *tg.env.CFLAGS_M* is added to *tg.env.CFLAGS*

The variable names are defined in *waflib.Tools.ccroot.USELIB_VARS* and are bound to the feature names.

5.4 apply_incpaths

The task generator method apply_incpaths no longer processes the attribute *CPPPATH* but the attribute *INCLUDES* which is consistent with the task generator attribute *tg.includes*. After execution, two variables are set:

1. *tg.env.INCPATHS* contains the list of paths to add to the command-line (without the *-I* flags)
2. *tg.includes_nodes* contains the list of paths as node objects for use by other methods or scanners

5.5 New wrappers for programs and libraries

For convenience, the following wrappers *program*, *shlib* and *stlib* have been added. The following declarations are equivalent:

```
def build(ctx):
        ctx(features='cxx cxxprogram', source='main.cpp', target='app1')
        ctx.program(source='main.cpp', target='app2')

        ctx(features='c cstlib', source='main.cpp', target='app3')
        ctx.stlib(source='main.cpp', target='app4')
```

6 New features

6.1 Task dependencies

The method *BuildContext.use_the_magic* disappears, and a heuristic based on the task input and outputs is used to set the compilation order over the tasks. For compilation chains and indirect dependencies it will still be necessary to provide the extension symbols, but this will be the exception. For example:

```
top = '.'
out = 'build'

def configure(ctx):
        ctx.load('gcc')

def build(ctx):
        ctx(rule='touch ${TGT}', target='foo.h', ext_out=['.h'])
        ctx(features='c cprogram', source='main.c', target='app')
```

In this case, the *.h* extension is used to indicate that the unique task produced by this rule-based task generator must be processed before the c/cxx compilation (the c and cxx tasks have ext_in=[*.h*]).

6.2 Configuration API

The configuration methods bound to the ConfigurationContext by the decorator *@conf* are now bound to the BuildContext class too. This means that the configuration API may be used during the build phase.

```
def options(ctx):
    ctx.load('compiler_cxx')

def configure(ctx):
    ctx.load('compiler_cxx')
    ctx.check(header_name='stdio.h', features='cxx cxxprogram')

def build(bld):
        bld.program(source='main.cpp', target='app')
        if bld.cmd != 'clean':
                bld.check(header_name='sadlib.h', features='cxx cprogram')
```

To redirect the output, a logger (from the python logging module) must be provided. Here is an example demonstrating a convenience method for creating the logger:

```
def options(ctx):
    ctx.load('compiler_cxx')

def configure(ctx):
    ctx.load('compiler_cxx')
    ctx.check(header_name='stdio.h', features='cxx cxxprogram')

def build(bld):
        if bld.cmd != 'clean':
                from waflib import Logs
                bld.logger = Logs.make_logger('test.log', 'build')
                bld.check(header_name='sadlib.h', features='cxx cprogram')
                bld.logger = None
```

Note that the logger must be removed after the configuration API is used, else the build output will be redirected as well.

6.3 Downloading Waf tools

The call to *conf.load* will normally fail if a tool cannot be found.

```
top = '.'
out = 'build'

def configure(conf):
        conf.load('swig')
```

By configuring with the option *--download*, it is possible to go and download the missing tools from a remote repository.

```
$ ./waf configure --download
Checking for Setting top to                : /comp/test
Checking for Setting out to                : /comp/test/build
http://waf.googlecode.com/svn//branches/waf-1.6/waflib/extras/swig.py
downloaded swig from http://waf.googlecode.com/svn//branches/waf-1.6/waflib/extras/swig.py
Checking for Checking for program swig     : /usr/bin/swig
'configure' finished successfully (0.408s)
```

The list of remote and locations may be configured from the user scripts. It is also recommended to change the function *Configure.download_check* to check the downloaded tool from a white list and to avoid potential security issues. This may be unnecessary on local networks.

```
top = '.'
out = 'build'

from waflib import Configure, Utils, Context

Context.remote_repo = ['http://waf.googlecode.com/svn/']
Context.remote_locs = ['branches/waf1.6/waflib/extras']

def check(path):
        if not Utils.h_file(path) in whitelist:
                raise ValueError('corrupt file, try again')
Configure.download_check = check

def configure(conf):
    conf.load('swig')
```

6.4 The command *list*

A new command named *list* is used to list the valid targets (task generators) for "waf build --targets=x"

```
$ waf list
foo.exe
my_shared_lib
my_static_lib
test_shared_link
test_static_link
'list' finished successfully (0.029s)

$ waf clean --targets=foo.exe
Waf: Entering directory `/disk/comp/waf-1.6/demos/c/build'
[1/2] c: program/main.c -> build/program/main.c.0.o
[2/2] cprogram: build/program/main.c.0.o -> build/foo.exe
Waf: Leaving directory `/disk/comp/waf-1.6/demos/c/build'
'build' finished successfully (0.154s)
```

6.5 The command *step*

The new command *step* is used to execute specific tasks and to return the exit status or any error message. It is particularly useful for debugging:

```
$ waf clean step --file=test_shlib.c
'clean' finished successfully (0.017s)
Waf: Entering directory `/disk/comp/waf-1.6/demos/c/build'
c: shlib/test_shlib.c -> build/shlib/test_shlib.c.1.o
 -> 0
cshlib: build/shlib/test_shlib.c.1.o -> build/shlib/libmy_shared_lib.so
 -> 0
Waf: Leaving directory `/disk/comp/waf-1.6/demos/c/build'
'step' finished successfully (0.146s)
```

To restrict the tasks to execute, just prefix the names by *in:* or *out:*:

```
$ waf step --file=out:build/shlib/test_shlib.c.1.o
Waf: Entering directory `/disk/comp/waf-1.6/demos/c/build'
c: shlib/test_shlib.c -> build/shlib/test_shlib.c.1.o
 -> 0
Waf: Leaving directory `/disk/comp/waf-1.6/demos/c/build'
'step' finished successfully (0.091s)
```

7 Compatibility layer

The module *waflib.extras.compat15* provides a compatibility layer so that existing project scripts should not require too many modifications.